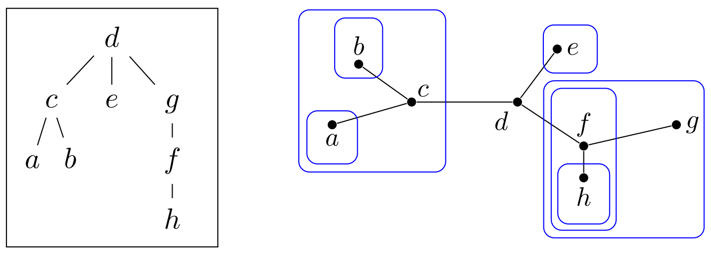
Search trees on graphs



Dissertation
zur Erlangung des Grades eines Doktors der Naturwissenschaften

vorgelegt am
Fachbereich Mathematik und Informatik
der Freien Universität Berlin

von
Benjamin Aram Berendsohn

Berlin, 2024

Betreuer und Erstgutachter:	Prof. Dr. László Kozma
Zweitgutachter:	Prof. Dr. Jean Cardinal
Datum der Disputation:	18.11.2024

Selbstständigkeitserklärung

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Dissertation selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht. Diese Dissertation wurde in gleicher oder ähnlicher Form noch in keinem früheren Promotionsverfahren eingereicht.

Mit einer Prüfung meiner Arbeit durch ein Plagiatsprüfungsprogramm erkläre ich mich einverstanden.

Berlin, den ____________

____
Benjamin Aram Berendsohn

Abstract

Search trees on graphs (STGs) are a far-reaching generalization of binary search trees (BSTs), where the key space is a *graph* instead of a totally ordered set. Intuitively, instead of comparing *keys*, we "compare" *vertices*, which tells us their relative position in the graph.

STGs can alternatively be seen as (1) a data structure for vertex searching that can be built on top of certain graphs that support vertex comparisons (e.g., *quadtrees*) or (2) as a hierarchical decomposition of the underlying graph. For the latter purpose, search trees have appeared in the literature under different names (e.g., *elimination trees* and *spines*), and there are strong connections to the ubiquitous notions of *tree-depth* and *tree-width.*

Many computational and combinatorial questions about BSTs naturally generalize to the STG setting. In this thesis, we concentrate on the following three.

- Suppose we are given a query distribution and want to compute the search tree that minimizes the expected search time. This is the *optimal static search tree problem.* Computing optimal static BSTs is possible in quadratic time, but for STGs no polynomial-time algorithm is known, even when the underlying graph is a tree. We discuss multiple approximation algorithms for trees and graphs of bounded tree-width, as well as NP-hardness of the problem in the general case.

- In contrast to static search trees, which cannot be modified after construction, *dynamic search trees* may be restructured after each search using the *rotation* operation. We generalize the well-known *Splay* algorithm from BSTs to STGs in the special case where the underlying graph is a tree. We also use that algorithm for a *dynamic forest data structure*, including an implementation and its experimental evaluation.

- Finally, the *rotation distance* between two STGs is the minimum number of rotations needed to transform one into the other. We are interested in the *maximum* rotation distance for a given underlying graph, which is exactly the diameter of the so-called *graph associahedron* of the underlying graph. Somewhat surprisingly, there is a tight connection between this combinatorial problem and the static and dynamic search tree problems. We present several new results, including a simple algorithm to compute the diameter of a graph associahedron when the underlying graph is a tree of bounded *path-width.*

Acknowledgements

First and foremost, I would like to thank my advisor László Kozma. During the last five years, László was always ready to help, advise, and discuss. He frequently suggested interesting new problems to me – this includes most of the problems studied in this thesis. He was always supportive in all my scientific endeavors, and it was a joy working with him.

I would further like to thank László as well as Alexander Baumann, Yoël Berendsohn, Michaela Borzechowski, and Alireza Selahi for agreeing to read drafts of this thesis and providing many helpful comments and suggestions. Last, but not least, I would like to thank Jean Cardinal for agreeing to review the thesis, and my friends and family for their support.

Notes on collaboration

This thesis is based mainly on the following four published papers.

- *Fast Approximation of Search Trees on Trees with Centroid Trees* [BGKK23], joint work with Ishay Golinsky, Haim Kaplan, and László Kozma. Chapter 4 mostly consists of content from this paper, with the exception of sections 4.4 to 4.6.
- *Splay Trees on Trees* [BK22], joint work with László Kozma. Chapters 5, 9 and 13 are partly based on this paper, though all three have been significantly revised and contain new results.
- *Fast and simple unrooted dynamic forests* [Ber24]. Chapter 10 consists almost entirely of this paper, with only minor additions and revisions. Some parts have been moved to chapter 9.
- *The Diameter of Caterpillar Associahedra* [Ber22]. Chapter 12 builds upon content from this paper, but significantly expands it.

Chapters 6 and 7 are entirely new. The remaining chapters contain mainly informal exposition, summaries, preliminaries (partly taken from the aforementioned papers), and some minor new observations and results.

Contents

1. Introduction

A *dictionary* data structure (also called *associative array* or *map*) stores a set of keys and potentially some associated data. Given a key, the dictionary reports whether the key is in the set and provides the associated data, if any. Usually, keys may also be added and deleted, and their associated data may be changed.

Dictionaries are everywhere; most programming languages provide one or more highly optimized implementations in their standard libraries.[1] When the key space is unstructured and keys are small, then usually *hash tables* are the best choice [Knu73].

However, to store keys from a totally ordered domain, *binary search trees* (BSTs) are a standard approach.[2] They do not require keys to be efficiently hashable (just efficiently comparable), and support certain additional operations like finding the predecessor of a given key.

In this work, we consider *search trees on graphs* (STGs), a generalization of BSTs that support key domains with a more general structure than total orders; namely, the domain is modeled as a *graph*. Aside from the direct use as data structure for searching in graphs (this will be made precise in the next section), search trees on graphs are also useful as *hierarchical decompositions* of the underlying graphs.

1.1. Searching in trees and graphs

We start by briefly describing the underlying model and motivation for binary search trees. Then, we present notions of *searching in a tree* and, more generally, *searching in a graph*. This allows us to derive an analogue of BSTs, the aforementioned *search trees on graphs*. For, now we focus on the basic *search* operation that finds a key in the stored set; in particular, we omit operations that change the underlying set by adding or deleting keys, and we do not associate values to keys.

Searching in total orders. Suppose we have a finite totally ordered set V of *known* values and want to identify a given *unknown* element x, i.e., find an element $v \in V$ with $x = v$. We are allowed to compare x to any element $v \in V$, which reports whether $x < v$, $x = v$, or $x > v$. This rules out all values greater than x, or all values smaller than v, or both. Repeated comparison narrows down the remaining set of candidates, until we find the element $v \in V$ with $x = v$ or we determine that V does not contain x.

[1]For example, `map` and `unordered_map` in the C++ standard library [ISO23]; `HashMap` and `BTreeMap` in Rust [Rus]; `HashMap`, `TreeMap`, `ConcurrentSkipListMap`, and others in Java [Ora24]; and `dict` in Python [Pyt24].

[2]For example, Java's `TreeMap` [Ora24] and the `map` implementation of gcc's libstc++ [GCC] are binary search trees.

The standard approach is to use binary search, i.e., always compare to the median of the remaining elements. This requires $\mathcal{O}(\log |V|)$ comparisons in the worst case, which is optimal [Knu73].

However, in certain situations, a different strategy may be preferable. Here, by *strategy* we mean a function that decides which comparison to perform based on previous comparisons and their results. Strategies correspond to *decision trees*, and decision trees in turn correspond to *binary search trees* (as long as no redundant comparisons are made). See figure 1.1 for an example.

Let us assume that $x \in V$. The number of comparisons required to find x in a binary search tree is the *depth* of the corresponding node in the BST.[3] Thus, if we know that a certain element is frequently searched, we may want to put that element close to the root of the BST.

Searching in trees. Suppose we have a tree[4] G and want to "find" an unknown vertex x. Instead of performing comparisons, we may *query* a vertex v, which either reveals that $v = x$, or reports the first edge on the path from v to x. Our goal is to eventually query a vertex v that reports $v = x$. Note that x is guaranteed to be contained in G, so unsuccessful searches like for BSTs are not possible.

Observe that each query narrows down the remaining candidates to a *connected subgraph* of G. By continuing to query vertices that are not ruled out yet, we will find x eventually. In particular, if G is a path, then we essentially recover searching in total orders: Fix a left-to-right direction on the path, and interpret the outcome of a query (left edge, right edge, or equality) as "smaller", "greater", or "equal".

As before, different *strategies* (i.e., decision trees) are possible. We represent a non-redundant strategy as a rooted tree called *search tree on a tree* (STT). More precisely, we define a *search tree* T on a tree G as a rooted tree constructed as follows. Pick some vertex r of G as the root of T. Then, for each component C of $G - r$, recursively construct

[3]Throughout the thesis, the depth of a node in a rooted tree is the number of non-strict ancestors, including the node itself. In particular, the depth of the root is one.

[4]Here, a tree is an acyclic and connected undirected graph; trees in the data structure sense (like BSTs) are always called *rooted trees* or *search trees* in this thesis.

"First, compare x to b. If $x < b$, then compare x to a. Otherwise, compare x to c."

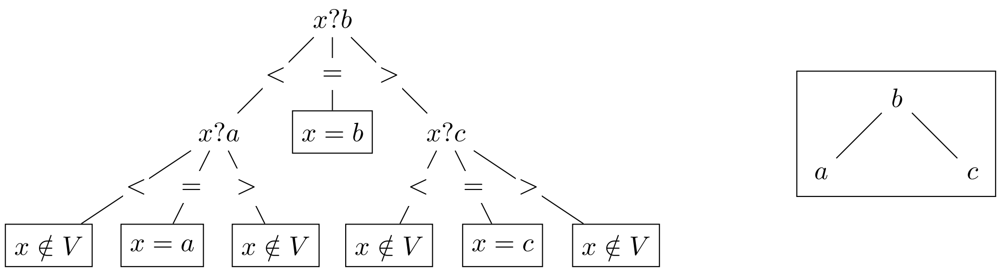


Figure 1.1.: A search strategy to find a value x in a totally ordered set $V = \{a, b, c\}$, with $a < b < c$. As simplified informal text (top), as a decision tree (bottom left), and as a binary search tree (bottom right).

a search tree on C and attach it as a child of r. See figure 1.2 (top) for an example. If G is a path, then we essentially recover BSTs.

We can use T as a search strategy as follows. The first query is always at r. Depending on the answer, we narrow down our search to some component of $G - r$, and descend into the respective subtree. As for BSTs, the number of queries required to find a node is its depth in the search tree.

To the author's knowledge, this search model has first been explicitly defined by Onak and Parys [OP06]. However, as we will see later in this chapter, related concepts have been studied already in the 19th century.

Searching in connected graphs. In this thesis, we mostly deal with search trees on trees. However, our definition of search trees on G also works when G is not a tree; it only requires G to be connected (see figure 1.2, bottom for an example). Search trees on graphs (STGs) are also known as *elimination trees* and are connected to important concepts such as *tree-depth* and *tree-width* [BGHK95, BCI+20]. In these settings, search trees are seen as hierarchical decompositions of the underlying graph instead of data structures for searching. This interpretation is our main motivation to study search trees in the general setting.

While *search trees* easily generalize as a graph decomposition, it is not quite obvious how *search queries* in graphs should be defined. The crucial difference is that vertex-to-vertex paths may not be unique in general graphs. For example, if we query a vertex v in a cycle, *both* incident edges may "point toward" the target. We briefly describe two approaches to interpret vertex search in graphs that are compatible with our definition of search trees:

- A query to $v \neq x$ yields (some description of) the component C of $G' - v$ that contains x, where G' is the subgraph induced by the currently remaining candidates. This has the disadvantage of making a query *non-local*, as the result depends on previous queries.
- A query to $v \neq x$ yields *all* edges incident to v that are on a path between v and the target x. This is cleaner, as it does not depend on previous queries, just v, x, and the input graph G. Still, the result can be a large list of edges, so we cannot reasonably expect the query to require only constant time.

Despite these caveats, many BST questions naturally generalize to our graph search setting. We will see in this thesis that some results originally motivated by searching have surprisingly wide-ranging applications in seemingly disjoint areas. We apply such results to obtain *dynamic graph* data structures (chapter 10) and combinatorial results on *graph associahedra* (chapter 12).

Another motivation for studying searching in general graphs is that it generalizes not just searching in trees, but also *edge-query* searching and the *list-update* model (both will be defined later).

1.2. Search tree problems

In this section, we discuss various BST problems and lift them into the STT (search trees on trees) and STG (search trees on graphs) settings.

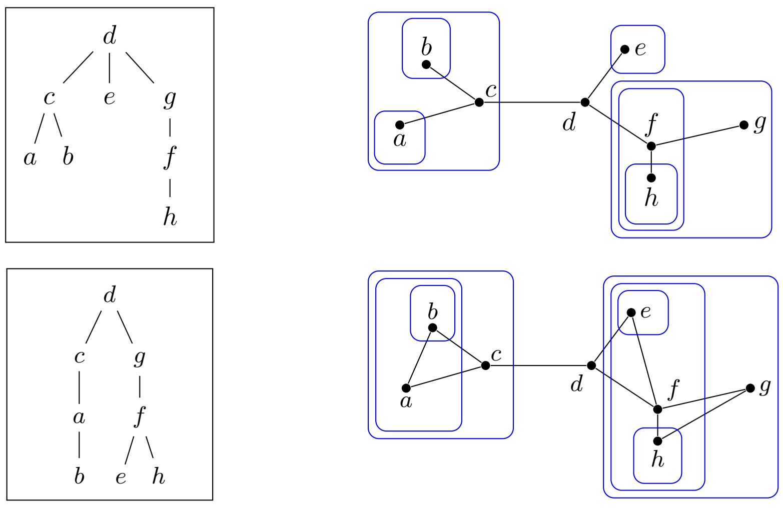


Figure 1.2.: A search tree (top left) on a tree (top right) and a search tree (bottom left) on a non-tree graph (bottom right). The blue boxes indicate remaining components after removing a vertex in the construction of the search tree.

1.2.1. Minimizing worst-case search time

The most obvious question for any data structure is to optimize the worst-case running time of its operations. We take the running time of a search to be the number of performed queries for simplicity. For a search tree, the worst-case search time is its *height* (i.e., the maximum depth of a node). Thus, we want to find a search tree of minimum height on a given connected graph G.

We start with the case when G is a path (so search trees on G are BSTs). We can use the search tree analogue of binary search, i.e., construct a search tree by always choosing the middle vertex of the remaining piece of the path. This is the *complete* BST, with height roughly $\log n$.[5] [Knu68]

Moving on to the case where G is a tree, it turns out that the concept of *middle* vertices generalizes. A *centroid* of a tree is a vertex that partitions the tree into components of size at most $\frac{1}{2}n$. Centroids have been known to exist for arbitrary trees since the 19th century [Jor69]. Building a search tree by repeatedly choosing the centroid yields a so-called *centroid tree*, again with height roughly $\log n$.

It is worth mentioning that, while logarithmic height is optimal for paths, the same is not necessarily true for trees in general. Consider, for example, a star:[6] By taking its center as the root, we always have a search tree of height at most two, regardless of how many leaves the star has. Centroid trees are not necessarily optimal (see section 4.5), even though they may have have height much less than $\log n$ (in the star case, for example). We can thus see a certain jump in complexity when going from paths to general trees. Still, computing the minimum-height search tree on a given tree is possible in linear time [Sch89b, OP06].

The general case, where G is an arbitrary graph, has been extensively studied before under the names *minimum elimination tree height*, *vertex/node ranking*, *ordered colorings*,

[5]Throughout this section, we write $n = |V(G)|$.
[6]A tree with a single non-leaf node (the *center*) that is adjacent to all leaves.

and *tree-depth.*[7] In this thesis, we adopt the latter term, i.e., we say the *tree-depth* of G is the minimum height of a search tree on G.

Computing the tree-depth of a general graph is NP-hard [Pot88], but fixed-parameter tractable [BDJ+98, RRVS14]. In this thesis, we do not present any new results for computing the tree-depth, but the concept will be useful.

1.2.2. Minimizing expected search time

Above, we discussed optimizing search trees for *worst-case* search time. While this is the most common benchmark for data structures in theoretical studies, in practice, data often has some inherent structure. (See, e.g., Roughgarden [Rou21].)

Assume the input (i.e., the element or vertex to search for) is randomly drawn from a known probability distribution p. We now want to minimize the *expected* search time. This is clearly the same as the *expected depth* of the target node. An important special case is when p is the uniform distribution; then, the problem is equivalent to minimizing the sum of node depths.

In the BST case, this problem is mostly solved. For the uniform distribution, the complete BST again is clearly optimal. For arbitrary distributions, a dynamic programming algorithm due to Knuth [Knu71] computes the optimal BST in quadratic time. It essentially computes an optimal BST on each contiguous subsequence of the input elements. Further, a weighted analogue of the complete BST provides a $(1 + o(1))$-approximation and can be computed in linear time [Meh75, Fre75, Meh77].

Contributions of this thesis. Part I is dedicated to the problem of computing optimal STGs for a given input distribution. We mainly focus on the case where the underlying graph is a tree. In this setting, we present multiple approximation algorithms, among them a fast 2-approximation using *weighted centroid trees* (chapter 4), a PTAS[8] using dynamic programming (chapter 5), as well as an FPTAS[9] and a pseudo-polynomial exact algorithm based on work on a related problem (chapter 6). The question whether the problem is polynomially tractable or NP-hard is left open.

We also study more general cases, where the underlying graph is not necessarily a tree. Our PTAS nicely generalizes to graphs with bounded tree-width (section 5.4.2). Finally, we show that the problem is NP-hard for graphs of tree-width at least 15 (section 7.1). It was observed before that for dense graphs, the problem is hard even when p is the uniform distribution [HBB+21] (see section 7.2).

1.2.3. Adaptive searching with rotations

A simple and well-known local operation to change a BST is the *rotation*, which essentially swaps a child node with its parent, along with transferring a child to maintain a valid BST.

[7]We refer to the textbook by Nešetřil and Ossona de Mendez [NOdM12] and the thesis of Sánchez Villaamil [SV17] for a more comprehensive treatment of the long history of tree-depth and equivalent concepts.

[8]Polynomial-time approximation scheme; i.e., an algorithm with parameter $\varepsilon > 0$ that computes a $(1 + \varepsilon)$-approximation in time $n^{f(\varepsilon)}$, for some f.

[9]Fully-polynomial-time approximation scheme; i.e., an algorithm with parameter $\varepsilon > 0$ that computes a $(1 + \varepsilon)$-approximation in time $\mathrm{poly}(n) \cdot \mathrm{poly}(1/\varepsilon)$.

The reader may be familiar with rotations from BST-based data structures such as *AVL trees* and *red-black trees* [CLRS01]. These data structures allow adding or removing new elements, and use rotations to restructure themselves into an (approximately) balanced state afterwards. Rotations nicely generalize to STGs. Figure 1.3 shows a rotation in a BST and a rotation in an STG; for precise definitions, consult section 2.2.2.

For now, we will not bother with additions and deletions of elements (it is not quite obvious what exactly that means for STTs and STGs). Instead, we focus on a different use of rotations: Making a search tree *adapt* to certain properties of the input, such as the distribution of its elements.

Our *dynamic search tree model* is roughly the following. We start with a search tree of our choice (on a given graph), and receive a sequence of elements to search for. Before every search, we are allowed to execute arbitrary rotations; then, we search for the element as usual. An algorithm determining the initial search tree and the rotations in each search is called a *dynamic search tree algorithm*. The time the algorithm takes for each element is measured as the number of rotations it executes plus the number of queries in the search (i.e., the current depth of the searched node).[10]

In the *online* model, the algorithm receives the input element-by-element, whereas in the *offline* model, the whole sequence of inputs is known from the start. In contrast, we call an unchanging search tree (as discussed above) a *static* search tree.

An intuitive and useful online heuristic is MoveToRoot, which simply rotates the found element to the root after every search. Elements that are searched often are usually found more quickly. In fact, for BSTs, it can be shown that this heuristic is as good as any static search tree, as long as the input is drawn independently from some fixed distribution (up to a constant factor, and as long as the number of searches is large enough) [AM78].

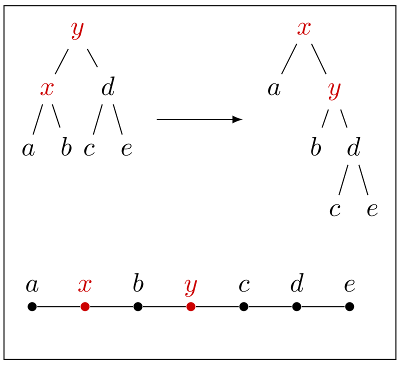
Note that MoveToRoot is online, it does not know the distribution, and the performed rotations do not take into account the *global* structure of the tree at all. However, the independence assumption on the input is necessary, since there exist input sequences where MoveToRoot requires linear time per search. MoveToRoot easily generalizes to STTs, but then may perform badly even for independently distributed searches (see section 8.2).

Perhaps the best-known adaptive BST algorithm is Splay [ST85b]. It matches the optimal static BST on any (long enough) input sequence (up to a constant factor), without the restriction that the searches need to be independent of each other. This property is called *static optimality* and also implies a worst-case running time of $\mathcal{O}(m \log n)$ for m searches among n elements, since Splay in particular matches the balanced BST. Splay has many more interesting adaptive properties (see section 8.1 for a brief overview, and Chalermsook, Goswami, Kozma, Mehlhorn, and Saranurak [CGK+16] for more details).

Perhaps the most important question in dynamic BST research is the *dynamic optimality conjecture*. It states that there exists some dynamic BST algorithm that is constant-competitive with the *offline optimum*, i.e., its running time is at most a constant factor above the minimum time needed by an *offline* algorithm to serve an input sequence. [ST85b]

Besides Splay, the Greedy algorithm [DHI+09] is another prime candidate for dynamic optimality. However, for both algorithms, no better competitive ratios than the trivial $\mathcal{O}(\log n)$ are known. The best known competitive ratio of $\mathcal{O}(\log \log n)$ is achieved by *Tango trees* [DHIP07].

[10]This is a simplification; we make the model precise in chapter 8.



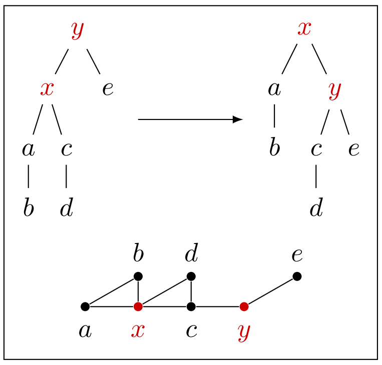


Figure 1.3.: Examples of a BST rotation (left) and an STG rotation (right). In both cases, the node x is rotated with its parent y. The respective underlying graph is sketched at the bottom.

In the general STG setting, work on the dynamic search tree problem has been scarce so far. The model was first defined by Bose, Cardinal, Iacono, Koumoutsos, and Langerman [BCI$^+$20], who generalized Tango trees to the STT setting (i.e., the underlying graph is a tree). The very specific case where the underlying graph is a clique has been studied as the *list-update* problem [ST85a] (see section 8.3).

Contributions of this thesis. In part II , we investigate the dynamic search tree model, in particular the STT case. Our main result is a generalization of Splay to STTs (chapter 8), for which we prove static optimality, along with further adaptivity properties. We use and expand the framework of *Steiner-closed search trees* (or *2-cut search trees*) of Bose, Cardinal, Iacono, Koumoutsos, and Langerman [BCI$^+$20].

We also extensively discuss an application of dynamic search tree algorithms to *dynamic graph problems* (chapter 10). Essentially, a dynamic graph data structure maintains a graph under edge insertion/deletion, and additionally allows some query, such as computing the distance between two given vertices. We build a *dynamic forest* data structure where each tree in the underlying forest is represented as a search tree on that tree. While our algorithms only work if the underlying graph is a forest, we discuss some ideas to expand them to bounded-tree-width graphs (section 10.10).

Remark. Dynamic forests can also be seen as the "truly" dynamic STT model. So far, we have ignored possible modifications of the underlying graph. However, BST-based data structures usually allow adding and removing elements, and sometimes even the strictly stronger operations of merging and splitting whole BSTs. Observe that adding and removing edges from a dynamic forest, where each underlying tree is represented by an STT, amounts to merging and splitting STTs in the same way.

1.2.4. Minimizing worst-case search time with non-uniform cost

A different static search tree model assigns each vertex a *cost* required to query it; the time/cost for a search is taken as the total cost of all queries. We refer to this model as the *weighted-cost model*. Optimizing worst-case search time (in a static tree) leads to a very different problem, where reducing the height by placing “central” vertices at the root is still advantageous, but, at the same time, high-cost vertices should be *far* from the root. This problem is also called *weighted vertex ranking* and known to be strongly NP-hard [DN06] but not APX-hard under standard complexity theory assumptions [DKUZ17]. We do not investigate the weighted-cost model in this thesis.

1.2.5. The diameter of graph associahedra.

We now turn to a combinatorial problem related to STGs. The *rotation distance* between two search trees (on a common graph) is the minimum number of rotations needed to transform one into the other. We are interested in the *maximum* rotation distance between any two search trees on a given graph.

This question is related to *graph associahedra*, a family of polytopes whose graphs (1-skeleta) are exactly the *rotation graphs* of STGs. The rotation graph $\mathcal{R}(G)$ of a graph G is defined as follows: The vertices of $\mathcal{R}(G)$ are the search trees on G, and two search trees are adjacent in $\mathcal{R}(G)$ if they differ by a single rotation. Clearly, the maximum rotation distance is the diameter of the rotation graph, which is the diameter of the polytope by definition. Graph associahedra have been studied before in several different settings; we refer to chapter 11 for more information.

In the special case where the underlying graph is a path, we obtain the classical *associahedra* or *Stasheff polytopes*. The 1-skeleton of an associahedron is, of course, the rotation graph of BSTs, but it also represents the so-called *flip graphs* of parenthesizations[11] and convex triangulations [Tam54, Pou14b]. Both combinatorial and geometric aspects of associahedra have been extensively studied (see, e.g., Ceballos, Santos, and Ziegler [CSZ15]). The diameter is known to be linear in the number of nodes of the BSTs.

For some restricted graph classes, such as cycles and *trivially perfect graphs*, the approximate diameters of the respective graph associahedra are known (see chapter 11).

Contributions of this thesis. In part III, we study the diameter of graph associahedra. We again focus on the case where the underlying graph is a tree. It is known that *tree associahedra* have diameter between $\Omega(n)$ and $\mathcal{O}(n \log n)$, where n is the number of vertices in the underlying tree. We attempt a more precise classification of trees that lie between these two extremes.

Our main result is an algorithm that approximates the diameter of the rotation graph of a given *tree with bounded path-width.* The class of trees with bounded path-width includes graph classes such as caterpillars, spiders (star-like trees), and lobsters; we give a simple method to determine the diameter of the rotation graph of any such graph, up to a constant factor. The problem is left open for trees of unbounded path-width, in particular subdivisions of binary trees (see chapter 12 for definitions and more details).

The algorithm itself is neither complicated nor particularly interesting, instead, the main result is the bound on its approximation ratio. The proof is quite involved, and uses techniques related to both static optima and dynamic search trees. We also give a similar approximation for all trees with no vertices of degree two.

[11]The *flips* are applications of the associativity rule.

For non-tree graphs, we prove a generic upper bound related to optimal static search trees, but our lower bound techniques largely fail in this general setting.

1.2.6. Computing rotation distance

A related problem is computing the rotation distance between two given search trees on a common graph G. It is known that this problem is NP-hard in general [IKK$^+$23], but polynomially tractable if G is a *complete split graph* [CPVP24]. The special case where G is a path, i.e., computing the rotation distance between two BSTs, is still wide open, as neither NP-hardness nor subexponential algorithms are known.

1.3. Related search models

A different model for vertex search uses *edge queries*. Let the underlying graph G be a tree. Querying an edge e reveals on which side of e the target vertex lies, i.e., which component of $G - e$ contains it. See figure 1.4 (b). In analogy to our search trees, we can define *edge-query trees*, and the definition can also be extended to general underlying graphs. We give a formal definition later in section 7.1.

The three static search tree problems discussed above (worst-case, expected-case and weighted-query) are all easily adapted to edge-query trees, and, in fact, are quite well studied. The worst-case problem is also known as *edge ranking* [Der08], in analogy to vertex ranking. Like the tree-depth/vertex ranking problem, it is known to be NP-hard in general graphs [LY98]. The special case of trees has inspired a series of works [IRV88, dlTGS95, BFN99] culminating in linear-time algorithms [LY01, MOW08].

The expected-case edge-query tree problem is known to be NP-hard even when the underlying graph is a tree [JCLM10, CJLM11], but constant-factor approximation algorithms are known [LM11, CJLM10, CJLM14]. The special case where the input distribution is uniform has been studied in the context of hierarchical clustering [Das16, HBB$^+$21].

The weighted-cost edge-query tree problem is also NP-hard for trees [Der06, CJLV12, CKL$^+$16].

In another more general model, the underlying search space is a partially ordered set, and querying a vertex reveals whether the target element is smaller or equal to the query element. Observe that if the poset is *tree-like*, i.e., its Hasse diagram is a tree, then we

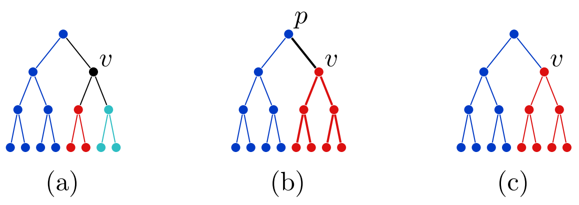


Figure 1.4.: Different types of queries in a tree. (a) Vertex-query at v: Indicates whether v is the searched vertex, or otherwise on which side of v the searched vertex is. (b) Edge-query at $\{v, p\}$: Indicates on which side of the edge the searched vertex is. (c) Tree-like-poset query at v: Indicates whether the searched vertex is smaller or equal to v (red), or not (blue).

again have the edge-query model: A poset-query at element v corresponds to a query to the edge between v and its parent in the Hasse diagram. See figure 1.4 (c). The poset-query model in general is also well-studied [LS85, CDKL04, Der08, CJLM10, CDL24].

Finally, in a certain variant of the vertex-query model, queries indicate the first edge of a *shortest path* to the target. In a tree, this model equivalent to ours (since paths are unique), but in general graphs, the shortest-path model is significantly stronger: $\mathcal{O}(\log n)$ queries suffice in the worst case for any graph. [EKS16]

To the author's knowledge, dynamic variants of any of these models have not been studied.

Edge queries vs. vertex queries. It is easy to see that an edge query in a graph G is essentially the same as a vertex query in the line graph $L(G)$ of G.[12] However, searching for a vertex is subtly different. In the edge-query model, a vertex is identified by querying all incident edges. This corresponds to finding all vertices of a certain *clique* in $L(G)$; a normal vertex search with vertex queries would only find a single vertex.

For the worst-case problem (i.e., edge ranking), this does not matter; we simply want to minimize the height in both cases. Thus, the linear-time edge ranking algorithm mentioned above [LY01, MOW08] translates to a linear-time vertex ranking/tree-depth algorithm for *line graphs of trees*.

Results for the expected-case problem, on the other hand, do not always transfer directly. Still, as we show in section 7.1, the edge-query version of the problem does reduce to the vertex-query version on the line graph if we only ever search for *leaves* of the original graph. This allows us to transfer the NP-hardness result of Cicalese, Jacobs, Laber, and Molinaro [CJLM11] to the vertex query setting.

Moreover, observe that an edge query can be simulated by a vertex query (at one of the endpoints of the edge). Conversely, a vertex query at v can be simulated by querying all incident edges. Thus, if the underlying tree has bounded maximum degree, the two models are equivalent up to a constant factor.

Finally, in the weighted-cost model, the edge-query version easily reduces to the vertex-query version by simply *subdividing* each edge with a single vertex. [Der06]

1.4. Applications

We end this chapter by briefly discussing a few examples of direct applications for our search model. We omit the previously discussed dynamic graphs (see chapter 10) and the many applications of tree-depth (i.e., worst-case-optimal search trees; see, e.g., the thesis of Sánchez Villaamil [SV17]).

File system comparison. Suppose we have an older and a newer copy of a file system on different computers in a network, and we want to find changed files with a minimum amount of network traffic. The file system is a rooted tree, with directories as inner nodes and files as leaves.[13]

[12]The vertex set of $L(G)$ is the set of edges of G, and two vertices of $L(G)$ are adjacent if the corresponding edges in G share an endpoint.

[13]We ignore special cases such as symbolic links.

For simplicity, assume only one file has changed. A simple approach is to compute a checksum for each file and each directory. To determine whether the changed file is in a given directory, we only have to compare the two checksums.[14] Observe that this is exactly a query in the tree-like poset search model, which is equivalent to edge queries [BFN99, MOW08]. Since we only search for leaves (files), this is equivalent to searching with vertex queries in the line graph of the underlying tree (see section 1.3).

Minimizing the worst-case network traffic with this method thus means finding a minimum-height edge-query tree on the file system tree, or, equivalently, a minimum-height (vertex-query) search tree on the *line graph* of the file system tree.

If we are given probabilities for the files to change (e.g., based on past experience), then minimizing the expected network traffic is precisely the optimal static search tree problem.

Quadtrees. A quadtree is a data structure for point location, which is essentially a hierarchical partition of a square into smaller squares. It is a rooted tree where each node v represents a square R_v. If p is the parent of v, then R_v must be fully contained in R_p. Each node v has at most four children, which usually represent the *quadrants* of R_v.[15]

A standard task for quadtrees is to find the leaf whose square contains a given point. Observe that we can use the quadtree itself like a search tree: First, determine which of the four children of the root contains the point, then recurse in the determined node. However, this takes time proportional to the height of the quadtree, which may be linear in the number of nodes.

Checking whether the input point is contained in R_v for a node v amounts to checking whether the target leaf is a descendant of v. Hence, we again have the tree-like poset-query, i.e., the edge-query model. Since we are only searching for leaves, this is again equivalent to the vertex-query model on line graphs. Additionally, our underlying tree has bounded degree, so we can simulate vertex queries with a constant number of edge queries (see section 1.3). Thus, this is an application of search trees on *trees*. In particular, we can use the aforementioned *centroid tree* to obtain logarithmic search time. This is a well-known technique, and a centroid tree on a quadtree is also called a *finger tree* [HP11].

Water surveillance. Let us model the *sewer system* of a city as a directed graph, with the edge directions indicating the flow of water. Usually, this is a tree where all edges point towards the root; the leaves are households and the root is a sewage treatment plant. If sufficient amounts of some interesting or unwanted substance is introduced by a household, then all water on the path between the household and the root is contaminated. We want to find the offending household. [CDV24]

Suppose we have a reliable test that detects whether the substance is contained in the water or not. A single test at some point in the system is an edge query: The affected household is *upstream* from the test point if and only if the test is positive. On the other hand, testing all incoming and outgoing pipes of a junction is a vertex query. We can thus apply our algorithms in an attempt to minimize the number of necessary tests.

[14] Assume there are no collisions.

[15] We refer to the textbook of Har-Peled [HP11] for a proper definition.

2. Preliminaries

In this chapter, we review some basic definitions and notation, and define search trees and related concepts.

We start with notation related to graphs and other basic structures, and then proceed to *rooted trees* in section 2.1. Most of this should be familiar to the reader. We then define search trees on graphs in section 2.2. Sections 2.2.3 to 2.2.6 are mainly needed in part III (though they are also used heavily in sections 2.3 and 2.4), so the reader may skip them first and refer to them later as needed.

Finally, sections 2.3 and 2.4 relate search trees to the concepts *trivially perfect graphs*, *tree-depth*, *chordal graphs* and *tree-width*. While most of these connections have been implicit in previous work, we present them in an explicit and systematic manner. The definitions and results in these sections only appear sparsely in the rest of the thesis and can be referred to as needed.

All definitions in this and later chapters are found in the index and symbol reference (pages 213 to 215).

Basic notation. We denote by $\mathbb{N}_+$ the set of positive integers, and by $\mathbb{N}_0$ or $\mathbb{N}$ the set of nonnegative integers. For $k \in \mathbb{N}_+$, we write $[k] = \{1, 2, \dots, k\}$.

If $f\colon A \to B$ is a function, then the *restriction* of f to a subset $X \subseteq A$, written $f|_X$, is the function $g\colon X \to B$ such that $g(x) = f(x)$ for each $x \in X$.

$f|_X$

Graphs. All graphs in this thesis are simple and undirected, unless specified otherwise. We denote the set of vertices of a graph G by $V(G)$ and the set of edges by $E(G)$. For a vertex set $U \subseteq V(G)$, we write $G[U]$ for the subgraph induced by A. We further write $\mathbb{C}(G)$ for the set of connected components of G.

$V(G)$, $E(G)$

$G[U]$, $\mathbb{C}(G)$

For convenience, if $v \in V(G)$, we write $G - v = G[V(G) \setminus \{v\}]$. Similarly, for $A \subseteq V(G)$, we write $G - A = G[V(G) \setminus A]$, and for $e \in E(G)$, we let $G - e$ be the graph obtained by removing e from G.

$G - v$, $G - A$, $G - e$

The degree of a vertex $v \in V(G)$ is denoted by $\deg_G(v)$. If G is a tree and $U \subseteq V(G)$, the *convex hull* $\mathrm{ch}_G(U)$ is the set of vertices that lie on the path between two vertices in U.

$\deg_G(v)$

convex hull $\mathrm{ch}_G(U)$

Contractions and subdivisions. If G is a graph and u, v are adjacent vertices in G, then the *contraction of u and v into s* produces a graph H with $V(H) = V(G) \setminus \{u, v\} \cup \{s\}$ and $E(G)$ consists of $E(G) \setminus \{\{u, v\}\}$ and an edge between s and x for each x that is adjacent to u or v in G.

contraction

On the other hand, a *subdivision* of an edge $\{u, v\}$ in G adds a new vertex "on" that edge. More formally, it produces a graph H with $V(H) = V(G) \cup \{x\}$ and $E(H) = E(G) \setminus \{\{u, v\}\} \cup \{\{u, x\}, \{x, v\}\}$, where x is a new vertex not contained in $V(G)$. Observe that a subdivision can be reverted by a contraction, but the opposite is not always true.

subdivision

separate

Separators. Let G be a connected graph. We say a set $A \subseteq V(G)$ *separates* two vertices $u, v \in V(G)$ in G if u and v are in different components of $G - A$. We say a single vertex a *separates* u and v in G if $\{a\}$ separates u and v. The following fact will be useful later.

Observation 2.1. *Let G be a graph and H be a connected subgraph of G. Suppose $A \subseteq V(G)$ separates two vertices $u, v \in V(H)$ from each other in G. Then $A \cap V(H)$ is nonempty and separates u from v in H.*

Special graphs. The reader will be familiar with *complete graphs* and *complete bipartite graphs.* We also call the former *cliques* and the latter *bicliques.*

complete split graph

Figure 2.1 shows examples of graph classes defined in the following. For all $m, n \in \mathbb{N}_0$, the *complete split graph* $\mathrm{SPK}_{m,n}$ is the graph G whose vertex set can be partitioned into two sets A and B of size m and n, respectively, such that $E(G) = \{\{a, a'\} \mid a, a' \in A\} \cup \{\{a, b\} \mid a \in A, b \in B\}$. In other words, the set A forms a clique, and A, B form the two parts of a biclique.

star

spider

A *star* is a tree where one vertex, the *center*, is adjacent to all other vertices. A *spider* or *star-like tree* is a tree that consists of a designated *center* vertex, and a collection of paths (the *legs*), such that for each leg, the center is connected to one of the endpoints of the leg. Observe that a star is a spider where each leg has one vertex.

caterpillar

A *caterpillar* is a tree with a designated path (the *spine*), whose removal splits the tree into isolated vertices (the *legs*).

lobster

A *lobster* is a graph with a designated path, such that all vertices have distance at most two to the path. Observe that every caterpillar is also a lobster.

$\omega(G)$

$\chi(G)$

Graph invariants. We write $\omega(G)$ for the *clique number* of a graph G (the number of vertices in a maximum clique of G), and $\chi(G)$ for the *chromatic number* of a graph G. We refer to standard textbooks [Har69] for more details.

$V(\sigma)$

$\sigma|_X$

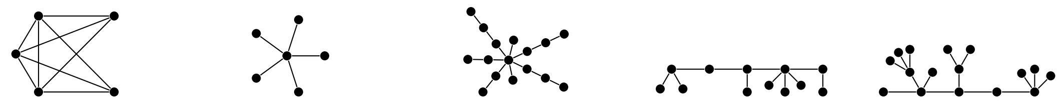
Sequences and permutations. In this thesis, sequences are always finite. For a sequence σ, let $V(\sigma)$ denote the set of elements occurring in σ. Given a set X, we define the *restriction* $\sigma|_X$ as the unique subsequence of σ obtained by removing all elements in $V(\sigma) \setminus X$.

permutation

$<_\pi$

partial permutation

A *permutation* of a finite set A is a sequence π with $V(\pi) = A$ that contains each element of A precisely once. For $a, b \in A$, we write $a <_\pi b$ if a precedes b in π, and we write $a \leq_\pi b$ if $a <_\pi b$ or $a = b$. We say *a directly precedes b* if $a <_\pi b$ and there are no elements between a and b in π. For two sets $B, C \subseteq A$, we say *B precedes C* if $b <_\pi c$ for all $b \in B, c \in C$. A *partial permutation* of a set A is a permutation of some subset $B \subseteq A$.



(a) Complete split graph $\mathrm{SPK}_{3,2}$ (b) Star (c) Spider (d) Caterpillar (e) Lobster

Figure 2.1.: Examples of special graphs classes.

2.1. Rooted trees

We follow Cormen, Leiserson, Rivest, and Stein's terminology [CLRS01, Appendix B.5.2], with some additions that are useful for us.

rooted tree | root(T) | (un)rooting | rooted forest

A *rooted tree* T is a tree where one vertex is designated as the *root*, denoted by $\mathrm{root}(T)$. We also refer to T as a *rooting* of the underlying (unrooted) tree G, and, conversely, call G the *unrooting* of T. To avoid confusion between normal trees and rooted trees, we call the vertices of a rooted tree *nodes*. A *rooted forest* is a collection of disjoint rooted trees.

ancestor, descendant | parent, child | grandparent

Ancestors and descendants. A node u is an *ancestor* of another node v if u lies on the path from v to $\mathrm{root}(T)$. In that case, we also call v a *descendant* of u. If u is an ancestor (descendant) of v and $u \neq v$, then we call u a *proper* ancestor (descendant) of v. If v is a descendant of u and there is an edge between u and v, then we call v a *child* of u and u a *parent* of v, and write $\mathrm{parent}_T(v) = u$. The *grandparent* of a node v is the parent of the parent of v.

$\preceq_T$

We write $u \preceq_T v$ if u is an ancestor of v in the rooted tree T. Here and in the other definitions in this section, we omit the subscript T if clear from context. We write $u \prec v$ if u is a proper ancestor of v.

lowest common ancestor | $\mathrm{LCA}_T(U)$

A *common ancestor* of a set of nodes U is a node v with $v \preceq u$ for all $u \in U$. A *lowest common ancestor (LCA)* of U is a node v that is a common ancestor and that satisfies $a \preceq v$ for every common ancestor a of U. It is easy to see that an LCA always exists and is unique. We denote it by $\mathrm{LCA}_T(U)$.

subtree | subforest

Subtrees. Let T be a rooting of a tree G. A *subtree* P of T is a connected subgraph of G, rooted at $\mathrm{LCA}_T(V(P))$ (observe that indeed $\mathrm{LCA}_T(V(H)) \in V(H)$). A *subforest* of T is a collection of disjoint subtrees.

$T - A, T - F$

We say a subtree is *induced* by its vertex set. Observe that each subset of vertices induces at most one subtree. The subforest of T *induced* by a node set U, denoted by $T[U]$, consists of the subtrees induced by the components of $G[U]$. We also write $T - A = T[V(T) \setminus A]$ for a set $A \subseteq V(T)$, and $T - F = T[V(T) \setminus V(F)]$ if F is a subforest of T. See figure 2.2 for examples.

prefix | suffix

A subtree P of T is called a *prefix* if $V(P)$ is closed under taking ancestors in T. A rooted forest F of subtrees of T is called a *suffix* if $V(F)$ is closed under taking descendants in T. Observe that $T - P$ is a suffix if P is a prefix, and $T - F$ is a prefix if F is a suffix. Also observe that a subtree P is a prefix of T if and only if $\mathrm{root}(P) = \mathrm{root}(T)$.

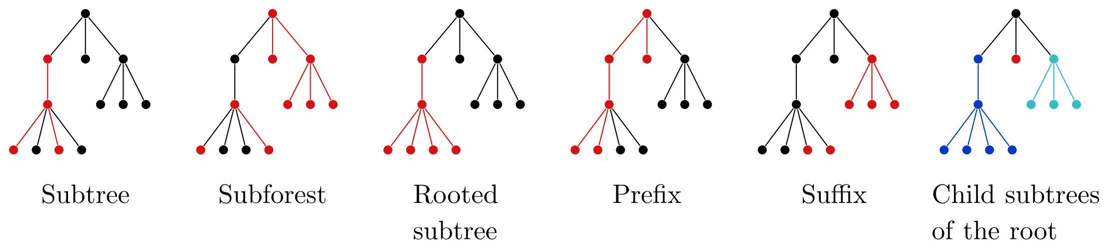


Figure 2.2.: Examples of the various subtree/subforest types.

The subtree of T induced by a node v and all its descendants in T is called the *subtree rooted at* v or just a *rooted subtree*, and is denoted by T_v. Observe that a subtree is a rooted subtree if and only if it is a suffix. A *child subtree* of a node v in T is the subtree of T rooted at some child of v.[1]

rooted subtree

child subtree

We extend the concept of rooted subtrees to rooted *forests*. If T is the rooted tree in a rooted forest F that contains v, then we write $F_v = T_v$. (Obviously, most of the other concepts can be extended similarly, but that is not necessary for our purposes.)

leaf

inner node

degenerate

binary tree

perfect

Leaves and special trees. A node v in T is a *leaf* if it has no children. Otherwise, it is called an *inner node.*

If T has only one leaf, it is called *degenerate*. Equivalently, T is degenerate if all nodes have at most one child. If each node has at most two children, then T is called *binary.* A binary tree is called *perfect* if all leaves have the same depth, and each inner node has exactly two children.

root path

$\mathrm{depth}_T(v)$

$\mathrm{height}(T)$

Depth and height. The path from $\mathrm{root}(T)$ to some node v is called the *root path* of v. Let $\mathrm{Path}_T(v)$ denote the set of nodes on the root path. Observe that $\mathrm{Path}_T(v)$ is exactly the set of non-strict ancestors of v. The number $|\,\mathrm{Path}_T(v)|$ of ancestors is called the *depth* of v in T and is denoted by $\mathrm{depth}_T(v)$. Note that the depth of the root is always one. The *height* of T, denoted $\mathrm{height}(T)$ is the maximum depth of a node in T.

2.2. Search trees on graphs

search tree

A *search tree* on a connected graph G is a rooted tree T with root $r \in V(G)$. Let $\{C_1, C_2, \dots, C_k\} = \mathbb{C}(G - r)$. Then r has precisely k children $c_1, c_2, \dots, c_k$ such that T_{c_i} is a search tree on C_i for $i \in [k]$.

Since we only define search trees on *connected* graphs, henceforth, whenever we write "search tree on a graph G", it will be implied that G is connected.

Observation 2.2. *Let T be a search tree on a graph G. Then $V(T) = V(G)$.*

If S is subtree of T, we write $G[S] = G[V(S)]$ for brevity. Observe that $G[T_v]$ is connected for every $v \in V(T)$. In fact, we can show the following equivalence.

Lemma 2.3 (Characterization of STGs)**.** *Let G be a connected graph. A rooted tree T with $V(T) = V(G)$ is a search tree on G if and only if*

(i) for each edge $\{u, v\} \in E(G)$, we have $u \prec_T v$ or $v \prec_T u$; and

(ii) for each node $v \in V(T)$, the subgraph $G[T_v]$ is connected.

Proof. First suppose that T is a search tree. For property (ii), recall that, by definition, each rooted subtree T_v of T is a search tree on some component C of some subgraph of G, and $V(T_v) = V(C)$ by observation 2.2.

To show property (i), take some edge $\{u, v\} \in E(G)$. If $u = \mathrm{root}(T)$, then $u \prec_T v$, so we are done, and similarly for the case $v = \mathrm{root}(T)$. If u and v are in different components

[1]Rooted subtrees and child subtrees are often just called *subtrees* in the literature.

of $G - \mathrm{root}(T)$, then there cannot be an edge between them. Finally, if u and v are in the same component C of $G - \mathrm{root}(T)$, then $\mathrm{root}(T)$ has a child c such that T_c is a search tree on C, so property (i) holds by induction.

We now show the other direction. Suppose properties (i) and (ii) hold. We show that for each child subtree T_c of $r = \mathrm{root}(T)$, the vertex set $V(T_c)$ induces a connected component of $G - r$. This implies that T is a search tree by induction.

Clearly, each $V(T_c)$ induces a connected subgraph of $G - r$ by property (ii). Now suppose $V(T_c)$ does not induce a connected component of $G - r$. Then there exists a vertex $v \notin V(T_c)$ with $v \neq r$ that is adjacent to some $u \in V(T_c)$. But then v must be an ancestor or descendant of u by property (ii), which means that either $v \in V(T_c)$ or $v = r$, a contradiction. ☐

Of course, connected subgraphs of G do not necessarily correspond to subtrees of a fixed search tree T on G. However, the following weaker statement holds and will be useful later.

Lemma 2.4. *Let T be a search tree on a graph G, and let $U \subseteq V(G)$ induce a connected subgraph of G. Then $\mathrm{LCA}_T(U) \in U$.*

Proof. Suppose, for the sake of contradiction, that $a = \mathrm{LCA}_T(U) \notin U$. Note that $G[U]$ is a subgraph of $G[T_a] - a$, since a is an ancestor of all $u \in U$. Further, more than one child subtree of a in T must contain nodes in U (otherwise u would not be the *lowest* common ancestor), so U is not contained in a single connected component of $G[T_a] - a$. Thus $G[U]$ cannot be connected, a contradiction. ☐

Finally, the following occasionally useful fact holds by definition.

Observation 2.5. *Let T be a search tree on a graph G, let $v \in V(T)$, and let x, y be descendants of v in T. Then x and y are in different child subtrees of v if and only if v separates x from y in $G[T_v]$.*

binary search tree

Search trees on paths. A *binary search tree* (BST) is a binary rooted tree where each non-root node is designated as either a *left* child or a *right* child, and each inner node has at most one child of each type.

As mentioned before, binary search trees are essentially equivalent to search trees on *paths*. Let T be a search tree on a path G. Clearly T is binary. Arbitrarily fix one of the endpoints of G as the *leftmost* vertex. Then a child c of a parent p is a *left* child if and only if c is closer to the leftmost vertex than p. This gives us a binary search tree. Rotations within T, as defined in section 2.2.2, are precisely binary search tree rotations.

2.2.1. Boundaries

outer boundary

k-cut subgraph

Let G be a graph and H be an induced subgraph of G. The *outer boundary* of H, denoted $\partial_G(H)$, is the set of vertices in $G - H$ that are adjacent to some vertex in H. We call H a *k-cut subgraph* of G if $|\partial_G(H)| \leq k$.[2]

[2]Observe that H can be split off G with a *vertex cut* of size k.

k-cut search tree

We write $\partial_G(A) = \partial_G(G[A])$ for a set $A \subseteq V(G)$. If T is a search tree on G and $v \in V(T)$, we write $\partial(T_v) = \partial_G(V(T_v))$ for short. The node $v \in V(T)$ is called *k-cut* if $|\partial(T_v)| \leq k$, i.e., if $G[T_v]$ is k-cut. The search tree T itself is called *k-cut* if every node of T is k-cut. Observe that 1-cut search trees only exist when the underlying graph is a tree and are then precisely the rootings of the underlying tree.

The following technical observations will be useful later.

Observation 2.6. *Let G be a connected graph, and let $A \subseteq B \subseteq V(G)$, and let H be a component of $G[B \setminus A]$. Then $\partial_G(H) \subseteq \partial_G(B) \cup A$.*

Observation 2.7. *Let p be a node in a search tree T on a graph G, and let v be a child of p. Then $p \in \partial(T_v)$ and $\partial(T_v) \subseteq \partial(T_p) \cup \{p\}$.*

Corollary 2.8. *For each node v in a search tree T on G, the boundary $\partial(T_v)$ consists entirely of strict ancestors of v in T.*

Remark. The concept of k-cut search trees is a generalization of so-called *Steiner-closed STTs* [BCI+20], which correspond to the case when $k = 2$ and the underlying graph is a tree [BK22].

2.2.2. Rotations

We now generalize BST rotations to STGs [MP15, CLPL18]. Rotations feature most heavily in parts II and III, but are also used in part I to make certain algorithms or proofs more intuitive.

Essentially, a rotation is a *local* operation that swaps a parent and child in a search tree, and otherwise makes a minimal amount of changes in order to preserve the search tree properties.[3] These changes amount to moving some children of one of the rotated nodes to the other. Figure 2.3 sketches a rotation in search tree on a *tree*. (Rotations in general STGs are a little more difficult to sketch, since multiple children may change parents.)

rotation

We now formally define rotations. Let v be a node in a search tree T on a graph G, and let p be the parent of v. A *rotation* of v with its parent (also called a *rotation at v*) is performed as follows. Make p a child of v and make v a child of the previous parent of p, if it exists (otherwise, make v the root). Then, every child c of v with $p \in \partial(T_c)$ is made a child of p.

Lemma 2.9. *Let G be a* tree. *When rotating a node v with its parent p in a search tree T on G, at most one child is transferred from v to p.*

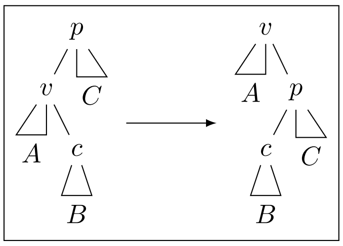
Proof. Suppose two children c, c' of v are transferred to p. Then $p \in \partial(T_c)$ and $p \in \partial(T_{c'})$, which implies that there must be at least two edges from p to $V(T_v)$. But $G[T_v]$ is connected, so there is a cycle, contradicting the assumption that G is a tree. □

$\mathrm{rot}(T, v, p)$

Let $\mathrm{rot}(T, v, p)$ denote the search tree obtained by rotating a node v with its parent p in T. If p is not the parent of v, then $\mathrm{rot}(T, v, p)$ is undefined.

At its core, a rotation of a node v with its parent p swaps p and v while changing the rest of the search tree as little as possible. The following lemma shows this strong *locality*

[3]We will make this idea precise later; see lemma 2.14 in section 2.2.3.



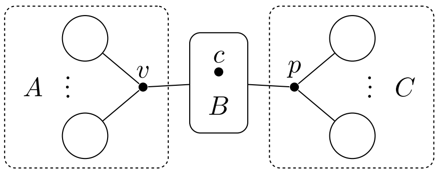


Figure 2.3.: A rotation of a node v with parent p in an STT. The underlying tree is shown on the right. The node c the unique child of v with $p \in \partial(T_c)$. If c does not exist, then $B = \emptyset$ and v, p are neighbors in the underlying graph.

of the rotation operation, and further proves that a rotation indeed results in a valid search tree.

Lemma 2.10. *Let T be a search tree on a graph G, and let $T' = \mathrm{rot}(T, v, p)$. Then T' is a search tree on G, and*

(i) $V(T'_v) = V(T_p)$.

(ii) $V(T'_p) = V(C)$, *where C is the component of $G[T_p] - v$ that contains p.*

(iii) $V(T'_x) = V(T_x)$ *for all $x \in V(G) \setminus \{v, p\}$.*

Proof. Let $U = V(T_p)$. Property (i) holds since we only re-arrange the nodes in U, and v becomes their lowest common ancestor. This also shows property (iii) for nodes $x \in V(T) \setminus U$. For nodes in $V(T_p) \setminus \{v, p\}$, observe that they may change parents, but cannot gain or lose descendants, so property (iii) holds.

For property (ii), let $\mathcal{K} = \mathbb{C}(G[U] - \{p, v\})$. The component C of $G[U] - v$ that contains p clearly consist of p and all $K \in \mathcal{K}$ such that $p \in \partial(K)$. Observe that for each such K, there is some child c of p or v in T such that $K = G[T_c]$, and thus $K = G[T'_c]$ by property (iii). These are precisely the children of p in T', by definition. Hence, we have $V(T'_p) = V(C)$.

To prove that T' is a search tree on G, we use lemma 2.3. Properties (i) to (iii) imply that $G[T_x]$ is connected for all $x \in V(T)$.

Now consider an edge $\{x, y\} \in E(G)$. Without loss of generality, assume $x \prec_T y$. If $x \notin \{v, p\}$, then $x \prec_{T'} y$ by property (iii). If $x = v$, then $V(T_p) \supseteq V(T_v)$ with property (i) implies $x \prec_{T'} y$. If $x = p$ and $y \neq v$, then we have $x \in V(T_p)$ and clearly y is in the same connected component of $G[V(T_p)] - v$ as $x = p$ (there is an edge between them, after all). Thus, we have $x \preceq_{T'} y$ by property (ii). Finally, if $x = p$ and $y = v$, clearly $y \prec_{T'} x$. □

Observe that lemma 2.10 fully characterizes the search tree $T' = \mathrm{rot}(T, v, p)$.

2.2.3. Topological orderings

Constructing a search tree on a graph G can be seen as the following process, illustrated in figure 2.4. Start with G in its entirety, as a single *graph node*. Pick some vertex r as the root, thereby splitting the graph node G into a node r and the child graph nodes $C_1, C_2, \ldots, C_k$, corresponding to the components of $G - r$. Continue to expand graph nodes in this way, until none are left.

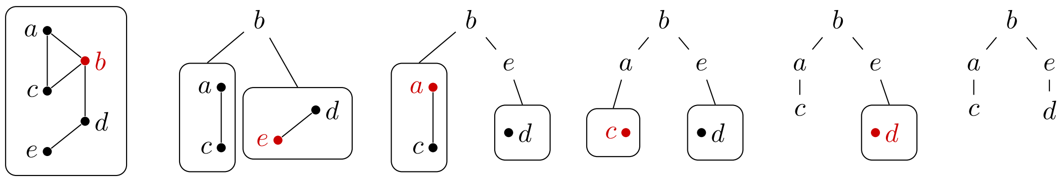


Figure 2.4.: Constructing an STG by repeated graph node expansion. Vertices picked next are highlighted in red.

Clearly, every search tree on G can be constructed this way, and depends on the *order* in which we pick vertices, although two different orders may give the same tree (e.g., in figure 2.4, we could swap expansion of e and a). We now formalize this concept.

topological ordering

Let T be a rooted tree and let π be a permutation of $V(T)$. We call π a *topological ordering* of T if for each $u, v \in V(G)$ with $u \prec_T v$, we have that $u <_\pi v$.

Observe that this definition corresponds to the classical notion of topological orderings (also called *topological sorts* [CLRS01]) of directed acyclic graphs, when we orient all edges of the rooted tree away from the root. A topological ordering of a *search tree* intuitively corresponds to an expansion process, as above.

Topological orderings will be a useful tool in formal proofs throughout the thesis. In this section, we prove a few basic and necessary facts. We start with a formal proof that search trees are uniquely determined by their topological orderings.

Lemma 2.11. *For each permutation π of $V(G)$, there is a unique search tree T^π on G such that π is a topological ordering of T^π.*

Proof. We first prove existence. Construct T^π as follows. Make the first element r of π the root of T^π. Let $C \in \mathbb{C}(G - r)$, and let $\pi' = \pi|_{V(C)}$ be the restriction of π to $V(C)$. Construct the child subtree of r on C recursively as $T^{\pi'}$.

To see that π is a topological ordering of T, consider $u, v \in V(T)$ with $u \prec_T v$. By construction, $T^\pi_u = T^{\pi'}$ with $\pi' = \pi|_{V(T^\pi_u)}$, and thus, u is the first element of π', implying u precedes v in π.

Second, we prove uniqueness by induction on $n = |V(G)|$. If $n = 1$, there is only one possible permutation and one possible search tree. Now let $n \geq 2$, and let π be a topological ordering of two trees T^1 and T^2. We show that $T^1 = T^2$.

Let r be the first element in π. Clearly, we have $\mathrm{root}(T^1) = \mathrm{root}(T^2) = r$. Thus, if $C_1, C_2, \dots, C_k$ are the components of $G - r$, then r has k child subtrees $T^{1,1}, T^{1,2}, \dots, T^{1,k}$ in T^1 and k child subtrees $T^{2,1}, T^{2,2}, \dots, T^{2,k}$ in T^2, such that $T^{1,i}$ and $T^{2,i}$ are search trees on C_i for each $i \in [k]$.

Take some $i \in [k]$ and let $\pi_i = \pi|_{V(C_i)}$. Clearly, π_i is a topological ordering of both $T^{1,i}$ and $T^{2,i}$, so $T^{1,i} = T^{2,i}$ by induction. Applying this argument to every $i \in [k]$ implies $T^1 = T^2$. $\square$

Lemma 2.11 implies that the *number* of search trees on G is bounded by $|V(G)|!$, the number of permutations of $V(G)$. Further observe that a search tree is degenerate if and only if it has exactly one topological ordering.

The following lemma characterizes the search tree determined by a given topological ordering and will be very useful later on.

Lemma 2.12. *Let T be a search tree on a graph G, let π be a topological ordering of T, and let $u, v \in V(T)$. Then $u \prec_T v$ if and only if $u <_\pi v$ and there does not exist a vertex set $A \subseteq V(G)$ that separates u from v and precedes u in π.*

Proof. Clearly, if $u \not<_\pi v$, then $u \not\prec_T v$, so the statement holds. From now on, suppose $u <_\pi v$.

If u is the root, then $u \prec_T v$, and no vertex precedes u in π, hence the statement is true. Now suppose $r := \mathrm{root}(T) \notin \{u, v\}$. It is clear that r is the first element in π.

Suppose u and v lie in different child subtrees of r. Then $u \not\prec_T v$, and r separates them by observation 2.5, so the statement holds.

Now suppose some child subtree T_c of r contains both u and v. If $u \not\prec_T v$, then $u \not\prec_{T_c} v$, so, by induction, there exists a set $A \subseteq V(T_c) \subseteq V(G) \setminus \{r\}$ that separates them in $G[T_c]$ and precedes them in $\pi|_{V(T_c)}$. Clearly, the set $A \cup \{r\}$ separates them in G and precedes them in π.

On the other hand, if there is a set $A \subseteq V(G)$ that separates u from v in G and precedes them in π, then $A \cap V(T_c)$ separates them in $G[T_c]$ (by observation 2.1) and precedes them in $\pi|_{V(T_c)}$. Thus, we have $u \not\prec_{T_c} v$ and therefore $u \not\prec_T v$. □

Corollary 2.13. *Let T be a search tree on a graph G, let π be a topological ordering of T, and let $u, v \in V(T)$. If $u \not\prec_T v$ and $v \not\prec_T u$, then there exists a vertex set $A \subseteq V(G)$ that separates u from v and precedes u and v in π.*

We now show that rotations basically correspond to adjacent swaps in topological orderings. This makes precise the previous remark that rotations make a *minimal* amount of changes besides swapping the two rotated nodes.

Lemma 2.14. *Let T be a search tree on a graph G, and let p be a parent of v in T. Let π be a topological ordering of T where p directly precedes v, and let π' be the permutation obtained by swapping p and v in π. Then π' is a topological ordering of* $\mathrm{rot}(T, v, p)$.

Proof. Let T' be the search tree with topological ordering π'. We need to prove that $T' = \mathrm{rot}(T, v, p)$. We show that T' has the properties in lemma 2.10; recall that those fully characterize $\mathrm{rot}(T, v, p)$.

First, take some $x \in V(G) \setminus \{v, p\}$. Since the *sets* of predecessors and successors of x is the same in both π and π', we have $V(T_x) = V(T'_x)$ by lemma 2.12. This proves property (iii).

We now show property (i), starting with $V(T'_v) \supseteq V(T_p)$. First, since v is a descendant of p in T, we know that p is a descendant of v in T', by lemma 2.12. Since v and p are adjacent in π', we have that p is a child of v in T' in particular. Now take $x \in V(T_p) \setminus \{v, p\}$, and suppose $x \notin V(T'_v)$. Again using lemma 2.12, there must be a set $A \subseteq V(G)$ that precedes v in π' and separates v from x in G. But then A also precedes p in T, so $V(T_p)$ and A are disjoint. Since $x, v \in V(T_p)$, that means $G[T_p]$ cannot be connected, contradicting lemma 2.3. The proof of $V(T'_v) \subseteq V(T_p)$ is similar.

Finally, we show property (ii). Let C be the component of $G[T_p] - v$ that contains p. We need to show that $V(T'_p) = V(C)$. First, observe that $V(T'_p) \subseteq V(T'_v) \setminus \{v\}$, since p is a child of v in T'. Since also trivially $V(C) \subseteq V(T_p) \setminus \{v\} = V(T'_v) \setminus \{v\}$, we can restrict ourselves to the set $V(T'_v) \setminus \{v\}$ in the following discussion.

Accordingly, take some $x \in V(T'_v) \setminus \{v\}$. We consider two cases. First, if x and p are in different child subtrees of v in T', then clearly $x \notin V(T'_p)$. Moreover, observation 2.5 implies that v separates x from p in $G[T'_v] = G[T_p]$, so $x \notin V(C)$ by definition of C.

Second, if x and are in the same child subtree of v in T', then $x \in V(T'_p)$, since p is a child of v in T'. Moreover, observation 2.5 implies that v does not separate x from p in $G[T'_v] = G[T_p]$, so $x \in V(C)$. Thus, we have $V(T'_p) = V(C)$, as desired. □

Finally, we observe that taking a prefix of a topological ordering amounts to taking a prefix of the search tree.

Lemma 2.15. *Let T be a rooted tree and let π be a topological ordering on T. Then, for each prefix π' of π, there is a prefix P of T such that $V(P) = V(\pi')$, and π' is a topological ordering of P.*

Proof. Let π' be a prefix of π. Since π is a topological ordering of T, the set $V(\pi')$ is closed under taking ancestors in T, hence it induces a prefix of T. □

Remark. A concept related to topological orderings of search trees are *elimination orderings*; we briefly discuss them in section 2.4.

2.2.4. Projections

Projections are a way to "reduce" a search tree T on a graph G to a connected subgraph H of G. Note that $V(H)$ may be disconnected in T, so just taking a subtree may not work. However, there is a natural graph-minor-like approach to this, which turns out to be quite useful.

Projections were first defined for connected subgraphs of trees [CLPL18], and later extended to *convex* subgraphs (defined below) of *chordal* graphs [CPVP22]. The original definition involves a process of repeatedly removing leaves (resp. simplicial vertices) from the underlying graph and modifying the search tree accordingly.[4]

The concept of topological orderings allows us to give a simpler equivalent definition that extends to arbitrary connected graphs, and allows taking arbitrary (even non-induced) connected subgraphs, although it is still most useful with convex subgraphs.

projection
$T|_H$

Let T be a search tree on a graph G and let H be a connected subgraph of G. The *projection* of T to H, written $T|_H$, is the search tree S on H such that for each topological ordering π of T, the restriction $\pi|_{V(H)}$ is a topological ordering of S. If $A \subseteq V(G)$ induces a connected subgraph of G, we also write $T|_A = T|_{G[A]}$ for short.

Observe that the projection $T|_H$ is unique by definition. We now show existence.

Proposition 2.16. *For each search tree T on G and connected subgraph H of G, the projection $T|_H$ exists.*

Proof. For each topological ordering π of T, there exists a unique search tree S^π on H with topological ordering $\pi|_{V(H)}$, by lemma 2.11. We need to prove that all these search trees are equal, by showing that each S^π depends only on T, not on π.

[4]A different, but closely related notion was used by Cicalese, Jacobs, Laber and Molinaro [CJLM14] in the edge-query setting.

Take some S^π. Since $V(H)$ is connected in G, lemma 2.4 implies that $r = \mathrm{LCA}_T(V(H))$ is contained in $V(H)$. Thus, the first element of $\pi|_{V(H)}$, and hence the root of S^π, must be r. Now consider the child subtrees S^π_c of r in S^π. Each S^π_c is a search tree on some component C of $H - r$, and admits the topological ordering $\pi|_{V(C)}$. Thus, by induction, we have $S^\pi_c = T|_C$. It follows that S^π does not depend on π. □

Two examples of projections are shown in figures 2.5 and 2.6. For the next lemma, we need the following definition. A subgraph H of a graph G is *convex* if the following holds for each pair $u, v \in V(H)$: If there is a path between u and v in G with all inner vertices outside of H, then $\{u, v\} \in E(H)$.[5]

convex

Observe that if G is a tree, then convex subgraphs are precisely connected subgraphs. In general connected graphs, every convex subgraph is connected and induced, but not every connected induced subgraph is necessarily convex. The following fact is crucial for us.

Observation 2.17. *Let H be a convex subgraph of G, and let $u, v \in V(H)$ and $A \subseteq V(H)$. If A separates u from v in H, then A separates u from v in G.*

We are now ready to prove the following lemma, which underscores the usefulness of the projections, in particular in the convex case.

Lemma 2.18. *Let T be a search tree on G, let H be a connected subgraph of G, and let $S = T|_H$. Then, for all $u, v \in V(H)$, we have $u \prec_S v \implies u \prec_T v$.*

If H is a convex subgraph, we even have $u \prec_S v \iff u \prec_T v$.

Proof. Let π be a topological ordering of T. Then $\sigma = \pi|_{V(H)}$ is a topological ordering of S by definition. If $u \not<_\sigma v$, then neither $u \prec_S v$ nor $u \prec_T v$, and we are done. Suppose $u <_\sigma v$ from now on. This implies $v \not\prec_S u$ and $v \not\prec_T u$.

Suppose $u \prec_S v$ and $u \not\prec_T v$ for the sake of contradiction. Since also $v \not\prec_T u$, by corollary 2.13, there is a vertex set A that precedes u and v in π, and A separates u from v in G. By observation 2.1, $A' = A \cap V(H)$ separates u from v in H, and A' clearly precedes u and v in σ. So, by lemma 2.12, we have $u \not\prec_S v$, a contradiction.

[5]Our definition of convexity coincides with Farber and Jamison's [FJ86] when restricted to chordal graphs. For non-chordal graphs, our definition does *not* always induce a *convex geometry* in their sense.

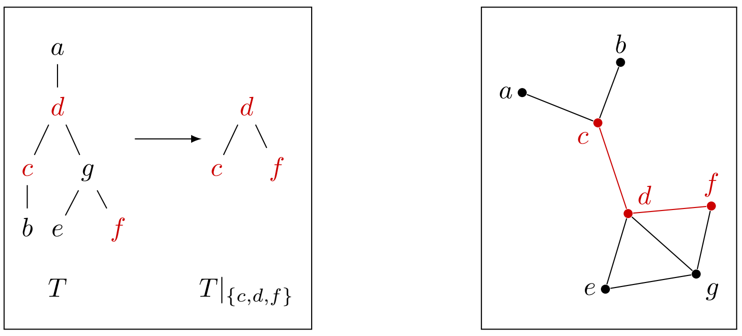


Figure 2.5.: A projection of a search tree to (the subgraph induced by) $\{c, d, f\}$. The underlying graph is shown to the right.

Now let H be a convex subgraph of G and suppose $u \prec_T v$ and $u \not\prec_S v$ for the sake of contradiction. Recall that also $v \not\prec_S u$. Again, by corollary 2.13, there is a set $A \subseteq V(H)$ that precedes u and v in σ and separates u from v in H. Note that A also separates u from v in G (by convexity and observation 2.17), and clearly also precedes u and v in π, so $u \not\prec_T v$, a contradiction. □

Corollary 2.19. *Let T be search tree on G, let H be a convex subgraph of G, and let $v, p \in V(H)$ such that v is a child of p in T. Then v is still a child of p in $T|_H$.*

Observe that corollary 2.19 does *not* necessarily hold when H is non-convex. Consider, for example, the nodes b and c in figure 2.6.

We finish with a discussion of the relation between rotations and projected search trees.

Lemma 2.20. *Let T be a search tree on a graph G, let u be a node with parent v in T, and let $T' = \mathrm{rot}(T, u, v)$. Let $U \subseteq V(G)$ be convex. Then $T'|_U = \mathrm{rot}(T|_U, u, v)$ if $u, v \in U$, and $T'|_U = T|_U$ otherwise.*

Proof. Observe that there exists a topological ordering π of T where v directly precedes u. Let π' be the permutation obtained by swapping u and v in π. Then π' is a topological ordering of T' by lemma 2.14. By definition, we have that $\rho = \pi|_U$ and $\rho' = \pi'|_U$ are topological orderings of $T|_U$ and $T'|_U$. If $u \notin U$ or $v \notin U$, then $\rho = \rho'$ and thus $T'|_U = T|_U$, as desired.

Suppose now $u, v \in U$, then ρ' is obtained from ρ by swapping u and v. Observe that u is a parent of v in $T|_U$ by corollary 2.19 (using convexity of U). This means that $\mathrm{rot}(T|_U, u, v)$ exists and admits ρ' as a topological ordering, by lemma 2.14. Thus, we have $\mathrm{rot}(T|_U, u, v) = T'|_U$, as desired. □

Remark. It is not hard to show that our definition of projections is equivalent to the one used by Cardinal, Pournin, and Valencia-Pabon [CPVP22], though theirs was restricted to chordal graphs. Notably, they already observed that lemma 2.20 holds for chordal graphs [CPVP22, observation 2]. One of the papers partially included in this thesis [BGKK23] contains yet another equivalent definition that is similar to ours, but is restricted to trees.

2.2.5. Search trees on stars and cliques

We already observed that search trees on *paths* are BSTs. It turns out that search trees on stars and cliques correspond to much simpler combinatorial structures: *partial*

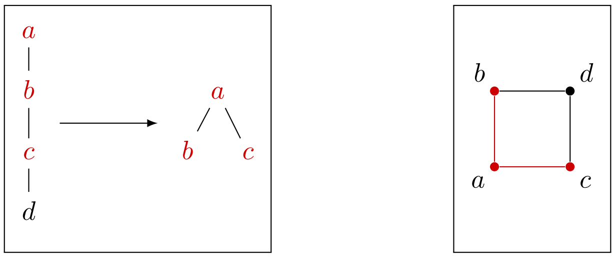


Figure 2.6.: A projection to an induced (but non-convex) subgraph.

permutations and *permutations*, respectively. We now briefly explain this correspondence and make some simple observations. Later on, these special cases will serve as useful examples.

We start with cliques. Let T be a search tree on a clique G with n vertices. Clearly, removing one or more vertices cannot split up G into multiple components; thus, every inner node in T has exactly one child. This means that every search tree on G is degenerate, and thus has a unique topological ordering. A rotation in a search tree on a clique simply swaps the two vertices in the topological ordering (by lemma 2.14). Thus, we can identify search trees on n-vertex cliques with n-element permutations and rotations with *adjacent swaps.*

Observe that there are are precisely $n!$ search trees on G, one for each topological ordering. This property is unique to cliques. Indeed, suppose a connected graph G has two nonadjacent vertices u, v. Take two permutations π, ρ of $V(G)$ where u and v come last, such that π and ρ only differ in the order of u and v. Since the vertex set $V(G) \setminus \{u, v\}$ separates u from v, both π and ρ are topological orderings of the same tree, in which u and v are both leaves (lemma 2.12). Thus, there are strictly less than $n!$ search trees on G.

We now turn to stars. Let G be a star with n vertices, and denote by c its center vertex. Suppose we construct a search tree on G by successive vertex removal. As long as we only remove leaves, the graph is not split, i.e., a single component remains. However, as soon as we remove c, the graph splits into isolated vertices, and we are done. Thus, a search tree on G looks as follows. The strict ancestors of c (if any) form a degenerate prefix of the search tree, and the strict descendants of c are all children of c. See figure 2.7 for examples.

Observe that each search tree is uniquely determined by the degenerate prefix (specifically, the maximal prefix that does not contain c), which corresponds to a *partial permutation* of $V(G) \setminus \{c\}$. Denote this prefix by P in the following.

Consider now a rotation (see figure 2.7). If the rotation does not involve c, then it must involve two nodes of P (all other nodes are leaves and cannot be rotated with each other). The two nodes are then simply swapped. This corresponds to an adjacent swap in the corresponding partial permutation.

If we rotate c with one of its children v, then v becomes the parent of c, so v is added at the end of P. If we rotate c with its parent v, the opposite happens, namely v is removed from P. This corresponds to adding an element at the end of the partial permutation, or removing its last element.

In conclusion, search trees on an n-vertex star can be identified with partial permutations of an $(n-1)$-element set, and each rotation corresponds to an adjacent swap, appending an element, or removing the last element. Clearly, the number of search trees on an n-vertex star is

$$\sum_{k=0}^{n-1} \binom{n-1}{k} \cdot k! = (n-1)! \cdot \sum_{k=0}^{n-1} \tfrac{1}{k!} \in \Theta((n-1)!).$$

2.2.6. Search tree prefixes and torsos

Observe that a prefix P of a search tree on G may not be a valid search tree on $G[P]$; indeed, $G[P]$ may not even be connected. In this section, we discuss *torsos*, which are

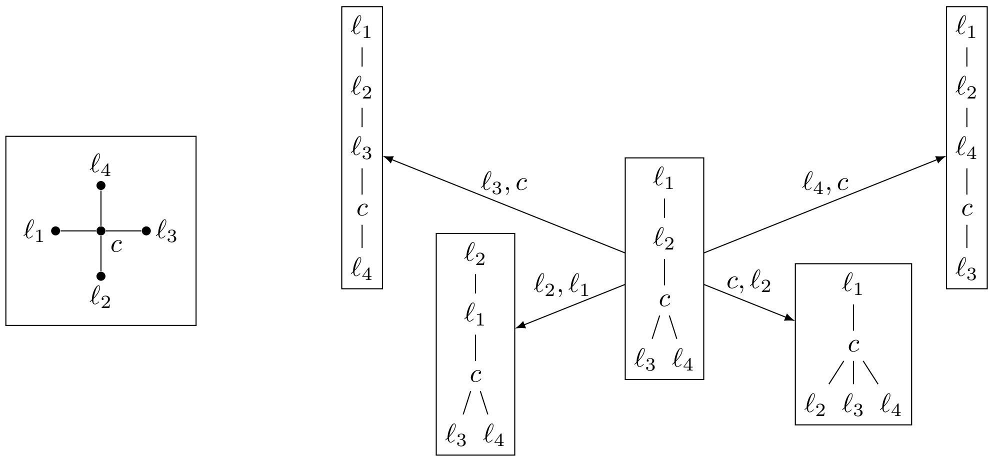


Figure 2.7.: A star on five vertices (left) and a search tree on the star with all possible rotations (right).

essentially supergraphs of $G[P]$ that admit P as a search tree. Torsos will make brief appearances in all three parts of the thesis.

Let G be a graph and $A \subseteq V(G)$. Let E_A be the set of pairs of vertices $\{u, v\} \subseteq A$ where there exists a path between u and v in G with all inner vertices outside of A. In particular, $E(G[A]) \subseteq E_A$. The *torso* of A in G, denoted by $\text{torso}_G(A)$, is the graph H with $V(H) = A$ and $E(H) = E_A$.

torso

Observe that a subgraph H of G is *convex* if and only if $\text{torso}_G(V(H)) = H$. The following is the main result of this section.

Theorem 2.21. *Let T be a search tree on G and let P be a prefix of T. Then P is a search tree on $H = \text{torso}_G(V(P))$.*

On the other hand, each search tree on H is a prefix of some search tree T on G.

We prove a few technical facts about torsos first.

Lemma 2.22. *Let G be a graph, let $A \subseteq V(G)$, and let $H = \text{torso}_G(A)$. Then, for each path P in G, restricting P to A (removing all vertices in $V(G) \setminus A$ from P) yields a path in H.*

Proof. Let P' be the restriction of P to A, and let u, v be two consecutive vertices on P'. If u, v are also consecutive on P, then u, v are adjacent in G, and thus adjacent in H. Otherwise, let $v_1, v_2, \dots, v_k$ be the sequence of vertices strictly between u and v in P. Then $u, v_1, v_2, \dots, v_k, v$ is a path in G with all inner vertices in $V(G) \setminus A$, hence u, v are adjacent in H by definition. This implies that P' is indeed a path in H. ☐

Lemma 2.23. *Let G be a connected graph and let $A \subseteq V(G)$. Let $x, y \in A$ and $B \subseteq A$. Then B separates x from y in G if and only if B separates x from y in $H = \text{torso}_G(A)$.*

Proof. Suppose B separates x from y in G. This means that each path between x and y in G intersects B. Clearly, each path P between x and y in H can be extended into a path P' in H with $V(P) = V(P') \cap A$, so P must also intersect B.

On the other hand, suppose B does not separate x from y in G. Then there exists a path P between x and y in G that does not intersect B. Lemma 2.22 implies that $V(P) \cap A$ induces a path in H that also does not intersect B. $\square$

Proof of theorem 2.21. We start with the first statement. It is easy to see that T has a topological ordering π such that $V(P)$ induces a prefix σ of π. Let S be the search tree on H with topological ordering σ.

We show that $x \prec_P y \iff x \prec_S y$, implying $P = S$. Consider $x, y \in V(P)$ with $x <_\sigma y$. By lemma 2.12, we have $x \prec_P y$ (resp., $x \prec_S y$) if and only if there exists no set $A \subseteq V(P)$ that precedes x in σ and separates x from y in G (resp., in H). By lemma 2.23, we have that A separates x from y in G if and only if it separates them in H.

We now prove the second statement in a similar way. Let S be a search tree on H with topological ordering σ, and let T be some search tree on G with a topological ordering that starts with σ. Clearly $V(S)$ induces a prefix P of T. Applying lemma 2.12 as above, we get $P = S$. $\square$

Remark. Torsos are useful in the context of *tree-width* (see, for example, Korhonen and Lokshtanov [KL23]). Theorem 2.21 is implicit in the literature if one considers the equivalence between search trees and tree decompositions (see section 2.4).

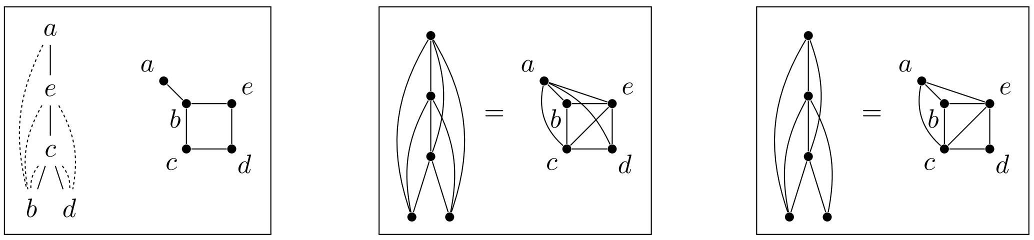
2.3. Tree closures, trivially perfect graphs, and tree-depth

In this section, we introduce the concept of the *closure* of a rooted tree, which allows us to connect search trees with *trivially perfect graphs* and the original definition of *tree-depth*. While these connections have been made before, we show a self-contained explicit characterization of the two concepts with STGs. Apart from that, these results will be used in chapter 7 (until then, this section can be safely skipped).

Let T be a rooted tree. The *closure* of T, denoted $\mathrm{cl}(T)$, is the graph G with $V(G) = V(T)$ and $E(G) = \{\{u, v\} \mid u \preceq_T v\}$. See figure 2.8 (b) for an example. closure

Observe that for each search tree T on G, we have $E(G) \subseteq E(\mathrm{cl}(G))$. In fact, it is easy to see that lemma 2.3 can be written equivalently as follows.

Lemma 2.24. *Let T be a rooted tree and G be a connected graph with $V(T) = V(G)$. Then T is a search tree on G if and only if*



(a) Search tree T on G (b) Closure $\mathrm{cl}(T)$ (c) Boundary closure $\mathrm{bcl}(T)$

Figure 2.8.: Examples of closure and boundary closure. In (a), the dashed edges are the graph edges, superimposed over the search tree.

(i) $E(G) \subseteq E(\mathrm{cl}(T))$*; and*

(ii) for each $v \in V(T)$*, the subgraph* $G[T_v]$ *is connected.*

The following technical lemma will be useful later.

Lemma 2.25. *Each rooted tree* T *is a search tree on* $\mathrm{cl}(T)$*.*

Proof. By lemma 2.24, it suffices to show that each rooted subtree T_v of T induces a connected subgraph of $\mathrm{cl}(T)$. This is clearly the case, since every edge of T is also an edge of $\mathrm{cl}(T)$. □

trivially perfect

Trivially perfect graphs. A connected graph is *trivially perfect* if it is the closure of some rooted tree T. A disconnected graph is trivially perfect if each of its components is trivially perfect.

This graph class was first described by Wolk [Wol62, Wol65] as *comparability graphs of trees*. There are many alternative characterizations; the most well-known one, due to Golumbic [Gol78], is that trivially perfect graphs are precisely those graphs where, in each induced subgraph, the number of maximal cliques is precisely the size of the maximum independent set. This property makes it trivial to show that these graphs are *perfect*, hence the name. (A graph is *perfect* if, for each induced subgraph, the chromatic number and the clique number are equal.)

We now relate chromatic number and clique number of a connected trivially perfect graph to the tree that generates it.

Lemma 2.26. *Let* T *be a rooted tree, and let* $G = \mathrm{cl}(T)$*. Then the clique number* $\omega(G)$ *and the chromatic number* $\chi(G)$ *of* G *are both precisely* $\mathrm{height}(T)$*.*

Proof. Clearly, we have $\omega(G) \leq \chi(G)$. Since the root path of any vertex forms a clique in G, taking a maximum-depth vertex gives us $\omega(G) \geq \mathrm{height}(T)$. On the other hand, a proper coloring of G can be found by coloring each vertex according to its depth in T, so $\chi(G) \leq \mathrm{height}(T)$. □

A well-known problem, of particular interest to us, is the *trivially perfect completion* problem, in which we need to determine the minimum number of additional edges needed to make a given graph G trivially perfect. In other words, given G, we need to find a rooted tree T with $V(T) = V(G)$ such that G is a subgraph of $\mathrm{cl}(T)$ and the number of edges in $\mathrm{cl}(T)$ is minimal. Denote that number of edges by $\mathrm{TPC}(G)$. The trivially perfect completion problem is NP-hard [Yan81].[6]

TPC(G)

It has been observed before [HBB$^+$21] that the trivially perfect completion problem is equivalent to the *unweighted static optimum* problem, i.e., finding a search tree minimizes expected search cost when the input distribution is uniform. Hence, the latter problem is also NP-hard. We will give a simple proof of the equivalence in section 3.1.

tree-depth

Tree-depth. In the introduction, we characterized the *tree-depth* of a graph G as the minimum height of a search tree on G. The original definition was slightly different, as follows. Let G be a connected graph. The *tree-depth* of G, denoted $\mathrm{td}(G)$, is the

[6]Yannakakis [Yan81] showed that the related *chordal completion* problem is NP-hard, and it is easy to adapt the proof.

minimum height of a rooted tree T with $V(T) = V(G)$ such that G is a subgraph of $\mathrm{cl}(T)$ [NOdM06]. By lemma 2.26, the minimum clique number of a trivially perfect supergraph of G is precisely $\mathrm{td}(G)$. As mentioned in the introduction, computing the tree-depth is NP-hard [Pot88].

Characterization via STGs. Observe that the definitions of $\mathrm{TPC}(G)$ and $\mathrm{td}(G)$ both involve rooted trees whose closure is a supergraph of G (i.e., trivially perfect supergraphs of G). We now show that we can actually restrict ourselves to *search trees on G*.

Theorem 2.27. *Let T be a rooted tree and let G be a connected subgraph of $\mathrm{cl}(T)$. Then there exists a search tree T^* on G with $E(\mathrm{cl}(T^*)) \subseteq E(\mathrm{cl}(T))$.*

Proof. Observe that T is a search tree on $H = \mathrm{cl}(T)$ by lemma 2.25, and let $T^* = T|_G$. Now lemma 2.18 directly implies the statement. □

Theorem 2.27 implies that all *edge-minimal* trivially perfect supergraphs of G are generated by search trees on G. Since $\mathrm{td}(G)$ and $\mathrm{TPC}(G)$ are both defined via edge-monotone properties on trivially perfect supergraphs (clique number and edge number, respectively), we have

Corollary 2.28. *Let G be a connected graph. Then,*

- $\mathrm{td}(G)$ *is the minimum height of a search tree on G; and*
- $\mathrm{TPC}(G)$ *is the minimum number of edges in* $\mathrm{cl}(T)$ *among search trees T on G.*

Remark. Corollary 2.28 is not a new result. The equivalence of tree-depth and *minimum elimination tree height* is well-known, and elimination trees are the same as search trees on graphs [DKKM94]. The characterization of TPC seems lesser known, but has essentially been observed by Høgemo, Bergougnoux, Brandes, Paul, and Telle [HBB+21]. Theorem 2.27, however, has not been stated before, as far as the author is aware.

2.4. Boundary closures, chordal graphs, and tree-width

In the following, we introduce the concept of *boundary closures*, and connect it to *chordal graphs* and *tree-width*. Again, these results are mostly implicit in earlier work (e.g., Bodlaender, Gilbert, Hafsteinsson and Kloks [BGHK95]), but, to the author's knowledge, have not been stated before in this particular manner. Our main result, used in chapter 5, is the following.

Theorem 2.29. *Let G be a connected graph. Then the tree-width of G is the minimum k such that G admits a k-cut search tree.*

boundary closure

We start with definitions. Let T be a search tree on a connected graph G. The *boundary closure* of T is the graph $H = \mathrm{bcl}_G(T)$ with $V(H) = V(G)$ and $E(H) = \{\{v, u\} \mid v \in V(G), u \in \partial_G(T_v)\}$. Observe that G is a subgraph of H, and that H is a subgraph of $\mathrm{cl}(T)$. See figure 2.8 (c) for an example.

The usual definition of *chordal graphs* is that a graph is chordal if and only if all induced cycles are triangles. In other words, each cycle of length four or more has a *chord*, i.e., an extra edge between two vertices of the cycle.

perfect elimination ordering

chordal graph

chordal completion

We use a different characterization due to Rose [Ros70]. Let G be a graph and π be a permutation of $V(G)$. We say π is a *perfect elimination ordering* if for each $v \in V(G)$, the set of vertices that are adjacent to v and precede v in π induce a clique in G.[7] A graph is *chordal* if and only if it admits a perfect elimination ordering.

A *chordal completion* of a graph G is a chordal graph H with $V(H) = V(G)$ and $E(H) \subseteq E(G)$, i.e., a chordal graph obtained by adding edges to G. Observe that every graph has at least one chordal completion: the complete graph. Chordal completions are also called *triangulations* and are a well-studied concept with many applications, e.g. in matrix factorization [Ros72].

Characterization via STGs. Our first goal is to obtain an analogue of theorem 2.27, i.e., edge-minimal chordal completions can be characterized by STGs. We start with some technical lemmas.

Lemma 2.30. *Let T be a search tree on a graph G. For each $v \in V(G)$, the set $\partial_G(T_v) \cup \{v\}$ induces a clique of $\mathrm{bcl}_G(T)$.*

Proof. All vertices of $\partial_G(T_v)$ are adjacent to v in $\mathrm{bcl}_G(T)$ by definition. Now take $a, b \in \partial_G(T_v)$ with $a \prec_T b$. Since $a \in \partial_G(T_v)$, some node $x \in V(T_v) \subseteq V(T_b)$ is adjacent to a, implying $a \in \partial_G(T_b)$. This means that a and b are adjacent in $\mathrm{bcl}_G(T)$, as desired. □

Lemma 2.31. *Let T be a search tree on a chordal graph G, and let π be a topological ordering of T that is a perfect elimination ordering of G. Then, for each $v \in V(T)$ and $u \in \partial_G(T_v)$, we have $\{u, v\} \in E(G)$.*

Proof. We proceed by induction on $k = \mathrm{height}(T_v)$. If $k = 1$, then T_v consists only of v, and thus v is adjacent to $\partial(T_v)$ by definition.

Now let $k \geq 2$. Suppose $u \in \partial(T_v)$. Then there is some $x \in V(T_v)$ that is adjacent to u. If $x = v$, then we are done. Otherwise, let c be the child of v such that $x \in V(T_c)$.

We have $\{u, v\} \subseteq \partial(T_c)$ by observation 2.7, so c is adjacent to u and v by induction. Since π is a topological ordering of T, we have $u <_\pi v <_\pi c$. Combining that with the fact that π is a perfect elimination ordering of G yields $\{u, v\} \in E(G)$, as desired. □

Lemma 2.32. *Let T be a search tree on a chordal graph G. Then $G = \mathrm{bcl}_G(T)$ if and only if each topological ordering of T is a perfect elimination ordering of G.*

Proof. We start with the "only if" direction. Suppose $G = \mathrm{bcl}_G(T)$, and let π be a topological ordering of T. We show that π is a perfect elimination ordering.

Let $v \in V(G)$, and let A be the set of neighbors of v in G that precede v in π. Since π is a topological ordering, and by lemma 2.3, all vertices of A are ancestor nodes of v in T, implying that $A \subseteq \partial(T_v)$. Lemma 2.30 implies that $G[A]$ is a clique.

We now continue with the "if" direction. Let π be a topological ordering of T, and suppose that π is also a perfect elimination ordering of G. Then lemma 2.31 implies that

[7] Perfect elimination orderings are usually defined in reverse order; our definition is more convenient here.

$E(G) \supseteq \operatorname{bcl}_G(T)$. But we also have $E(G) \subseteq \operatorname{bcl}_G(T)$ by definition, so $G = \operatorname{bcl}_G(T)$, as desired. □

Lemma 2.33. *Let T be a search tree on a graph G, and let H be a subgraph of G. Then $E(\operatorname{bcl}_H(T|_H)) \subseteq E(\operatorname{bcl}_G(T))$.*

Proof. Write $S = T|_H$. By lemma 2.18, we have $V(S_v) \subseteq V(T_v)$ for each $v \in V(H)$. Thus $\partial_H(S_v) \subseteq \partial_H(T_v) \subseteq \partial_G(T_v)$, and the statement follows. □

We are now ready to show the characterization.

Theorem 2.34. *Let G be a connected graph and let H be a chordal completion of G. Then there exists a search tree T on G such that $E(\operatorname{bcl}_G(T)) \subseteq E(H)$.*

Proof. Let π be a perfect elimination ordering of H. By lemma 2.32, the search tree T' on H with topological ordering π satisfies $\operatorname{bcl}_H(T') = H$. Let $T = T'|_G$, i.e., let T be the search tree on G with topological ordering π. By lemma 2.33, we have $E(\operatorname{bcl}_G(T)) \subseteq E(\operatorname{bcl}_G(T')) \subseteq E(\operatorname{bcl}_H(T')) = E(H)$. □

Like in section 2.3, there are two graph invariants that we can now state in terms of search trees on trees. First, the *chordal completion problem* consists of finding a chordal completion H of a given graph G such that the number of edges in H is minimized. Let $\operatorname{CC}(G)$ denote this minimum number of edges. From theorem 2.34, we immediately get: $\boxed{\operatorname{CC}(G)}$

Corollary 2.35. *Let G be a connected graph. Then $\operatorname{CC}(G)$ is precisely the minimum number of edges in $\operatorname{bcl}_G(T)$ among search trees T on G; or, equivalently, the minimum value $\sum_{v \in V(T)} |\partial(T_v)|$ among search trees T on G.*

Second, the *tree-width* of a graph G, written $\operatorname{tw}(G)$, is the minimum clique number of a chordal completion of G, minus one. Again, theorem 2.34 implies that we can restrict our attention to boundary closures of search trees on G, instead of considering all chordal completions. Moreover, the clique number of boundary closures can be nicely characterized as follows. $\boxed{\text{tree-width}}$

Lemma 2.36. *Let G be a chordal graph with a search tree T such that $\operatorname{bcl}_G(T) = G$. Then the clique number $\omega(G)$ of G is the minimum k such that T is $(k-1)$-cut.*

Proof. We show that $\omega(G) > k$ if and only if T is not $(k-1)$-cut.

Suppose $\omega(G) > k$, and let K be a clique of size $k+1$ in G. Then, by lemma 2.3, there is some permutation $c_1, c_2, \dots, c_{k+1}$ of K such that $c_1 \prec_T c_2 \prec_T \dots \prec_T c_{k+1}$. By definition, $V(K) \setminus \{c_{k+1}\} \subseteq \partial(T_{c_{k+1}})$, so $|\partial(T_{c_{k+1}})| \geq k$, implying that T is not $(k-1)$-cut.

Now suppose T is not $(k-1)$-cut. Then there exists a $v \in V(T)$ with $|\partial_G(T)| \geq k$. By lemma 2.30, the set $\partial_G(T) \cup \{v\}$ induces a clique of $G = \operatorname{bcl}_G(T)$, so $\omega(G) > k$. □

We can now prove our main theorem.

Theorem 2.29. *Let G be a connected graph. Then the tree-width of G is the minimum k such that G admits a k-cut search tree.*

Proof. Let G be a graph. By theorem 2.34, the tree-width of G is the minimum value of $\omega(\mathrm{bcl}_G(T)) - 1$ among all search trees T on G. By lemma 2.36, this is precisely the minimum k such that T is k-cut. □

The following well-known fact can be easily proved with our characterization, and will be useful in chapter 5.

Lemma 2.37 (Bodlaender [Bod96]). *For each graph G, we have $|E(G)| \leq \mathrm{tw}(G) \cdot |V(G)|$.*

Proof. Let $k = \mathrm{tw}(G)$ and let T be a k-cut search tree on G. For each $v \in V(G)$, the number of ancestors of v in T that are adjacent to v is at most $|\partial(T_v)| \leq k$. Summing over all vertices covers all edges, and yields $k \cdot |V(G)|$. □

tree decomposition

Tree decompositions. Tree-width is frequently defined differently, via so-called *tree decompositions* [RS86]. A tree decomposition of a graph G consists of an unrooted tree D and function $X\colon V(D) \to 2^{V(G)}$ that maps vertices of D to so-called *bags*. A valid tree decomposition must have the following properties:

(i) The bags cover all vertices, i.e., we have $\bigcup_{v \in V(D)} X(v) = V(G)$.

(ii) For each edge $\{x, y\}$, there is some $v \in V(D)$ with $x, y \in X(v)$.

(iii) For all $u, v_1, v_2 \in V(D)$, if u lies on the path between v_1 and v_2 in D, then $X(v_1) \cap X(v_2) \subseteq X(u)$.

The *width* of the decomposition is $\max_{v \in V(D)} |X(v)| - 1$, and the tree-width is the minimum width of a decomposition.

path-width $\mathrm{pw}(G)$

If D is a path, then we have a *path decomposition*, and the *path-width* $\mathrm{pw}(G)$ of a graph is the minimum width of a path decomposition [RS83]. Path-width will be important in chapter 12.

Observe that search trees on a graph G can be seen as tree decompositions, as follows. Take the search tree T itself (or its unrooting) as the decomposition tree D, and let the bag of each vertex v be $\partial_G(T_v) \cup \{v\}$. It can be seen that properties (i) to (iii) indeed hold. Clearly, the width of the decomposition is at most k if the search tree is k-cut. On the other hand, every tree decomposition of width k can be transformed into one that resembles a k-cut search tree, with a series of local modifications. We omit the rather inelegant proof of the last statement.

Elimination trees. We have remarked before that search trees on graphs are also called *elimination trees*. Elimination trees can be defined using the so-called *elimination game*, which works as follows. Take a connected graph G, and delete the vertices one-by-one in some given order. Whenever a vertex is deleted, its neighborhood is made a clique. The edges introduced in this way form a chordal completion H of G (the *elimination graph*), and the order of deletion is a reverse perfect elimination ordering of H. Elimination trees are then defined as the "tree structure" of the elimination game [BGHK95]. Table 2.1 compares elimination trees and related vocabulary with search trees on graphs.

It should be noted that elimination trees are sometimes defined slightly differently (see definition 4 in Sanchez Villaamil [SV17] and the references there). We use the name *perfect*

Elimination tree	**STG**	
Elimination ordering	Topological ordering	
Edge-minimal trivially perfect completion	Closure	(theorem 2.27)
Tree-depth	Minimum-height STG	(corollary 2.28)
Elimination graph Edge-minimal chordal completion	Boundary closure	(theorem 2.34)
Tree-width Tree decomposition	k-cut STG	(theorem 2.29)

Table 2.1.: Informal list of some concepts and their equivalents in STG language.

elimination tree here to maintain the distinction from our definition. A perfect elimination tree is a search tree on a *chordal* graph G such that $G = \mathrm{bcl}(T, G)$. By lemma 2.32, we can equivalently say that all topological orderings of T must be perfect elimination orderings of G.

It is easy to see that there exist search trees on chordal graphs that are not perfect elimination trees; for example, trees are chordal, and elimination trees on trees are perfect only if they are 1-cut. The difference of the two definitions is further emphasized by the fact that computing minimum-height *perfect* elimination trees on chordal graphs is possible in polynomial time [Liu89], but computing the minimum height elimination tree (i.e., determining the tree-depth) of a chordal graph is NP-hard [DN06].

Part I.

Static search trees

3. Expected-case optimal search trees

In the first part of this thesis, we study the following problem. Given is a graph G and a probability distribution p on its vertices. We want to build a search tree T on G that minimizes the *expected search time* with respect to p.

More formally, we say a tuple (G, w) is a *weighted graph* when G is a graph and $w\colon V(G) \to \mathbb{R}_{\geq 0}$ is a *weight function* that assigns a nonnegative *weight* to each vertex. We sometimes do not distinguish between G and (G, w), e.g., "(G, w) is a tree" means that G is a tree. The *cost* of a search tree T on (G, w) is defined as

weighted graph

weight function

cost

$$\mathrm{cost}(T, w) = \sum_{v \in V(T)} \mathrm{depth}_T(v) \cdot w(v).$$

Observe that if w is a probability distribution, then $\mathrm{cost}(T, w)$ is precisely the expected search time in T if the input is distributed according to w. On the other hand, if w is integral, then $\mathrm{cost}(T, w)$ is the total time of searching each vertex v exactly $w(v)$ times.

A search tree T which minimizes $\mathrm{cost}(T, w)$ is an *optimal search tree* on (G, w), also called the *static optimum*, and we denote its cost as $\mathrm{StOPT}(G, w)$. We call the problem of computing an optimal search tree the *optimal static search tree problem.*

optimal search tree

$\mathrm{StOPT}(G, w)$

Overview. In this chapter, we summarize our results on the optimal static search tree problem along with known result for the BST case, deferring the details to chapters 4 to 7. We group the results by technique. First, we use a weighted variant of *centroid trees* (introduced in chapter 1) to obtain a fast 2-approximation algorithm for trees. This algorithm can be seen as a generalization of a BST heuristic due to Mehlhorn [Meh75, Fre75, Meh77].

Second, we discuss a dynamic programming approach based on Knuth's well-known optimal static BST algorithm [Knu71]. From this, we obtain polynomial-time approximation algorithms for trees and graphs with bounded tree-width, and an exact (but exponential-time) algorithm for arbitrary graphs.

Third, we use the connection between STGs and *edge-query trees* (see section 1.3) to transfer some results in the edge-query tree model to our STG model. We obtain a pseudo-polynomial exact algorithm, a fast approximation algorithm, and NP-hardness of the problem when restricted to bounded-tree-width graphs.

Fourth, we discuss the *unweighted* case (all weights are equal). The *unweighted static search tree problem* is equivalent to the *trivially perfect completion problem* (see section 2.3), which implies NP-hardness.

Finally, we briefly discuss the issue of handling *zero-weight* vertices, for which we give a general technique. Then, we summarize the main takeaways and pose some open questions.

Table 3.1 shows all these results, ordered by generality, approximation quality, and performance. In the following, we let n denote the number of vertices in the underlying graph.

Description	**Graphs**	**Weights**	**Approx.**	**Time**
Complete BST	path	uniform	exact	$\mathcal{O}(n)$
Centroid trees [Meh77]	path	arbitrary	$1 + o(1)$	$\mathcal{O}(n)$
Basic DP [Knu71]	path	arbitrary	exact	$\mathcal{O}(n^2)$
Centroid trees (thm. 3.2)	tree	$0 \cup [1, W]$	2	$\mathcal{O}(n \log\log(n + W))$
EQT-based DP (thm. 3.7)	tree	$0 \cup [1, W]$	exact	$\mathcal{O}(W^{3.42} \cdot n^{4.42})$
Centroid trees (thm. 3.1)	tree	arbitrary	2	$\mathcal{O}(n \log n)$
k-cut DP (thm. 3.4)	tree	arbitrary	$1 + \varepsilon$	$\mathcal{O}(n^{\lceil 2/\varepsilon \rceil + 1})$
EQT-based DP (thm. 3.8)	tree	arbitrary	$1 + \varepsilon$	$\mathcal{O}((\frac{1}{\varepsilon})^{3.42} \cdot n^{7.84})$
k-cut DP (thm. 3.5)	$\mathrm{tw} = t$	arbitrary	$1 + \varepsilon$	$n^{\mathcal{O}(t/\varepsilon)}$
Basic DP (thm. 3.3)	arbitrary	arbitrary	exact	$\mathcal{O}(n^3 \cdot N \log N)$
EQT reduction (thm. 3.6)	$\mathrm{tw} \geq 15$	arbitrary	exact	NP-hard
Triv. perf. compl. (thm. 3.9)	arbitrary	uniform	exact	NP-hard

Table 3.1.: Overview of the most important algorithms and hardness results for the optimal static STG problem. We denote by n the number of vertices in the underlying graph G, and by N the number of connected subgraphs of G.

Weighted centroid trees. Mehlhorn [Meh75] gave the following heuristic for the optimal static BST problem (described here in the search-trees-on-paths setting). As the root, select the vertex that splits the path into two parts such that the weight difference between the two parts is minimized. Then, recurse on the two parts. The resulting search tree is a $(1 + o(1))$-approximation [Meh77] and can be computed in linear time [Fre75].

This idea can be generalized to trees using a weighted variant of *centroid vertices* (discussed in section 1.1). A *centroid* of a weighted tree is a vertex that splits the tree into parts with weight at most half of the total weight. It can be shown that every weighted tree has a centroid, so we can build a search tree by recursively choosing centroids as subtree roots, just like in Mehlhorn's heuristic. We call such a search tree a *centroid tree.* Chapter 4 is dedicated to the study of centroid trees.

It is not hard to see that a centroid can be found in $\mathcal{O}(n)$ time. Thus, the naive algorithm to compute centroid trees runs in time $\mathcal{O}(n^2)$. We give an improved $\mathcal{O}(n \log n)$-time algorithm using techniques from dynamic graph algorithms, and show its optimality in a decision tree model. Recall that a centroid tree on a *path* can be constructed in only $\mathcal{O}(n)$ time, implying a jump in complexity when going from paths to trees.

We also show a tight approximation ratio of 2; again, this is worse than the ratio $1 + o(1)$ for the BST algorithm. Mehlhorn's analysis uses the key fact that the cost of the *optimal* static BST is asymptotically equal to the so-called *Shannon entropy* of the input distribution (defined in section 4.4). This is not true in the STT setting: the entropy is not a lower bound for optimal static search trees. (It is, however, an upper bound, as we show in section 4.4).

Our proof instead directly compares the cost of centroid STTs and optimal STTs. We now state the first result.

Theorem 3.1. *Given a weighted tree (G, w) with n vertices, we can compute a search tree T on G such that* $\mathrm{cost}(T, w) \leq 2 \cdot \mathrm{StOPT}(G, w)$*, in time $\mathcal{O}(n \log n)$.*

We prove theorem 3.1 in chapter 4, along with tightness and hardness results. In section 4.6, we also give a much simpler centroid-based 4-approximation algorithm with the same running time.

When the *spread* of the input weights is bounded, we obtain the following improvement (section 4.7).

Theorem 3.2. *Let (G, w) be a given weighted tree with n vertices such that $w\colon V(G) \to \{0\} \cup [1, W]$ for some $W \in \mathbb{R}$. We can compute a search tree T on (G, w) such that $\operatorname{cost}(T, w) \leq 2 \cdot \operatorname{StOPT}(G, w)$ in time $\mathcal{O}(n \log\log(n + W))$.*

Note that, in particular, the algorithm runs in time $\mathcal{O}(n \log\log n)$ if all weights are polynomial in n.

The algorithm in theorem 3.2 is actually *output-sensitive*: If the height of the output tree is h, then it runs in time $\mathcal{O}(n \log h)$.

Dynamic programming on subgraphs. Knuth's dynamic programming algorithm [Knu71] for optimal static BST computes the optimum on each sub-path (i.e., each interval of keys), by trying all possible roots and using the already computed optimum for the left and right subtrees. The algorithm is correct because a rooted subtree T_v of an optimal static BST is also optimal on $V(T_v)$. Since there are $\mathcal{O}(n^2)$ subproblems, and we need to try at most n possible roots for each of them, the naive implementation runs in $\mathcal{O}(n^3)$ time. This can be reduced to $\mathcal{O}(n^2)$ using a certain monotonicity property [Knu71, Yao80].

The optimal-subtree property clearly also holds for STGs, and the above algorithm is easily generalized by considering *connected induced subgraphs* as subproblems (instead of sub-paths). However, the number of subproblems can be exponential even if G is a tree: For example, in a *star*, each subset of leaves plus the center induces a connected subgraph. Still, observe that the *number of search trees* can be super-exponential (see section 2.2.5). Thus, we still obtain an improvement over the brute-force approach of simply trying every search tree.

Theorem 3.3. *Given a weighted graph (G, w) with n vertices, we can find an optimal static STG on (G, w) in time $\mathcal{O}(n^3 \cdot N \log N)$, where $N \leq 2^n$ is the number of connected induced subgraphs of G.*

More importantly, we can modify the algorithm to obtain a polynomial-time approximation scheme (PTAS) when G is a tree. The idea is to restrict the dynamic programming to a subset of search trees – specifically, k-cut trees (as defined in section 2.2.1). This makes the running time polynomial, but we cannot guarantee that the optimum is a k-cut search tree, only that some k-cut search tree approximates the optimum.

Theorem 3.4. *Given a weighted tree (G, w) and an integer $k \geq 2$, we can compute a search tree T on G in time $\mathcal{O}(k \cdot n^{k+1})$ such that*

$$\operatorname{cost}(T, w) \leq \left(1 + \tfrac{1}{\lfloor k/2 \rfloor}\right) \cdot \operatorname{StOPT}(G, w).$$

This idea generalizes to graphs with bounded tree-width, since they admit k-cut search trees if $k \geq \operatorname{tw}(G)$ (see section 2.4). The approximation factor scales less well as the tree-width grows: To get a $(1 + \varepsilon)$-approximation, we need to set $k \approx \frac{2t}{\varepsilon}$.

Theorem 3.5. *Given a weighted graph (G, w) with tree-width t and an integer $k \geq 3t + 1$, we can compute a search tree T on G in time $\mathcal{O}(k^3 \cdot n^{k+3})$ such that*

$$\operatorname{cost}(T, w) \leq \left(1 + \tfrac{2t+2}{k-3t}\right) \cdot \operatorname{StOPT}(G, w).$$

Note in particular the restriction $k \geq 3t + 1$, so the running time bound is always at least n^{3t+4}.

Results from edge-query trees. We now discuss how some results from edge-query trees (EQTs, see section 1.3) can be transferred to the STG setting. First, Cicalese, Jacobs, Laber, and Molinaro [CJLM11] show that computing optimal static EQTs is NP-hard even on a tree with maximum degree 16. Using the connection between EQTs and search trees on line graphs, in section 7.1 we show that computing optimal static search trees is hard even if the underlying graph has bounded tree-width.

Theorem 3.6. *Computing the optimal static search tree on a given weighted graph is NP-hard, even if the graph has tree-width at most 15.*

The proof of theorem 3.6 is a direct reduction to the optimal static EQT problem. The constant 15 arises as follows: The hardness construction of Cicalese et al. [CJLM11] gives an underlying tree of maximum degree 16. Its line graph thus has clique number 16. Since line graphs of trees are chordal, the tree-width is exactly one less.

Second, the same authors [CJLM14] give a fully-polynomial-time approximation scheme (FPTAS) for finding optimal EQTs on bounded-degree trees, which in turn is based on a pseudo-polynomial exact algorithm. Both algorithms can be adapted to STGs if the underlying graph is a tree. We remark that the degree restriction is not necessary for our results.

Theorem 3.7. *Let (G, w) be a given weighted tree with n vertices such that $w \colon V(G) \to 0 \cup [1, W]$ for some $W \in \mathbb{R}$. We can compute an optimal search tree on (G, w) in time*

$$\mathcal{O}(W^{2/\log(3/2)} \cdot n^{1+2/\log(3/2)} \cdot \log^2(Wn)) \subseteq \mathcal{O}(W^{3.42} \cdot n^{4.42}).$$

Theorem 3.8. *For each $\varepsilon > 0$, there exists an algorithm that computes a search tree T on a given weighted tree (G, w) such that $\operatorname{cost}(T, w) \leq (1 + \varepsilon) \cdot \operatorname{StOPT}(G, w)$, in time*

$$\mathcal{O}\left((\tfrac{1}{\varepsilon})^{2/\log(2/3)} \cdot n^{1+4/\log(2/3)} \cdot \log^2 \tfrac{n}{\varepsilon}\right) \subseteq \mathcal{O}\left((\tfrac{1}{\varepsilon})^{3.42} \cdot n^{7.84}\right).$$

Theorem 3.7 implies that computing an optimal STT is *not* strongly NP-hard, in contrast to computing the optimum in the weighted-cost model (see section 1.2). We prove both theorems in chapter 6. Theorem 3.7 is a dynamic programming algorithm, just like our PTAS (theorem 3.4), though the subproblems are very different. Theorem 3.8 follows from theorem 3.7 with a standard reduction.

Observe that the centroid tree algorithm (theorem 3.1) computes 2-approximations much faster than the PTAS (theorem 3.4) and FPTAS (theorem 3.8). The PTAS is best for approximation ratios between in $[\frac{4}{3}, 2)$, and the FPTAS is best for all smaller approximation ratios.

Unweighted static optima. An interesting special case is when the weight of every vertex is one. We denote the corresponding *unit weight function* by $\mathbb{1}$ (the domain is always implicit), and call $\mathrm{StOPT}(G, \mathbb{1})$ the *unweighted* static optimum of G. Observe that $\mathrm{StOPT}(G, \mathbb{1})$ minimizes the average depth of a node in a search tree on G.

$\mathbb{1}$

unw. static optimum

Høgemo, Bergougnoux, Brandes, Paul and Telle [HBB+21] observed that $\mathrm{StOPT}(G, \mathbb{1})$ is essentially equivalent to the trivially perfect completion problem (see section 2.3). We give a formal proof of this equivalence in section 7.2. Since the trivially perfect completion problem is NP-hard [Yan81], we have:

Theorem 3.9 ([HBB+21])**.** *Computing* $\mathrm{StOPT}(G, \mathbb{1})$ *for a given graph* G *is NP-hard.*

On the positive side, if the underlying graph is a tree, our previously stated results imply that we can compute a 2-approximation in $\mathcal{O}(n \log\log n)$ time (theorem 3.2) and an exact optimal search tree in $\mathcal{O}(n^{4.42})$ time (theorem 3.7).

Zero-weight vertices. All our algorithms can handle arbitrary nonnegative weights, including vertices of weight zero. Unsurprisingly, zero-weight vertices tend to be easier to handle, as their depth does not affect the overall cost. Still, their placement may affect the overall structure of the tree. For example, the center of a *star* is almost always forced to be the root of the optimal search tree, even if it has weight zero.

In section 3.2, we give a general preprocessing algorithm that allows removing most zero-weight vertices. It can be applied to all algorithms given above, leading to speedups when the number of positive-weight vertices is *sublinear*.

Theorem 3.10. *Let* (G, w) *be a given weighted tree with* n *vertices, and let* m *be the number of vertices with positive weight. In* $\mathcal{O}(n)$ *time, we can transform* G *into a tree* G' *with* $V(G') \subseteq V(G)$ *and* $\mathrm{StOPT}(G, w) = \mathrm{StOPT}(G', w)$*, such that* $|V(G')| \leq 2m$ *and* G' *has no zero-weight vertices of degree two or lower.*

Summary and open questions. We start with the tree case. With the FPTAS (theorem 3.8), the theoretical question of approximability can be considered settled. However, the algorithm is far from practical, especially compared with the centroid tree algorithm. Perhaps a faster algorithm exists for approximation ratios smaller than two.

Open question 3.1. Is there an algorithm for $\mathrm{StOPT}(G, w)$ with approximation factor less than 2 that runs in time $\mathcal{O}(n \operatorname{polylog} n)$?

However, the most interesting question is whether the problem is polynomially tractable (if the underlying graph is a tree). Recall that the problem is polynomially tractable for paths, but NP-hard for graphs with tree-width 15 (theorem 3.6).

Open question 3.2. Can $\mathrm{StOPT}(G, w)$ be computed exactly in polynomial time when G is a tree?

Turning to more general graphs, in the bounded-tree-width case, we have a PTAS (theorem 3.5), and a generalization of our FPTAS to this setting seems possible to the author. Together with the NP-hardness result, this would settle the bounded tree-width case.

Conjecture 3.3. Theorem 3.8 can be generalized to graphs with bounded tree-width.

It could be interesting to consider other graph classes, in particular ones not based on *sparsity* like tree-width. For example, observe that the optimal static search tree problem is easy on cliques: the optimum is always the degenerate search tree that orders nodes by weights, with the heaviest nodes at the top.

Open question 3.4. Are there other natural graph classes that admit constant-factor approximations for the optimal static search tree problem?

Finally, let us consider the unweighted case. For general graphs, the problem is again NP-hard (theorem 3.9), and for trees, it is polynomially tractable via the pseudo-polynomial algorithm (theorem 3.7), though the running time is high (roughly $n^{4.42}$). Recall that the related problem of computing the tree-depth of a tree (i.e., minimizing the *worst-case* search time) is solvable in linear time [Sch89b]. Is the same true for the unweighted static search tree problem?

Open question 3.5. Can $\mathrm{StOPT}(G, \mathbb{1})$, i.e., the minimal trivially perfect completion, be computed in linear time if G is a tree?

3.1. Preliminaries

In this section, we first introduce some more technical definitions related to cost and weight functions, and then provide a few useful tools for the following chapters.

In our algorithms we always assume that the given weight function can be computed in $\mathcal{O}(1)$ time, and arithmetic operations involving weights and depths likewise can be evaluated in $\mathcal{O}(1)$ time. We frequently use the *sum* $w + w'$ of two weight functions w, w', which is the weight function w'' defined as $w''(v) = w(v) + w'(v)$, as well as the product $\alpha \cdot w$ of a real α and a weight function w, defined in the obvious way. For a subset $S \subseteq V(G)$, we write $w(S) = \sum_{v \in S} w(v)$, and for a rooted (sub)tree T, we write $w(T) = w(V(T))$.

$w + w'$

$\alpha \cdot w$

Sometimes it is convenient to use a weight function w with a larger domain (e.g., when G is a subgraph of a larger graph). In that case, we still write $\mathrm{cost}(T, w) = \mathrm{cost}(T, w')$, where w' is w restricted to $V(G)$.

We sometimes use the notation $\mathrm{cost}(T, w)$ when T is an arbitrary rooted tree and w is a weight function on $V(T)$. The definition generalizes in the obvious way.

The following observation is immediate from double counting and provides a useful alternative characterization of the cost of a search tree.

Observation 3.11. *For each rooted tree T and weight function w, we have*

$$\mathrm{cost}(T, w) = \sum_{v \in V(T)} w(T_v).$$

Lifting. We now define a certain search tree operation that will be useful in chapters 4 and 5. Let T be a search tree on G and let $v \in V(G)$. Let T^v the search tree on G that is obtained as follows. Set $\mathrm{root}(T^v) = v$, and for each component C in $G - v$, add $T|_C$ as a child subtree to v. We say that T^v is obtained from T by *lifting* the vertex v. This transformation can also be defined via *rotations* [BCI+20]. T^v is obtained by repeatedly rotating v with its parent until v becomes the root.

lifting

Observation 3.12. *Let T be a search tree on a tree G and let T^v be obtained by lifting v in T. Then, for each component C of $G - v$, and each $u \in V(C)$, we have*

$$\mathrm{Path}_{T^v}(u) = \{v\} \cup (\mathrm{Path}_T(u) \cap V(C)),$$

and, consequently,

$$\mathrm{depth}_{T^v}(u) \le \begin{cases} \mathrm{depth}_T(u), & \text{if } \mathrm{root}(T) \notin V(C) \\ \mathrm{depth}_T(u) + 1, & \text{if } \mathrm{root}(T) \in V(C). \end{cases}$$

We can also lift a *set* of vertices. Let T be a search tree on a graph G, and let $U \subseteq V(G)$. *Lifting* U in T means constructing a search tree T' where U induces an arbitrary prefix P, and for each component C of $G - U$, attach the tree $T|_C$ to the respective node in P. Equivalently, if π is a topological ordering of T, then T' is a the search tree with topological ordering π', where π' is obtained from π by moving all nodes of U to the start.

lifting (set)

Adding weight functions. We now show that $\mathrm{StOPT}(G, w)$ is superadditive and *approximately* subadditive in w. This will be useful when dealing with certain *error terms* of the form $\mathrm{StOPT}(G, w)$ that come up in chapter 9.

Lemma 3.13. *For each connected graph G and each pair of weight functions w, w' on G, we have*

$$\mathrm{StOPT}(G, w) + \mathrm{StOPT}(G, w') \le \mathrm{StOPT}(G, w + w') \le 2 \cdot (\mathrm{StOPT}(G, w) + \mathrm{StOPT}(G, w'))$$

Proof. For the first inequality (superadditivity), let T be an optimal search tree on $(G, w + w')$. Clearly, we have

$$\mathrm{cost}(T, w + w') = \mathrm{cost}(T, w) + \mathrm{cost}(T, w') \ge \mathrm{StOPT}(G, w) + \mathrm{StOPT}(G, w').$$

For the second inequality (approximate subadditivity), the idea is to construct a search tree T^* by choosing subtree roots in a top-down manner, alternating between T and T'. See figure 3.1 for an example.

Formally, let $\mathrm{Alt}(T, T', G)$ be the search tree with root $r = \mathrm{root}(T)$, and a child subtree $\mathrm{Alt}(T'|_C, T|_C, C)$ for each component C of $G - v$ (note the swapped T and T'). If G has only one vertex, then $\mathrm{Alt}(T, T', G)$ consists of that single node.

Let $T^* = \mathrm{Alt}(T, T', G)$. We show that

$$\mathrm{depth}_{T^*}(v) \le \min(2\,\mathrm{depth}_T(v) - 1, 2\,\mathrm{depth}_{T'}(v))$$

for each vertex $v \in V(G)$. If v is the root of T, then $\mathrm{depth}_{T^*}(v) = 1$ and we are done. Otherwise, let $r = \mathrm{root}(T^*) = \mathrm{root}(T)$ and let C be the component of $G - r$ that contains v. Let $S = T|_C$ and let $S' = T'|_C$. By lemma 2.18, we have $\mathrm{depth}_{S'}(v) \le \mathrm{depth}_{T'}(v)$, and, because r is an ancestor of v in T, but not in S, we have $\mathrm{depth}_S(v) \le \mathrm{depth}_T(v) - 1$.

Now let $S^* = \mathrm{Alt}(S', S, C)$. We have

$$\begin{aligned} \mathrm{depth}_{T^*}(v) &= 1 + \mathrm{depth}_{S^*}(v) \\ &\le 1 + \min(2\,\mathrm{depth}_{S'}(v) - 1, 2\,\mathrm{depth}_S(v)) && \text{by induction} \\ &\le \min(2\,\mathrm{depth}_{T'}(v), 2\,\mathrm{depth}_T(v) - 1) && \square \end{aligned}$$

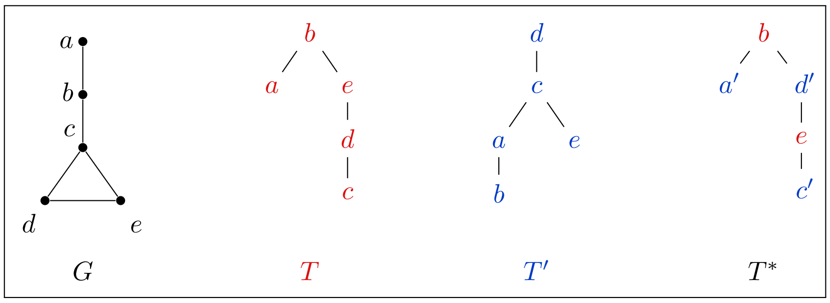


Figure 3.1.: Two search trees T and T' on a graph G, and the "merged" search tree T^* in the proof of lemma 3.13. In T^*, the blue color and the $'$ symbol indicate a node is chosen from T' instead of T.

3.2. Handling zero-weight vertices

In this section, we prove:

Theorem 3.10. *Let (G, w) be a given weighted tree with n vertices, and let m be the number of vertices with positive weight. In $\mathcal{O}(n)$ time, we can transform G into a tree G' with $V(G') \subseteq V(G)$ and $\mathrm{StOPT}(G, w) = \mathrm{StOPT}(G', w)$, such that $|V(G')| \leq 2m$ and G' has no zero-weight vertices of degree two or lower.*

We first need two technical lemmas.

Lemma 3.14. *Let (G, w) be a weighted tree, let v be a vertex in G with degree at most two and weight zero, and let u be some neighbor of v. For each search tree T on G with root v, there is a search tree S with root u such that $\mathrm{cost}(S, w) \leq \mathrm{cost}(T, w)$.*

Proof. We construct S as follows. Let π be a topological ordering of T, and let σ be obtained from π by moving u to the start and v to the end. Let S be the search tree with topological ordering σ.

We show that $\mathrm{depth}_S(x) \leq \mathrm{depth}_T(x)$ for each $x \in V(G) \setminus \{v\}$. Since $w(v) = 0$, this implies $\mathrm{cost}(S, w) \leq \mathrm{cost}(T, w)$, as desired.

We clearly have $\mathrm{depth}_S(u) = 1 \leq \mathrm{depth}_T(u)$. Now let $x \in V(G) \setminus \{v, u\}$. It is clear that x loses v as an ancestor and may gain u as an ancestor when transforming T to S. We claim that x gains no other ancestors, which implies $\mathrm{depth}_S(x) \leq \mathrm{depth}_T(x)$, as desired.

To prove our claim, we show that for each $y \in V(G) \setminus \{u\}$ with $y \prec_S x$, we also have $y \prec_T x$.

We have $y <_\sigma x$ by assumption, and thus also $y <_\pi x$. Suppose for the sake of contradiction that $y \not\prec_T x$. Then, by lemma 2.12, there is a set A that precedes y in π and separates x, y in G. If $v \notin A$, then A also precedes y in σ, so $y \not\prec_S x$, a contradiction. If $v \in A$, then observe that $B = A \setminus \{v\} \cup \{u\}$ precedes y in σ and separates x, y in G, since v has degree at most two and u is a neighbor of v. Thus, again $y \not\prec_S x$, a contradiction. □

Lemma 3.15. *Let (G, w) be a weighted tree, and let $U \subseteq V(G)$ be the set of vertices with degree larger than two, or positive weight (or both). Then U induces a prefix of some optimal search tree on (G, w).*

Proof. Suppose $U \neq \emptyset$; otherwise the statement is vacuous.

Consider the following recursive procedure on an optimal search tree T on (G, w). Let $r = \mathrm{root}(T)$. If $r \notin U$, then repeatedly apply lemma 3.14 until the root of the search tree is in U, and call the resulting search tree T'. If $r \in U$, then simply let $T' = T$.

Afterwards, recurse on the child subtrees of $\mathrm{root}(T')$. To see that we can indeed apply lemma 3.14 properly in recursive calls, observe that in every subgraph of (G, w), each vertex $v \notin U$ still has weight zero and degree at most two.

By induction, the final result has a prefix induced by U. □

Proof of theorem 3.10. We transform (G, w) by progressively removing zero-weight leaves, and replacing zero-weight degree-two vertices with an edge. (Equivalently, we progressively *contract* edges between zero-weight vertices of degree at most two and an arbitrary neighbor.) This can be done with a single traversal in $\mathcal{O}(n)$ time.

In the resulting tree G', each zero-weight vertex has degree at least three, implying that there are no more than $m - 2$ zero-weight vertices, and no more than $2m - 2$ vertices in total, as desired. Let $U = V(G')$ denote the remaining vertices and observe that $G' = \mathrm{torso}_G(U)$. We now show that $\mathrm{StOPT}(G', w) = \mathrm{StOPT}(G, w)$.

Let T be an optimal search tree on (G, w). By lemma 3.15, we can assume w.l.o.g. that U induces a prefix P of T. Clearly, we have $\mathrm{cost}(P, w) = \mathrm{cost}(T, w)$, since all vertices outside of U have weight zero. Since P is a search tree on G' by theorem 2.21, we have $\mathrm{StOPT}(G', w) \leq \mathrm{StOPT}(G, w)$.

Now let P be an optimal search tree on (G', w). By theorem 2.21, there is a search tree T on G with prefix P, implying $\mathrm{StOPT}(G, w) \leq \mathrm{StOPT}(G', w)$. □

4. Centroid trees

In this chapter, we study efficient approximation algorithms for the optimal static search tree problem based on centroid trees.

We start with giving the definitions and history of (weighted) centroids in some more detail. A vertex v is a *centroid* of a tree G if every component of $G - v$ has at most $\frac{1}{2}|V(G)|$ vertices. The fact that a centroid always exists was already shown in the 19th century by Jordan [Jor69]. The argument is constructive and goes as follows: Start at an arbitrary vertex of G and, as long as the current vertex is not a centroid, move one edge in the direction of the component with largest number of vertices. It is not hard to see that the procedure succeeds, visiting each vertex at most once. There is always either a unique centroid, or exactly two centroids, which must be adjacent [Kőn90].

centroid

Clearly, a search strategy that repeatedly selects the centroid of the remaining subgraph of G requires only $\lceil \log(n+1) \rceil$ queries (see section 1.1). A *centroid tree* T is the search tree corresponding to this strategy, where every node v is the centroid of $G[T_v]$. Centroid trees have a wide range of applications outside of searching [FJ83, GHL+87, GT98, BFCK06, Fer13, KPR+14, GHLW15, FV16].

centroid tree

The existence of centroid trees implies the following well-known result. Recall that the tree-depth $\operatorname{td}(G)$ is the minimum height of a search tree on G.

Theorem 4.1. *If G is a tree with n vertices, then* $\operatorname{td}(G) \leq \lceil \log(n+1) \rceil$.

We now discuss a weighted variant of centroids and centroid trees.

Weighted centroids. Let (G, w) be a weighted tree. A vertex $v \in V(G)$ is a *centroid* of (G, w) if every component C of $G - v$ satisfies $w(C) \leq \frac{1}{2} w(G)$. A *centroid tree* on (G, w) is constructed by choosing a weighted centroid as the root of every subtree.

weighted centroid

weighted centroid tree

To find a weighted centroid, we can use essentially the same procedure as for unweighted centroids, just replacing "number of vertices" with "weight". Hence, a weighted centroid tree always exists.

In contrast to the unweighted case, there can be more than two weighted centroids. For example, if all weights are zero, every vertex is a centroid. However, there can be at most two centroids with positive weight, and all other centroids must lie on a path between them (see section 4.2, theorem 4.12).

To the author's knowledge, weighted centroids have been first studied by Kariv and Hakimi [KH79], who proved equivalence with so-called *(1-)medians.* Results on computing medians include a simple linear-time algorithm [Gol71] (essentially the algorithm discussed above) and a dynamic forest algorithm that maintains medians with logarithmic update time [AHLT05]. Both algorithms will be useful to efficiently compute weighted centroid trees. We refer to Rosenthal and Pino [RP89] for applications and related concepts.

4.1. Overview

We now present the new results proved in this chapter. The proofs are deferred to the following sections.

Approximation ratio. We observed above that (even unweighted) centroids are not necessarily unique. This means there may be multiple centroid trees, even with different costs.[1] We denote by $\mathrm{Cent}(G, w)$ the *maximum* cost of a centroid tree of (G, w). Our first result is an upper bound on the approximation ratio of centroid trees.

$\boxed{\mathrm{Cent}(G, w)}$

Theorem 4.2. *Let (G, w) be a weighted tree. Then*

$$\mathrm{Cent}(G, w) \leq 2 \cdot \mathrm{StOPT}(G, w) - w(G).$$

We show that this result is optimal, including in the additive term. Moreover, the constant factor 2 cannot be improved even for unweighted instances.

Theorem 4.3. *For every $\varepsilon > 0$ there is a sequence of weighted trees (G_n, w_n) such that, for every centroid tree C^n of (G_n, w_n),*

$$\mathrm{cost}(C^n, w_n) \geq 2 \cdot \mathrm{StOPT}(G_n, w_n) - w_n(G_n) - \varepsilon,$$

and $\lim_{n\to\infty} \mathrm{StOPT}(G_n, w_n)/w_n(G_n) = \infty$.

Theorem 4.4. *There is a sequence of trees G_n, where for every unweighted centroid tree C^n of G_n,*

$$\mathrm{cost}(C^n, \mathbb{1}) \geq 2 \cdot \mathrm{StOPT}(G_n, \mathbb{1}) - 2|V(G_n)|.$$

and $\lim_{n\to\infty} \mathrm{StOPT}(G_n, w_n)/|V(G_n)| = \infty$.

Note that the fact that StOPT tends to infinity in theorems 4.3 and 4.4 establishes that the *asymptotic* approximation ratio is 2. By this we mean that every bound of the form $\mathrm{Cent}(G) \leq c \cdot \mathrm{StOPT}(G) + o(\mathrm{StOPT}(G))$ must have $c \geq 2$.

When the underlying tree G has maximum degree Δ, the approximation ratio can be improved to $2 - 2^{-\Delta}$; we refer to the paper this chapter is based on [BGKK23] for details.

All results on the approximation ratio are proved in section 4.3. We further show in section 4.4 that the cost of the centroid tree is bounded by the *Shannon entropy* of the weight function, extending a result for BSTs [Meh75]. This does not give us any good approximation guarantee, but will be useful in later chapters.

We make another related observation in section 4.5: As mentioned in the introduction, the *height* of a centroid tree can be far from optimal (i.e., far from the tree-depth).

[1]Consider, for instance, the two different centroid trees of a path on four vertices, with weights $(2, 3, 2, 3)$.

Computing centroid trees. The procedure mentioned above for finding centroids can be implemented in $\mathcal{O}(n)$ time, where $n = |V(G)|$, using some straight-forward bookkeeping. Using this to compute an unweighted centroid tree yields a running time of $\mathcal{O}(n \log n)$; this is because computing a node v in the centroid tree T is done in time $\mathcal{O}(|V(T_v)|)$, for a total running time of $\mathcal{O}(\operatorname{cost}(T, \mathbb{1})) \subseteq \mathcal{O}(n \log n)$ by observation 3.11.

This running time has been improved to $\mathcal{O}(n)$ by carefully using certain data structures [BFPÖ01, DGPV19]. These guarantees, however, do not readily generalize from the unweighted to the weighted setting. Intuitively, the difficulty lies in the fact that in the weighted case, the removal of a centroid vertex may split the tree in a very unbalanced way, leaving up to $n - 1$ vertices in one component. Thus, a naive recursive approach will take $\Theta(n^2)$ time in the worst case. Recall that a weighted centroid *BST* can be computed in $\mathcal{O}(n)$ time [Fre75, Meh77]; this is essentially because "unbalanced" centroids themselves can be found quickly in paths.

One solution, if a worse approximation ratio is acceptable, is to construct a search tree by alternating between choosing a weighted and an unweighted centroid tree. We call such a search tree a *semi-weighted centroid tree.* Computing it again only requires $\mathcal{O}(n \log n)$ time, since it has height $\mathcal{O}(\log n)$. In section 4.6, we precisely define semi-centroid trees and prove:

Theorem 4.5. *Let (G, w) be a weighted tree with n vertices. A semi-centroid tree T on G satisfies $\operatorname{cost}(T, w) \leq 4 \cdot \operatorname{StOPT}(G, w)$ and can be computed in $\mathcal{O}(n \log n)$ time.*

With the help of some heavy machinery we can also compute an actual weighted centroid tree in $\mathcal{O}(n \log n)$ time. The main step of our algorithm, finding the weighted centroid of a tree, is achievable in $\mathcal{O}(\log n)$ time, assuming that the underlying tree is stored in a *top tree* data structure [AHLT05]. Iterating this procedure in combination with known algorithms for *constructing* and *splitting* top trees yields the algorithm that runs in $\mathcal{O}(n \log n)$ time. Together with our result on the approximation ratio (theorem 4.2), this essentially implies theorem 3.1; the only caveat is that top trees only support positive weights.

In section 4.7.1, we develop an improved, *output-sensitive* algorithm, with running time $\mathcal{O}(n \log h)$, where h is the height of the resulting centroid tree.

Theorem 4.6. *Let (G, w) be a weighted tree with n vertices and only positive weights. We can compute a centroid tree of (G, w) in time $\mathcal{O}(n \log h)$, where h is the height of the computed centroid tree.*

It is not hard to see that the height of a centroid tree is at most $\log(W \cdot n)$ if all weights are between 1 and W, since each node v of depth k satisfies $w(v) \leq 2^{-k} \cdot (W \cdot n)$. The output-sensitive algorithm therefore has running time $\mathcal{O}(n \log \log(Wn))$ in this case, which is $\mathcal{O}(n \log \log n)$ if all weights are polynomial.

In section 4.7.2, we address the handling of zero-weight vertices, partially with the help of theorem 3.10, culminating in the main result of this chapter:

Theorem 4.7. *Let (G, w) be a given weighted tree with n vertices such that all weights are in $\{0\} \cup [1, W]$ for some $W \in \mathbb{R}_{\geq 0}$. Let m be the number of vertices with positive weight. We can compute a centroid tree on (G, w) in time $\mathcal{O}(n + m \log \log(m + W))$ or $\mathcal{O}(n + m \log m)$, whichever is lower.*

Together with the approximation ratio proved in theorem 4.2, this implies theorems 3.1 and 3.2.

Hardness. One may ask whether the weighted centroid tree can be computed in *linear* time, like the unweighted centroid tree [BFPÖ01, DGPV19], or the weighted centroid *BST* [Fre75]. In section 4.8 show that, assuming a general decision tree model of computation, this is not possible: The algorithm of theorem 4.6 is optimal for all n and h (up to a constant factor). Informally, our lower bound on the running time applies to any deterministic algorithm in which the input weights affect program flow only in the form of binary decisions. The model thus excludes using the weights for addressing memory, e.g., via hashing.

decision tree

More precisely, fix a tree G with n vertices. We say that a *binary decision tree* D_G *solves* G for a class of weight functions $\mathcal{W}$ mapping $V(G)$ to $\mathbb{R}_{\geq 0}$ if:

- The leaves of D_G are search trees on G.
- Every inner node of D_G is of the form "$f(w) = 1$?" for some function $f : \mathcal{W} \to \{0, 1\}$.
- For every weight function $w \in \mathcal{W}$, starting from the root of D_G and following branchings down the tree, we reach a leaf T of D_G that is a centroid tree of (G, w).

The height of D_G is then a lower bound on the worst-case running time.

Theorem 4.8. *Let $h \geq 3$ and $n \geq h + 1$ be integers. Then there exists a tree G with at most n vertices and a class $\mathcal{W}$ of weight functions on $V(G)$ such that for every $w \in \mathcal{W}$, every centroid tree of (G, w) has height at most h, and every binary decision tree that solves G for $\mathcal{W}$ has height $\Omega(n \log h)$.*

This implies that theorem 4.6 is indeed optimal for all h and n. We obtain a similar, but not quite tight lower bound for theorem 4.7.

Theorem 4.9. *Let $n \in \mathbb{N}$ and $W \in \mathbb{R}$ with $2 \leq W \leq 2^{n-2}$. Then there exists a tree G on at most n vertices and a class $\mathcal{W}$ of weight functions on $V(G)$ with codomain $[1, W]$, such that every binary decision tree that solves G for $\mathcal{W}$ has height $\Omega(n \log\log W)$.*

We now discuss for which range of the parameter W the bounds in theorems 4.7 and 4.9 are tight. First note that any reasonable model of computation will require $\Omega(n)$ time to read the input. Since we can pad the tree in the lower bound construction of Theorem 4.8 with zero-weight leaves, we can get a lower bound of $\Omega(n + m \log\log W)$ for all $m \leq n$, where m is a prescribed number of positive-weight vertices.

Recall that the upper bounds in theorem 4.7 are $\mathcal{O}(n + m \log\log(m + W))$ and $\mathcal{O}(n + m \log m)$. If $W \geq 2^m$, then the complexity of the problem is $\Theta(n + m \log m)$. If otherwise $\log\log W \in \Omega(\log\log m)$, i.e., if $W \geq 2^{\log^{\varepsilon} m}$ for some $\varepsilon > 0$, then the complexity is $\Theta(n + m \log\log W)$. There only is a gap between the upper and lower bounds when W grows very slowly; we leave the complexity in that case open.

Open question 4.1. What is the complexity of computing a centroid tree on a tree with n vertices, when all weights are in $[1, W]$ for $W \in \omega(1)$ and $W \leq 2^{\log^{o(1)} n}$?

Approximate centroid trees. Our final result is somewhat separate from the others. We show that optimal static search trees have a certain relaxed centroid tree property. This will be useful in chapter 6.

Let us call a vertex v of a tree G an *α-centroid*, for $0 \leq \alpha \leq 1$, if $w(H) \leq \alpha \cdot w(G)$ for each component H of $G - v$. An *α-centroid tree* is a search tree in which every vertex x is an α-centroid of its rooted subtree $G[V(T_x)]$.

α-centroid

α-centroid tree

Observe that the standard centroid tree is a $\frac{1}{2}$-centroid tree, and all search trees are 1-centroid trees. Also note that an α-centroid is a β-centroid for all $\beta \geq \alpha$ and that the existence of an α-centroid is not guaranteed for $\alpha < \frac{1}{2}$ (consider a single edge with the two endpoints having the same weight). In the paper on which this chapter is based, we show guarantees for the maximum cost of α-centroid trees [BGKK23]. Here, we focus only on the following result (proof in section 4.9).

Theorem 4.10. *Let T be an optimal search tree on a weighted tree (G, w). Then T is a $\frac{2}{3}$-centroid tree of (G, w).*

A special case of this result (for BSTs) was shown by Hirschberg, Larmore, and Molodowitch [HLM86], who also showed that the ratio $\frac{2}{3}$ is tight (in the special case of BSTs, and thus, also for STTs).

An important consequence is that the height of an optimal tree is limited by the *spread* of the weights.

Corollary 4.11. *Let T be an optimal search tree on a weighted tree (G, w), with weights in $[1, W]$. Then* $\operatorname{height}(T) \leq \log_{3/2}(n \cdot W)$.

In section 4.9, we show a generalization of corollary 4.11 that allows zero-weight vertices in a limited capacity.

Corollary 4.11 has an interesting algorithmic consequence: In chapter 6, we give an exact algorithm for the optimal static STT problem whose running time is polynomial in the size of the underlying tree, but *exponential* in the height of the output search tree. With the help of corollary 4.11, this algorithm can be transformed into a pseudo-polynomial algorithm (see section 6.2 for details).

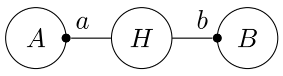
4.2. Some centroid facts

Before proceeding with the proofs of our main results, we show a few useful facts about centroids. First, we characterize the shapes that the *set of all centroids* of a weighted tree can take. Recall that in the unweighted case, there can be at most two centroids, which must be adjacent [Kőn90]. The following theorem is a strict generalization.

Theorem 4.12. *Let (G, w) be a weighted tree where at least one vertex has positive weight. The set of centroids of (G, w) induces a path, and every centroid that is not an endpoint of that path has weight zero.*

Proof. Without loss of generality, assume $w(G) = 1$. If there is only one centroid, we are done. Otherwise, assume a and b are two centroids of (G, w).

Let H be the unique connected component of $G - \{a, b\}$ that is adjacent to both a and b, or let H be the empty graph if no such component exists. Let A (resp. B) be the connected component of $G - H$ that contains a (resp. b). See below for an illustration.



Since a is a centroid, we have $w(B) + w(H) \leq \frac{1}{2}$, and likewise $w(A) + w(H) \leq \frac{1}{2}$, because b is a centroid. Further, we have $w(A) = 1 - w(B) - w(H) \geq \frac{1}{2}$ and similarly $w(B) = 1 - w(A) - w(H) \geq \frac{1}{2}$, so ultimately $w(A) = w(B) = \frac{1}{2}$ and $w(H) = 0$.

Now take some vertex v on the path between a and b. Clearly, each connected component $C \in \mathbb{C}(G - v)$ fully contains A, or fully contains B, or is disjoint from both. Thus, we have $w(C) \leq \frac{1}{2}$ and v is a centroid. Moreover, we have $w(v) = 0$ since $v \in V(H)$.

On the other hand, if $v \in V(H)$ is not on the path between a and b, then some component of $\mathbb{C}(G - v)$ contains both A and B, implying that v is not a centroid.

Taking a and b as the centroids with maximum distance concludes the proof. □

We proceed with showing that *contractions* essentially preserve centroids. We first need to properly define contractions in weighted trees.

Let (G, w) be a weighted tree, and let $\{u, v\}$ be an edge of G. Then *contracting* $\{u, v\}$ produces a weighted tree (G', w'), defined as follows. The tree G' is obtained from G by contracting $\{u, v\}$ into a new vertex s. The weight function w' is defined as $w'(s) = w(u) + w(v)$ and $w'(x) = w(x)$ for each $x \in V(G') \setminus \{s\}$.

Lemma 4.13. *Let (G, w) be a weighted tree, and let (G', w') be obtained from (G, w) by contracting the edge $\{u, v\}$ into a new vertex s. Then s is a centroid of (G', w') if and only if u or v is a centroid of (G, w). Further, a vertex $x \in V(G') \setminus \{s\}$ is a centroid of (G', w') if and only if x is a centroid of (G, w).*

Proof. Assume w.l.o.g. that $w(G) = 1$. Consider first a vertex $x \in V(G) \setminus \{u, v\}$. The contraction $\{u, v\}$ does not change the weight of any component $C \in \mathbb{C}(G - x)$. Thus, x is a centroid of (G, w) if and only if x is a centroid of (G', w').

Consider now the vertices u, v, and s. Observe that $\mathbb{C}(G - \{u, v\}) = \mathbb{C}(G' - s)$, and the contraction does not affect the weights of these components. If u or v is a centroid of (G, w), then no component of $\mathbb{C}(G - \{u, v\})$ has weight more than $\frac{1}{2}$. Thus, s is a centroid of (G', w').

On the other hand, suppose that s is a centroid of (G', w'). Consider a component $C \in \mathbb{C}(G - u)$. If C does not contain v, then $C \in \mathbb{C}(G - \{u, v\}) = \mathbb{C}(G' - s)$, so $w(C) \leq \frac{1}{2}$ by assumption. In the same way, for each $C \in \mathbb{C}(G - v)$, if C does not contain u, then $w(C) \leq \frac{1}{2}$. Now let C_v be the component of $G - u$ that contains v, and let C_u be the component of $G - v$ that contains u. Since u and v are adjacent, we know that C_v and C_u are disjoint. Thus, we have $w(C_v) + w(C_u) \leq 1$, so either $w(C_v) \leq \frac{1}{2}$ or $w(C_u) \leq \frac{1}{2}$. Therefore, one of u and v is a centroid of (G', w'). □

4.3. Approximation ratio

In this section, we prove our upper and lower bounds on the approximation quality of centroid trees.

4.3.1. Upper bound

Theorem 4.2. *Let (G, w) be a weighted tree. Then*

$$\mathrm{Cent}(G, w) \leq 2 \cdot \mathrm{StOPT}(G, w) - w(G).$$

We need the following lemma. (See section 3.1 for the definition of *lifting*.)

Lemma 4.14. *Let (G, w) be a weighted tree and c be a centroid of (G, w). Let T be a search tree on G, and let T^c be obtained by lifting c in T. Then,*

$$\text{cost}(T^c, w) \leq \text{cost}(T, w) + \tfrac{1}{2}(w(G) - w(c)).$$

Proof. Let r be the root of T. If $c = r$, we have $T^c = T$, so the lemma is true (observe that $w(c) \leq w(G)$). Otherwise, by observation 3.12, the depth of a vertex v can only increase due to the lifting if v is in the same component C of $G - c$ as r. Moreover, the depth of c clearly decreases by at least one.

Since $w(C) \leq \frac{1}{2}w(G)$ because c is a centroid, we have

$$\text{cost}(T^c, w) \leq \text{cost}(T, w) + \tfrac{1}{2}w(G) - w(c) \leq \text{cost}(T, w) + \tfrac{1}{2}(w(G) - w(c)). \qquad \square$$

Lemma 4.15. *Let (G, w) be a weighted tree and c be a centroid of (G, w). Then,*

$$\sum_{C \in G-c} \text{StOPT}(C, w) \leq \text{StOPT}(G, w) - \tfrac{1}{2}(w(G) + w(c)).$$

Proof. Let T be an optimal search tree on (G, w), and let T^c be obtained by lifting c in T. Clearly, the left-hand side of the statement is at most $\text{cost}(T^c, w) - w(G)$, so applying lemma 4.14 yields the claim. $\square$

Proof of theorem 4.2. The proof is by induction on the number of vertices. When $|V(G)| = 1$ we have

$$2 \cdot \text{StOPT}(G, w) - w(G) = 2w(G) - w(G) = w(G) = \text{Cent}(G, w),$$

as required.

Assume $|V(G)| > 1$. Let T be a centroid tree on G and $c = \text{root}(T)$. We have

$$\begin{aligned} \text{cost}(T, w) &= w(G) + \sum_{C \in \mathbb{C}(G-c)} \text{Cent}(C, w) \\ &\leq w(G) + \sum_{C \in \mathbb{C}(G-c)} (2 \cdot \text{StOPT}(C, w) - w(C)) && \text{by induction} \\ &= w(c) + 2 \cdot \sum_{C \in \mathbb{C}(G-c)} \text{StOPT}(C, w), \end{aligned}$$

therefore it is enough to show that

$$w(c) + 2 \cdot \sum_{C \in \mathbb{C}(G-c)} \text{StOPT}(C, w) \ \leq \ 2 \cdot \text{StOPT}(G, w) - w(G),$$

which easily follows from lemma 4.15. This concludes the proof. $\square$

Note that in the edge-query model, a 2-approximation can be shown using similar techniques [CJLM10], but that result is not best possible [CJLM14].

4.3.2. Lower bounds

In this section, we show that theorem 4.2 is tight, even for unweighted trees.

Weighted lower bound. We start with:

Theorem 4.16. *There is a sequence of weighted trees (G_n, w_n) such that*

$$\mathrm{Cent}(G_n, w_n) \geq 2 \cdot \mathrm{StOPT}(G_n, w_n) - w_n(G_n).$$

and $\lim_{n\to\infty} \mathrm{StOPT}(G_n, w_n)/w_n(G_n) = \infty$.

Observe that this is slightly different than the promised theorem 4.3: In theorem 4.16, we bound $\mathrm{Cent}(G, w)$, which is the *maximum* cost of a centroid tree, so the bound does not necessarily hold for all centroid trees. At the end of this section, we will introduce a tie-breaking procedure that makes the centroid tree unique, a the cost of a small error term, which implies theorem 4.3.

We now construct the sequence (G_n, w_n). For the sake of the construction, we view G_n as a rooted tree. The base case G_0 is a tree with a single vertex v and $w_0(v) = 1$. For $n > 0$, take two copies (A, w_A) and (B, w_B) of (G_{n-1}, w_{n-1}). Connect the roots of A and B to a new vertex c. Finally, set $\mathrm{root}(G_n) = \mathrm{root}(A)$. See figure 4.1 (a). We define w_n as follows.

$$w_n(v) = \begin{cases} 0, & v = c \\ \frac{1}{2} w_A(v), & v \in V(A) \\ \frac{1}{2} w_B(v), & v \in V(B). \end{cases}$$

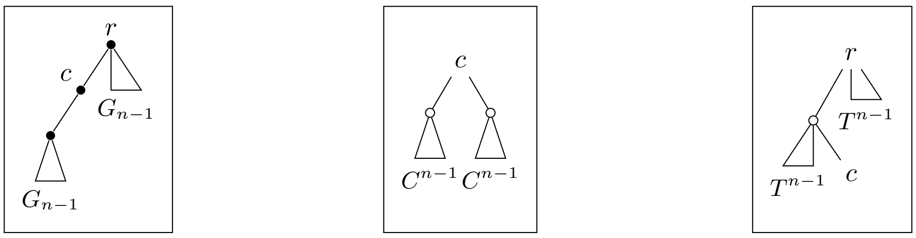
Let C^n denote the search tree on G_n obtained by setting c as the root and recursing; see figure 4.1 (b). Observe that C^n is a centroid tree of (G_n, w_n).

Lemma 4.17. *The following statements hold:*

(i) $\mathrm{cost}(C^n, w_n) = n + 1$.

(ii) $w_n(G_n) = 1$.

(iii) $\lim_{n\to\infty} \mathrm{StOPT}(G_n, w_n)/w_n(G_n) = \infty$.



(a) Underlying tree G_n (b) Centroid tree C^n (c) Better search tree T^n

Figure 4.1.: Illustration of the proof of Theorem 4.16. The vertex r is the root of G_n.

Proof. For part (i), let $c_n = \operatorname{cost}(C^n, w_n)$. Clearly, $c_0 = 1$. Assume $n > 0$. Let C_A and C_B be search trees on A and B respectively, each a copy of C_{n-1}. By construction of C^n we have

$$c_n = 1 + \tfrac{1}{2}\operatorname{cost}(C_A, w_A) + \tfrac{1}{2}\operatorname{cost}(C_B, w_B) = 1 + c_{n-1},$$

and (i) follows by induction.

Part (ii) is easily shown by induction. Finally, using (i), (ii) and theorem 4.2, we have $\operatorname{StOPT}(G_n, w_n) \geq \frac{n}{2} + 1$, which implies part (iii). □

Next, in order to bound $\operatorname{StOPT}(G_n, w_n)$ from above, we construct a sequence of search trees T^n on G_n. See figure 4.1 (c). The search tree T_0 is a single vertex. Now assume $n > 0$. Let A, B, and c be defined as in the definition of G_n. Let T_A and T_B be search trees on A and B, respectively, each a copy of T^{n-1}. Denote $r_A = \operatorname{root}(A)$ and $r_B = \operatorname{root}(B)$. The search tree T^n is obtained by setting $\operatorname{root}(T^n) = r_A$, making r_B a child of r_A, and making c a child of r_B.

Lemma 4.18. $\operatorname{cost}(T^n, w_n) = \frac{n}{2} + 1$.

Proof. Denote $t_n = \operatorname{cost}(T^n, w_n)$. Clearly $t_0 = 1$. Assume $n > 0$. The contribution of vertices of A to t_n is exactly $\frac{1}{2}\operatorname{cost}(T_A, w_A) = \frac{1}{2}t_{n-1}$. Since r is an ancestor of all vertices in B, the contribution of these vertices to t_n is exactly $\frac{1}{2}(1 + \operatorname{cost}(T_B, w_B)) = \frac{1}{2}(1 + t_{n-1})$. Summing the contribution of all vertices, we get $t_n = t_{n-1} + \frac{1}{2}$ and the claim follows. □

Lemmas 4.17 and 4.18 together imply theorem 4.16.

Unweighted lower bound. We now show:

Theorem 4.4. *There is a sequence of trees G_n, where for every unweighted centroid tree C^n of G_n,*

$$\operatorname{cost}(C^n, \mathbb{1}) \geq 2 \cdot \operatorname{StOPT}(G_n, \mathbb{1}) - 2|V(G_n)|.$$

and $\lim_{n\to\infty} \operatorname{StOPT}(G_n, w_n)/|V(G_n)| = \infty$.

It turns out that the construction above also works in the unweighted case. Let G_n, c, A, B, C^n, and T^n be as defined above. Observe that c is also the *unique unweighted* centroid of G_n. Indeed, observe $|V(A)| = |V(B)| = \frac{1}{2}(|V(G)| - 1)$. For any vertex $v \neq c$, the subgraph $G - v$ contains one component of that contains either $V(A) \cup \{v\}$ or $V(B) \cup \{v\}$, and thus has weight at least $\frac{1}{2}|V(G)| + \frac{1}{2}$. This means that v cannot be a centroid.

Consequentially C^n is the unique centroid tree of G_n, and no tie-breaking is necessary. The lemma below implies theorem 4.4.

Lemma 4.19. *The following statements hold:*

(i) $|V(G_n)| = 2^{n+1} - 1$.

(ii) $\operatorname{cost}(C^n, \mathbb{1}) = 2^{n+1}n + 1$.

(iii) $\operatorname{cost}(T^n, \mathbb{1}) = 2^n \cdot n + 2^{n+1} - 1$.

(iv) $\lim_{n\to\infty} \operatorname{StOPT}(G_n, \mathbb{1})/|V(G_n)| = \infty$.

Proof. Part (i) follows directly by induction.

Denote $c_n = \mathrm{cost}(C^n, \mathbb{1})$ and $t_n = \mathrm{cost}(T^n, \mathbb{1})$. Observe that c_n satisfies the recurrence

$$\begin{aligned} c_0 &= 1 \\ c_n &= |V(G_n)| + 2c_{n-1} = 2^{n+1} - 1 + 2c_{n-1}, \end{aligned}$$

and that $c_n = 2^{n+1}n + 1$ is the solution for this formula. Repeating the analysis from lemma 4.18, we get the recurrence

$$\begin{aligned} t_0 &= 1 \\ t_n &= t_{n-1} + (|V(G_{n-1})| + t_{n-1}) + 2 = 2^n + 1 + 2t_{n-1}. \end{aligned}$$

Observe that $t_n = 2^n \cdot n + 2^{n+1} - 1$ is the solution. Finally, part (iv) follows directly from (i), (ii), and theorem 4.2. □

Breaking ties. We now show how to derive theorem 4.3 from theorem 4.16, as outlined above, using the following lemma.

Lemma 4.20. *Let G be a tree, $w : V(G) \to \mathbb{R}_{\geq 0}$ a weight function and let S be a centroid tree of (G, w). For every $\varepsilon > 0$ there exists a weight function $w' : V(G) \to \mathbb{R}_{\geq 0}$ such that S is the unique centroid tree of (G, w') and $w(v) \leq w'(v) < w(v) + \varepsilon$ for all $v \in V(G)$.*

Before proving lemma 4.20, we show how to apply it. Let (G_n, w_n) be the sequence of trees from theorem 4.16. For each n, obtain a weight function w'_n using lemma 4.20 with the centroid tree $S = C^n$ defined above, and some ε to be determined later. Let T^n be an optimal search tree on the original sequence (G_n, w_n). By theorem 4.16, we have

$$\mathrm{Cent}(G_n, w_n) \geq 2 \cdot \mathrm{StOPT}(G_n, w_n) - w_n(G_n).$$

Clearly, we have $\mathrm{cost}(C^n, w'_n) = \mathrm{Cent}(G_n, w'_n) \geq \mathrm{Cent}(G_n, w_n)$. Moreover, we have $w'_n(v) \leq w_n(v) + \varepsilon$ for all $v \in V(G_n)$, so

$$\begin{aligned} &\mathrm{StOPT}(G_n, w'_n) \leq \mathrm{cost}(T^n, w'_n) \leq \mathrm{cost}(T^n, w_n + \varepsilon \cdot \mathbb{1}) \\ &= \mathrm{StOPT}(G_n, w_n) + \varepsilon \cdot \mathrm{cost}(T^n, \mathbb{1}) \leq \mathrm{StOPT}(G_n, w_n) + \varepsilon |V(G_n)|^2. \end{aligned}$$

The last inequality follows from the fact that $\mathrm{cost}(T, \mathbb{1}) \leq |V(T)|^2$ for every rooted tree. Overall we have $\mathrm{cost}(C^n, w'_n) \geq 2\,\mathrm{StOPT}(G_n, w'_n) - 2\varepsilon |V(G_n)|^2 - w_n(G_n)$. Since we can choose ε arbitrarily, even depending on n, this implies theorem 4.3

We now prove lemma 4.20 with a series of simple observations. If (G, w) is a weighted tree and $v \in V(G)$, let

$$M(v, w) = \max_{C \in \mathbb{C}(G-v)} w(C).$$

The following is a well-known alternative characterization of centroids (also called *median* in this context).

Lemma 4.21 (Kariv and Hakimi [KH79])**.** *Let (G, w) be a weighted tree. Each $v \in V(G)$ is a centroid of (G, w) if and only if it minimizes $M(v, w)$.*

Lemma 4.22. *Let w, w' be two weight functions on a tree G. Assume that a vertex $c \in V(G)$ is a centroid of (G, w) and the unique centroid of (G, w'). Then c is the unique centroid of $(G, w + w')$.*

Proof. For each $v \in V(G) \setminus \{c\}$, we have have $M(c, w) \leq M(v, w)$ and $M(c, w) < M(v, w)$, implying $M(c, w + w') < M(v, w + w')$, so c is the unique centroid of $(G, w + w')$. ☐

Lemma 4.23. *Let T be a search tree on G. There exists a weight function $w : V(G) \to \mathbb{R}_{\geq 0}$ such that T is the unique centroid tree of (G, w).*

Proof. We proceed by induction on n. When $n = 1$, every w has the desired property. Assume $n > 1$. Let $r = \mathrm{root}(T)$ and let $T^1, \dots, T^k$ be the child subtrees of r in T. Let $G_i = G[V(T^i)]$ for $i \in [k]$. By the induction hypothesis, there are weight functions w_i for $i \in [k]$ such that T^i is the unique centroid tree of (G_i, w_i). Let w be defined by

$$w(u) = \begin{cases} w_i(u), & u \in V(G_i) \\ 1 + \sum_{i=1}^{k} w_i(G_i), & u = r. \end{cases}$$

Observe that $w(r) > \frac{1}{2} w(G)$. This means that no other vertex can be a centroid of G, so r is the unique centroid of (G, w). Clearly, the components of $G - r$ are $G_1, G_2, \dots, G_k$, and restricting w to G_i yields w_i, so T is the unique centroid tree on (G, w). ☐

Proof of lemma 4.20. Using lemma 4.23, let $\tilde{w} : V(G) \to \mathbb{R}_{\geq 0}$ be such that S is the unique centroid tree of $(G, \tilde{w})$. We can scale $\tilde{w}$ so that $\tilde{w}(v) < \varepsilon$ for all $v \in V(G)$. Let $w' = w + \tilde{w}$. Using lemma 4.22, by induction on the height of T, it follows that T is the unique centroid tree of (G, w'). ☐

4.4. Centroid trees and entropy

In this section, we discuss the relation between the static optimum and the *Shannon entropy* of the weight function.

First, we show that the *entropy bound* for binary search trees holds for optimal static search trees on arbitrary trees (though not arbitrary graphs). We do this by bounding the cost of centroid trees. Let $p \colon X \to [0, 1]$ be a probability distribution. The *Shannon entropy* of p, written $H(p)$, is defined as

Shannon entropy

$$H(p) = \sum_{\substack{x \in X \\ p(x) > 0}} p(x) \cdot \log \tfrac{1}{p(x)}.$$

Proposition 4.24. *Let T be a centroid tree on a weighted tree (G, p), where p is a probability distribution on $V(G)$. Then*

$$\mathrm{cost}(T, p) \leq 1 + H(p)$$

Proof. Let U be the set of vertices with positive weight/probability, let $u \in U$, and let $d = \mathrm{depth}_T(u)$. By definition, we have $p(u) \leq 2^{1-d} \cdot p(G) = 2^{1-d}$. Thus, $d \leq 1 - \log p(u)$.

Overall, we have

$$\begin{aligned}\operatorname{cost}(T,p) &\le \sum_{v\in V(G)} p(v)\cdot \operatorname{depth}_T(v) = \sum_{u\in U} p(u)\cdot \operatorname{depth}_T(u)\\ &\le \sum_{u\in U} p(u)\cdot(1-\log p(u)) = 1+H(p).\end{aligned}$$ □

Second, we consider a recently proven related lower bound. Recall that $\mathrm{StOPT}(G,p)$ can be constant, e.g., when G is a star. On the other hand, if G is a path, then $H(p)$ is an asymptotic lower bound of $\mathrm{StOPT}(G,p)$ [Knu71]. Voderholzer proved the following interpolation between these two extremes.

Theorem 4.25 (Voderholzer [Vod23, §3.2])**.** *Let G be a tree with maximum degree Δ, and let p be a probability distribution on $V(G)$. Then*

$$\mathrm{StOPT}(G,p) \ge \frac{H(p)}{\log(\Delta+1)}.$$

4.5. Height of centroid trees.

We now show that centroid trees do *not* approximate optimal-height search trees, even in the unweighted case.

Proposition 4.26. *There is a sequence of graphs G_n, such that the height of each centroid tree of G_n is exponential in $\mathrm{td}(G_n)$, and $\lim_{n\to\infty} \mathrm{td}(G_n) = \infty$.*

Proof. Let the graph G_n be defined as follows: Start with a path on vertices $v_1, v_2, \dots, v_n$, in that order. For each $i \in [n]$, attach $3^{i-1}-1$ leaves to v_i.

Observe that $|V(G_n)| = \sum_{i=0}^{n-1} 3^n = \frac{1}{2}(3^n-1)$. Further observe that v_n is the unique centroid of G_n. Indeed, the subgraph consisting of v_n and its attached leaves contains $3^{n-1} > \frac{1}{2}|V(G_n)|$ vertices, so no centroid can be outside it. Clearly, none of the leaves is a centroid.

Removing v_n from G_n yields some isolated vertices and a copy of G_{n-1}. By induction, the centroid tree on G_n thus has height n.

On the other hand, the tree-depth of G_n is at most $\lceil\log(n+1)\rceil + 1$: The path $v_1, v_2, \dots, v_n$ has tree-depth $\lceil\log(n+1)\rceil$, and the attached leaves increase it by at most one. Thus, the height of the centroid tree is exponential in the tree-depth. □

4.6. Semi-weighted centroid trees

semi-weighted centroid tree

Let (G,w) be a weighted tree. A search tree T is a *semi-weighted centroid tree* on (G,w) if, for each $v \in V(T)$, we have:

(a) If $\operatorname{depth}_T(v)$ is even, then v is an (unweighted) centroid of $G[T_v]$.

(b) If $\operatorname{depth}_T(v)$ is odd, then v is a (weighted) centroid of $(G[T_v], w)$.

Clearly, a semi-weighted centroid tree can be constructed in a top-down manner for each weighted tree.

Lemma 4.27. *Let T be a semi-weighted centroid tree on a weighted graph (G, w) with n vertices. Then* $\operatorname{height}(T) \leq 2 \log n + 1$.

Proof. We proceed by induction. Let h_n denote the maximum height of a semi-weighted centroid tree on (G, w). Observe that $h_1 = 1 = 2 \cdot \log 1 + 1$, solving the case $n = 1$.

Now suppose $n > 1$, let T be a semi-weighted centroid tree on (G, w), and let $r = \operatorname{root}(T)$. Since r is an unweighted centroid, we have $|V(T_c)| \leq \frac{1}{2}n$ for every child c of r, and thus, we also have $|V(T_g)| \leq \frac{1}{2}n$ for every grandchild g of r. If Γ is the set of grandchildren of r, we have

$$\begin{aligned}\operatorname{height}(T) &\leq 2 + \max_{g \in \Gamma} \operatorname{height}(T_g) \\ &\leq 2 + h_{\lfloor n/2 \rfloor} \leq 2 + 2 \log \lfloor \tfrac{n}{2} \rfloor + 1 \leq 2 \log n + 1. \qquad \square\end{aligned}$$

Recall that we can compute a centroid or weighted centroid in linear time [Gol71, KH79]. Computing a semi-centroid tree T thus takes $\mathcal{O}(\sum_{v \in V(T)} |V(T_v)|) = \mathcal{O}(\operatorname{cost}(T, \mathbb{1})) \subseteq \mathcal{O}(n \log n)$ time by lemma 4.27.

We next prove the approximation ratio, starting with a simple technical lemma.

Lemma 4.28. *Let (G, w) be a weighted graph, and let $v \in V(G)$. Then*

$$\sum_{C \in \mathbb{C}(G-v)} \operatorname{StOPT}(C, w) \leq \operatorname{StOPT}(G, w).$$

Proof. Let T be an optimal search tree on (G, w). We have

$$\sum_{C \in \mathbb{C}(G-v)} \operatorname{StOPT}(C, w) \leq \sum_{C \in \mathbb{C}(G-v)} \operatorname{cost}(T|_C, w) \leq \operatorname{cost}(T, w) = \operatorname{StOPT}(G, w).$$

The second inequality follows from the fact that $\operatorname{depth}_{T|_C}(v) \leq \operatorname{depth}_T(v)$ for all $v \in V(C)$, by lemma 2.18. $\square$

Lemma 4.29. *Let T be a semi-weighted centroid tree on a weighted tree (G, w). Then,*

$$\operatorname{cost}(T, w) \leq 4 \cdot \operatorname{StOPT}(G, w).$$

Proof. We proceed by induction on $\operatorname{height}(T)$. First, assume that $\operatorname{height}(T) \leq 2$. Then, we have $\operatorname{cost}(T, w) \leq 2 \cdot w(G) \leq 2 \cdot \operatorname{StOPT}(G, w)$.

Now suppose $\operatorname{height}(T) > 2$. Let $r = \operatorname{root}(T)$ and let K be the set of children of r. First observe that for $c \in K$, if Γ_c is the set of children of c, then

$$\begin{aligned}\operatorname{cost}(T_c, w) &= w(T_c) + \sum_{g \in \Gamma_c} \operatorname{cost}(T_g, w) \\ &\leq w(T_c) + \sum_{g \in \Gamma_c} 4 \operatorname{StOPT}(G[T_g], w) && \text{by induction} \\ &\leq w(T_c) + 4 \operatorname{StOPT}(G[T_c], w). && \text{by lemma 4.28}\end{aligned}$$

With this, we have

$$\begin{aligned}\operatorname{cost}(T,w) &\le w(T_v) + \sum_{c\in K} \operatorname{cost}(T_c, w)\\ &\le w(T_v) + \sum_{c\in K} w(T_c) + 4\,\mathrm{StOPT}(G[T_c], w)\\ &\le 2\cdot w(G) + \sum_{C\in\mathbb{C}(G-r)} 4\cdot \mathrm{StOPT}(C,w)\\ &\le 4\cdot \mathrm{StOPT}(G,w) && \text{by lemma 4.15} \quad \square\end{aligned}$$

Lemmas 4.27 and 4.29 together imply

Theorem 4.5. *Let (G,w) be a weighted tree with n vertices. A semi-centroid tree T on G satisfies* $\operatorname{cost}(T,w) \le 4\cdot \mathrm{StOPT}(G,w)$ *and can be computed in $\mathcal{O}(n\log n)$ time.*

4.7. Computing centroid trees

In this section, we show how to compute weighted centroid trees using the *top tree* framework of Alstrup, Holm, de Lichtenberg, and Thorup [AHLT05]. *Top trees* are a data structure used to maintain dynamic forests under insertion and deletion of edges (see also chapter 10). They expose a simple interface that allows the user to maintain information in the trees of the forest. For this, the user only needs to implement a small number of internal operations.

Alstrup et al. in particular show how to maintain the *median* of trees in $\mathcal{O}(\log n)$ per operation. As mentioned above, median and centroid are equivalent [KH79].

Theorem 4.30 ([AHLT05, Theorem 3.6])**.** *We can maintain a forest with positive vertex weights on n vertices under the following operations:*

- *Add an edge between two given vertices u, v that are not in the same tree;*
- *Remove an existing edge;*
- *Change the weight of a vertex;*
- *Retrieve a pointer to the tree containing a given vertex;*
- *Find the centroid of a given tree in the forest.*

Each operation requires $\mathcal{O}(\log n)$ time. A forest without edges and with n arbitrarily weighted vertices can be initialized in $\mathcal{O}(n)$ time.

Note that theorem 4.30 only admits *positive* vertex weights, whereas we usually allow zero-weight vertices. We show how to handle this problem in section 4.7.2.

We now show how to use theorem 4.30 to construct a centroid tree in $\mathcal{O}(n\log n)$ time.

Theorem 4.31. *Given a tree G on n vertices and a positive weight function w, we can compute a centroid tree of (G,w) in $\mathcal{O}(n\log n)$ time.*

Proof. First build a top tree on G by adding the edges one-by-one, in $\mathcal{O}(n\log n)$ time. Find the centroid c, and remove each incident edge. Then, recurse on each newly created tree (except for the one containing only c). The algorithm finds each vertex precisely once and removes each edge precisely once, for a total running time of $\mathcal{O}(n\log n)$. $\square$

4.7.1. Output-sensitive algorithm

We now improve the algorithm given above to run in time $\mathcal{O}(n \log h)$, where n is the number of vertices in G and h is the height of the computed centroid tree.

The main idea of the algorithm is inspired by the linear-time algorithm for *unweighted* centroids by Della Giustina, Prezza, and Venturini [DGPV19]. Instead of building a top tree on the whole tree G, we first partition G into connected subgraphs of size roughly h, and build a top tree on each component. Contracting each component into a single vertex yields *super-vertices* in a *super-tree*. Each search for a centroid consists of a global search and a local search: We first find the super-vertex containing the centroid, then we find the centroid within that super-vertex. After finding the centroid, we remove it, which may split up the super-vertex into multiple super-vertices with a top tree each, and also may split the super-tree into a super-forest. Finally, we recurse on each component of the super-forest.

It can be seen that the total number of top tree operations needed is $\mathcal{O}(n)$. Since the top trees each contain only h vertices, a top tree operation takes $\mathcal{O}(\log h)$ time, for a total of $\mathcal{O}(n \log h)$. We now proceed with a more detailed description of the algorithm.

Ternarization. If the degree of a vertex of G is unbounded, then a partition into similarly sized connected subgraphs may not be possible. To fix this, we *ternarize* G by replacing each vertex v of degree $\deg_G(v) > 3$ by a path P_v of $\deg_G(v) - 2$ vertices with degree three. Call the new vertices *virtual* and let G' denote the resulting tree. Each virtual vertex of P_v is adjacent in G' to an (arbitrary) neighbor of v in G except for the two endpoints of P_v, which are adjacent to two neighbors each. See figure 4.2 for an example.

We maintain a link between each vertex in G and every associated virtual vertex in G' (if any). Observe that $|V(G')| \leq 2n$.

Let w' be a weight function on G' obtained from w by arbitrarily distributing weight from each deleted vertex to its associated virtual vertices. Note that (G, w) can be obtained from (G', w') by contracting every path P_v of associated virtual vertices. Thus, by lemma 4.13, a vertex of G is a centroid of (G, w) if and only if one of the associated virtual vertices is a centroid of (G', w).

Partition. Fix a parameter k. We now compute a partition into $\mathcal{O}(\frac{n}{k})$ connected subgraphs of size at most $3k$ as follows. Arbitrarily root G'. Iteratively remove minimal rooted subtrees of size at least k, using a simple linear-time bottom-up traversal. Since each node has at most three children, this produces connected subgraphs of size between k and $3k - 2$. The only exception are the nodes remaining at the end, which we put into a possibly smaller subgraph. The total number of subgraphs is at most $\frac{1}{k}|V(G')| + 1 = \frac{2n}{k} + 1$.

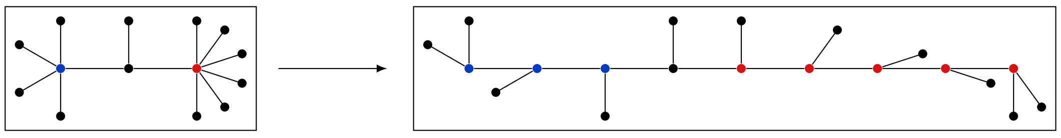


Figure 4.2.: Ternarization of a tree.

Building the super-tree. By contracting each connected subgraph of the partition into a single vertex, we obtain a weighted tree $(\mathcal{S}, W)$, the *super-tree.* We call each vertex $A \in V(\mathcal{S})$ a *super-vertex*, and write $G'[A]$ for the subgraph of G' contracted into A. We write $V(A) = V(G'[A])$ for short. By definition of W via the contraction, we have $W(A) = w'(V(A))$.

Note that each *super-edge* in $E(\mathcal{S})$ is associated with precisely one normal edge in G'; we maintain an explicit link between the super-edge and the normal edge. For each super-vertex A, we build a top tree on $G'[A]$, and store the weight $W(A)$. We call a vertex $v \in V(A)$ that is adjacent to some vertex $u \in V(G') \setminus V(A)$ an *inner boundary vertex* of A, and maintain a list of inner boundary vertices for each super-vertex.

inner boundary

Constructing $(\mathcal{S}, W)$ can be done in linear time. Setting up the top trees requires $\mathcal{O}(\frac{n}{k} \cdot k \log k) = \mathcal{O}(n \log k)$ time.

Main procedure and recursion. Below, we describe how to find a centroid and remove it, along with associated virtual vertices. Doing so may split up $\mathcal{S}$ (and implicitly G and G') into multiple connected components, on which we recurse. Hence, a recursive step operates on a tree G_r (a subgraph of G), a ternarization G'_r of G_r, and a super-tree $\mathcal{S}_\mathrm{r}$ of G'_r. Note that only $\mathcal{S}_\mathrm{r}$ is explicitly given, whereas G_r and G'_r are implicit in the super-tree data structure. Our task is to find a centroid c of G_r and remove it from G_r, i.e., for each component H of $G_\mathrm{r} - c$, we return a super-tree on a ternarization of H.

Finding centroids. We now describe how to find a centroid of G'_r with help of the super-tree $\mathcal{S}_\mathrm{r}$. Note that, by lemma 4.13, this is enough to find a centroid of G_r.

First, we find the centroid A^* of $\mathcal{S}_\mathrm{r}$ with the simple linear-time algorithm [Gol71, KH79]. By lemma 4.13, a centroid of G'_r must be contained in $G'_\mathrm{r}[A^*]$.

We now construct a suitable weight function w^* on $V(A^*)$ so that we can find a centroid of G'_r within $G'_\mathrm{r}[A^*]$. For each $v \in V(A^*)$, let C_v be the connected component of $G'_\mathrm{r} - (A^* \setminus \{v\})$ that contains v. Let $w^*(v) = w'(C_v)$. Note that $G'_\mathrm{r}[A^*]$ and w^* correspond to the tree and weight function obtained by contracting each C_v into a single vertex. If $V(C_v) = \{v\}$, i.e., v is not an inner boundary vertex, then there are no contractions and $w^*(v) = w'(v)$.

We compute $w'(C_v)$ for each inner boundary vertex v, and temporarily modify the weight of each v in the top tree on $V(A^*)$ to match $w^*(v)$. Then, we find a centroid c of $G'_\mathrm{r}[A^*]$ w.r.t. w^*, and undo the weight change.[2]

By lemma 4.13, a centroid of G'_r must be contained in C_v. We also know a centroid must be contained in A^*. Thus, by theorem 4.12, a centroid must be contained in $C_c \cap A^* = \{c\}$, so c is a centroid of (G'_r, w'). If c is not virtual, it also is a centroid of (G_r, w). If it is virtual, the linked non-virtual vertex in G_r is a centroid of (G_r, w).

The running time to find A^* is $\mathcal{O}(|V(\mathcal{S}_\mathrm{r})|)$. Computing $w(C_v)$ for each inner boundary vertex v can be done while traversing $\mathcal{S}_\mathrm{r}$, also in $\mathcal{O}(|V(\mathcal{S}_\mathrm{r})|)$ time. There are at most $\deg_{\mathcal{S}_\mathrm{r}}(A^*) \leq |V(\mathcal{S}_\mathrm{r})|$ inner boundary vertices, so changing the weights in the top tree takes $\mathcal{O}(|V(\mathcal{S}_\mathrm{r})| \log k)$ time.

[2]Undoing the weight change is necessary, since we re-use the top tree data structures in later steps.

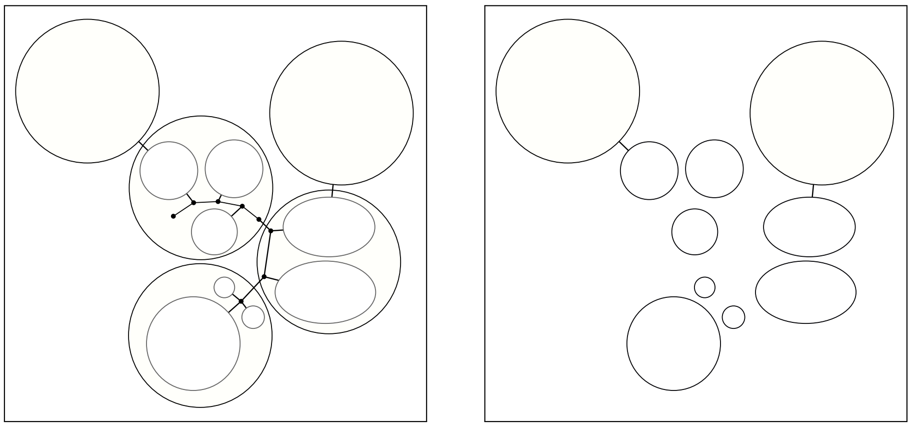


Figure 4.3.: A splitting step. Left: The super-tree before splitting. The large black circles are super-vertices. The small dots are is the set of vertices R to be removed. The gray circles and ellipses are components obtained after removing R. Right: The ten super-vertices within eight super-trees obtained after splitting.

Splitting the data structure. We now remove c and its virtual vertices from G'_{r} and $\mathcal{S}_{\mathrm{r}}$. Let R be the set of vertices to be removed. First, we remove each $v \in R$ from its associated top tree by deleting all of its incident edges. This may split up each of the top trees into multiple new top trees. Each new top tree corresponds to a new super-vertex.

The creation of new super-vertices may change the super-tree and split it up into a super-forest (see figure 4.3 for an example). Let A be a super-vertex from which we removed a vertex. For each inner boundary vertex u of A, we find the new top tree to which u belongs. Using this, we can compute all super-edges incident to the new super-vertices (recall that each super-edge corresponds to a normal edge between two inner boundary vertices in different super-vertices). Finally, we compute the connected components of the new super-forest $\mathcal{F}_{\mathrm{r}}$ using a simple traversal.

Removing a vertex v from the top trees requires $\mathcal{O}(\log k)$ time, because $\deg_{G'_{\mathrm{r}}}(v) \leq 3$. Since each inner boundary vertex is the endpoint of an edge in $\mathcal{S}_{\mathrm{r}}$, there are at most $\mathcal{O}(|E(\mathcal{S}_{\mathrm{r}})|) = \mathcal{O}(|V(\mathcal{S}_{\mathrm{r}})|)$ inner boundary vertices. The running time of splitting the super-forest is thus $\mathcal{O}(|R| \log k + |V(\mathcal{S}_{\mathrm{r}})| \log k + |V(\mathcal{F}_{\mathrm{r}})|)$.

Recursion. After splitting, we recurse on each component of the super-forest that contains more than one super-vertex or a super-vertex A with $|V(A)| \geq 2$. Components consisting of a single normal vertex do not have to be considered further.

Total running time. The preprocessing time is $\mathcal{O}(n \log k)$, as desired.

Consider one search-and-split step. Let G'_{r} be the input tree, let $\mathcal{S}_{\mathrm{r}}$ be the input super-tree, let R be the set of vertices removed in the splitting step, and let $\mathcal{F}_{\mathrm{r}}$ be the super-forest produced after splitting. The time required to execute the step is

$$\mathcal{O}\left(|V(\mathcal{S}_{\mathrm{r}})| \log k + |V(\mathcal{F}_{\mathrm{r}})| + \sum_{v \in R} \log k \right).$$

Since each vertex is removed only once, the third term adds up to $\mathcal{O}(n \log k)$ over all recursive calls. We charge the second term to the recursive calls made by the current call; observe that the first term in those calls dominates it. The only exception are components of $\mathcal{F}_\mathrm{r}$ that consist of a single super-vertex with a single normal vertex, since those are not covered by recursive call. In that case, we charge the cost to that normal vertex, for a total cost of $\mathcal{O}(n)$.

It remains to bound the first term. If $\mathcal{S}_\mathrm{r}$ consists of only one isolated super-vertex A, then we charge the $\mathcal{O}(\log k)$ cost to the centroid of $G'_\mathrm{r}[A]$, for a total of $\mathcal{O}(n \log k)$ over the course of the algorithm.

We analyze the other recursive calls in rounds. Assume that in each round, we execute one step on each remaining super-tree, thereby finding all centroids on a certain level of the centroid tree. Let $\mathcal{F}_0$ be the initial super-tree, and let $\mathcal{F}_i$ be the forest of super-trees after round i. Note that the number of *edges* cannot grow by splitting, so each $\mathcal{F}_i$ contains at most $\mathcal{O}(\frac{n}{k})$ edges, and therefore at most $\mathcal{O}(\frac{n}{k})$ non-isolated vertices, so the round requires $\mathcal{O}(\frac{n}{k} \log k)$ time in total.

If the height of the tree is h, then we have precisely h rounds. Thus, the running time of the algorithm is $\mathcal{O}((\frac{h}{k} + 1) n \log k)$.

In particular, if we know h and set $k = h$, then the running time is $\mathcal{O}(n \log h)$. If we do not know h, we start with $k = 2$ and run the algorithm for k rounds. If it stops, then $h \leq k$ and we are done. Otherwise, try again with $k \leftarrow k^2$. The last run of the algorithm, where $h \leq k \leq h^2$, dominates the running time with $\mathcal{O}(n \log h)$. Thus, we have:

Theorem 4.32. *Let G be a tree on n vertices and w be a positive weight function. We can compute a centroid tree of (G, w) in time $\mathcal{O}(n \log h)$, where h is the height of the computed centroid tree.*

4.7.2. Spread and zero-weight vertices

Define the *spread* of a weight function as the ratio between the maximum and minimum positive weight. In this section, we bound the cost of our output-sensitive algorithm in terms of the spread, obtaining an improvement when the spread is less than exponential in the number of vertices (which is very much expected in practice). Observe that we can arbitrarily scale the weight function without affecting the computed centroid tree; hence we can assume that all non-zero weights fall within the interval $[1, W]$, where W is the maximum weight (and also the spread).

With some adjustments to our algorithm to properly handle zero-weight vertices, we will obtain:

Theorem 4.7. *Let (G, w) be a given weighted tree with n vertices such that all weights are in $\{0\} \cup [1, W]$ for some $W \in \mathbb{R}_{\geq 0}$. Let m be the number of vertices with positive weight. We can compute a centroid tree on (G, w) in time $\mathcal{O}(n + m \log\log(m + W))$ or $\mathcal{O}(n + m \log m)$, whichever is lower.*

We start with the easy case where all weights are positive and within $[1, W]$. Let T be a centroid tree of (G, w). By definition, if u is a parent of v, then $w(T_v) \leq \frac{1}{2} w(T_u)$. More generally, if v has depth d in T, then $w(T_v) \leq 2^{1-d} w(G)$. Thus, the depth of a vertex v cannot be more than $1 + \log \frac{w(G)}{w(v)} \leq 1 + \log(nW)$. This implies that the running time of the algorithm of theorem 4.32 is $\mathcal{O}(n \log\log(nW)) = \mathcal{O}(n \log\log(n+W))$. The positive-weight case of theorem 4.7 follows.

If we allow zero-weight vertices, however, then the depth bound does not necessarily hold, since there could be a large unbalanced zero-weight subtree. Also, the top-tree component of our algorithm does not allow zero weights for technical reasons. It seems that the top tree implementation of Alstrup et al. [AHLT05] can be adapted to this changed requirement, but for a simpler presentation, we prefer to use the standard interface.

A solution to both problems is to change the zero weights by a very small amount, as follows. Let (G, w) be a weighted tree, and let $\varepsilon > 0$. We define the positive weight function $w_\varepsilon \colon V(G) \to \mathbb{R}_+$ on G as follows:

w_ε

$$w_\varepsilon(v) = \begin{cases} w(v), & \text{if } w(v) \neq 0 \\ \varepsilon, & \text{otherwise.} \end{cases}$$

We now prove that a centroid tree of (G, w_ε) is also a centroid tree of (G, w) if ε is small enough. For this, it is enough to show that for every subgraph, no new centroids are introduced.

Lemma 4.33. *Let (G, w) be a weighted tree on n with at least one positive-weight vertex. For some small enough $\varepsilon > 0$, each centroid of (G, w_ε) is a centroid of (G, w).*

Proof. Let $k < n$ be the number of zero-weight vertices in (G, w). Let c be a centroid of (G, w_ε) and let $C \in \mathbb{C}(G - c)$. We have

$$w(C) \leq w_\varepsilon(C) \leq \frac{1}{2} w_\varepsilon(G) = \frac{1}{2}\left(w(G) + \varepsilon k\right) < \frac{1}{2} w(G) + \frac{1}{2}\varepsilon n.$$

If ε is small enough (e.g., the minimum positive absolute difference between the weights of any two disjoint vertex sets), this implies that $w(C) \leq \frac{1}{2} w(G)$. Repeating the argument for each $C \in \mathbb{C}(G - c)$ shows that c is a centroid of (G, w). □

Note that if all weights are integers, we can simply set $\varepsilon = 1/n$ (or set $\varepsilon = 1$ and multiply each other weight by n). We could also instead treat ε symbolically as an infinitesimally small value, without explicitly computing a value for it.

We now show that the height of the centroid tree w.r.t. w_ε is essentially bounded by the spread of w; in other words, replacing w with w_ε ensures that the computed centroid tree is "reasonable". Note that we can ignore the spread of w_ε here (which may be very large, if ε is very small).

Lemma 4.34. *Let (G, w) be a weighted tree with n vertices such that all weights are in $\{0\} \cup [1, W]$ for some $W \in \mathbb{R}_{\geq 0}$, and let $\varepsilon > 0$ be defined as in lemma 4.33. Each centroid tree of (G, w_ε) has height $\mathcal{O}(\log n + \log W)$.*

Proof. Let T be a centroid tree of (G, w_ε). By lemma 4.33, T is also a centroid tree of (G, w). Thus, the depth of each node v with positive weight is at most $1 + \log(n \cdot W) = 1 + \log n + \log W$. This also covers all zero-weight vertices with positive-weight descendants.

Now consider a zero-weight leaf v. Let T_u be the maximal rooted subtree containing v with $w(T_u) = 0$. All nodes in T_u have weight ε w.r.t. W_ε, so the spread of w_ε restricted to $V(T_u)$ is 1. This means that the depth of v in T_u is at most $(1 + \log n)$. By maximality of T_u and the bound for positive-weight vertices, we have $\mathrm{depth}_T(u) \leq 2 + \log n + \log W$. This implies $\mathrm{depth}_T(v) \in \mathcal{O}(\log n + \log W)$, as desired. □

Lemmas 4.33 and 4.34 imply that we can replace w with w_ε before running the algorithm. We now have:

Lemma 4.35. *Let (G, w) be a weighted tree on n vertices such that all weights are in $\{0\} \cup [1, W]$ for some $W \in \mathbb{R}$. Then we can compute a centroid tree on (G, w) in time $\mathcal{O}(n \log\log(n + W))$.*

When the number of positive-weight vertices is sublinear, we can speed up the algorithm by first applying the reduction of theorem 3.10. If m is the number of positive-weight vertices, then theorem 3.10 reduces the number of overall vertices to $2m$ (without changing the weights of remaining vertices), so the running time of lemma 4.35 becomes $\mathcal{O}(m \log\log(m + W))$. If $W \geq 2^m$, we can just use theorem 4.31 for a running time of $\mathcal{O}(m \log m)$. This concludes the proof of theorem 4.7.

4.8. Hardness of computing centroid trees

In this section, we show that theorem 4.32 is tight for essentially all n and h (theorem 4.8), and theorem 4.7 is tight for some range of parameters (theorem 4.9). We start with a generic lower bound construction.

Lemma 4.36. *Let $k \geq 1$ and $\ell \geq 1$ be integers. There is a tree $G_{k,\ell}$ on $k \cdot (\ell + 1) + 1$ vertices and a class $\mathcal{W}_{k,\ell}$ of weight functions on $V(G_{k,\ell})$ such that:*

(i) $|\mathcal{W}_{k,\ell}| = (\ell!)^k$.

(ii) Each $w \in \mathcal{W}_{k,\ell}$ has codomain $[1, 2^\ell]$.

(iii) For each $w \in \mathcal{W}_{k,\ell}$, the weighted tree $(G_{k,\ell}, w)$ has a unique centroid tree C^w, and C^w has height either $\ell + 1$ or $\ell + 2$.

(iv) For each pair of distinct $w, w' \in \mathcal{W}_{k,\ell}$, we have $C^w \neq C^{w'}$.

Proof. Suppose first that $k \geq 2$. Let $G_{k,\ell}$ consist of a vertex c, and k stars of size $\ell + 1$, with centers $v_1, \ldots, v_k$ adjacent to c. The remaining ℓ vertices in the star with center v_i are denoted $v_{i,1}, \ldots, v_{i,\ell}$, for all $1 \leq i \leq k$. See figure 4.4. Observe that indeed $|V(G_{k,\ell})| = k \cdot (\ell + 1) + 1$.

Let S_ℓ denote the family of permutations of $\{1, \ldots, \ell\}$. For permutations $\pi_1, \ldots, \pi_k \in S_\ell$, let $w = w_{\pi_1,\ldots,\pi_k}$ denote the weight function defined as $w(v_{i,j}) = 2^{\pi_i(j)}$, for all $1 \leq i \leq k$ and $1 \leq j \leq \ell$. In words, $w_{\pi_1,\ldots,\pi_k}$ assigns the weights $2^1, \ldots, 2^\ell$ to the non-central vertices of the i-th star, permuted according to π_i, for all i. For the remaining vertices,

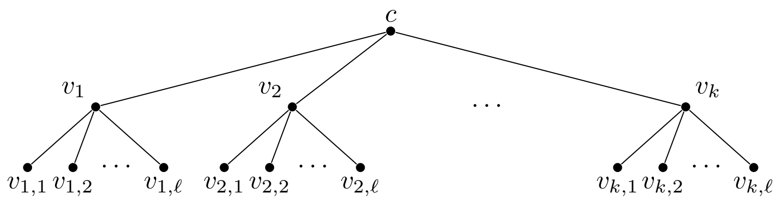

c
v_1
v_2
v_k
$\cdots$
$v_{1,1}$ $v_{1,2}$ $v_{1,\ell}$
$v_{2,1}$ $v_{2,2}$ $v_{2,\ell}$
$v_{k,1}$ $v_{k,2}$ $v_{k,\ell}$

Figure 4.4.: Tree $G_{k,\ell}$ in the proof of Lemma 4.36.

$w(v_1) = \cdots = w(v_k) = 0$, and $w(c) = 1$. Let $\mathcal{W}_{k,\ell}$ be the family of all such weight functions. Observe that $|\mathcal{W}_{k,\ell}| = (\ell!)^k$ and the maximum weight is 2^ℓ, so (i) and (ii) hold.

We claim that for any weight function $w \in \mathcal{W}$, the centroid tree of $(G_{k,\ell}, w)$ is unique, i.e., (iv) holds. Indeed, let $w = w_{\pi_1,\dots,\pi_k}$, and observe that the unique centroid of $(G_{k,\ell}, w)$ is c (recall that $k \geq 2$). The removal of c splits $G_{k,\ell}$ into the k stars with centers $v_1, \dots, v_k$. For all $1 \leq i \leq k$, the centroid tree of the star with center v_i is uniquely determined by the weight assignment by the permutation π_i, and it is easily seen to be the degenerate tree with nodes $v_{i,1}, \dots, v_{i,\ell}$ in decreasing order of weights, followed by the star center v_i. The entire search tree has height $\ell + 2$, thereby (iii) holds.

In the case $k = 1$, we omit the vertex c and directly build a single star. Claims (i) to (iv) are easy to verify if we build $\mathcal{W}_{k,\ell}$ as above. □

We are now ready to show our lower bounds.

Theorem 4.8. *Let $h \geq 3$ and $n \geq h + 1$ be integers. Then there exists a tree G with at most n vertices and a class $\mathcal{W}$ of weight functions on $V(G)$ such that for every $w \in \mathcal{W}$, every centroid tree of (G, w) has height at most h, and every binary decision tree that solves G for $\mathcal{W}$ has height $\Omega(n \log h)$.*

Proof. We use lemma 4.36 with $k = \lfloor \frac{n-1}{h-1} \rfloor$ and $\ell = h - 2$. The number of vertices in the obtained graph is $k \cdot (\ell + 1) + 1 \leq n$, and the height of each centroid tree is at most $\ell + 2 = h$.

The number of leaves in the decision tree is $|\mathcal{W}_{k,\ell}|$, so its height is at least $\log |\mathcal{W}_{k,\ell}| = \log((\ell!)^k) \in \Omega(\frac{n}{h} \log(h!)) = \Omega(n \log h)$. □

Theorem 4.8 implies that theorem 4.6 is tight (up to a constant factor) for all n and h. A slight adaptation of the argument yields:

Theorem 4.9. *Let $n \in \mathbb{N}$ and $W \in \mathbb{R}$ with $2 \leq W \leq 2^{n-2}$. Then there exists a tree G on at most n vertices and a class $\mathcal{W}$ of weight functions on $V(G)$ with codomain $[1, W]$, such that every binary decision tree that solves G for $\mathcal{W}$ has height $\Omega(n \log \log W)$.*

Proof. We use lemma 4.36 with $\ell = \lfloor \log W \rfloor \geq 2$ and $k = \lfloor \frac{n-1}{\ell+1} \rfloor \geq 1$. The number of vertices in the obtained graph is $k \cdot (\ell + 1) + 1 \leq n$, and the maximum weight of each $w \in \mathcal{W}_{k,\ell}$ is $2^\ell \leq W$, as desired. The height of each decision tree is $\log |\mathcal{W}_{k,\ell}| \in \Omega(k\ell \log \ell) = \Omega(n \log \log W)$. □

4.9. Optimal search trees are approximate centroid trees

In this section we prove the characterization of optimal STTs as α-centroid trees. We will make heavy use of the concept of *lifting* introduced in section 3.1.

Theorem 4.10. *Let T be an optimal search tree on a weighted tree (G, w). Then T is a $\frac{2}{3}$-centroid tree of (G, w).*

Proof. Let T be an optimal STT on G, and suppose towards contradiction that T is not a $\frac{2}{3}$-centroid. Assume w.l.o.g. that $w(G) = 1$. By taking T to be a minimum height counterexample, we can assume that the root x of T has a child y with $w(T_y) > \frac{2}{3}$. Let B_x

denote the set of vertices *not* in the same component of $G - x$ as y, and let B_y denote the set of vertices not in the same component of $G-y$ as x. Finally, let $B_{x,y} = V(T) - B_x - B_y$. See figure 4.5 (top left) for an illustration.

By our assumption, $w(B_y \cup B_{x,y}) = w(T_y) > \frac{2}{3}$, and thus, $w(B_x) < \frac{1}{3}$. We distinguish three cases (see figure 4.5 top, middle, bottom).

Case 1. If $w(B_y) > \frac{1}{3}$, then let $T^y = \operatorname{rot}(T, y, x)$. The depths of vertices in B_y decrease by one and the depths of vertices in B_x increase by one. The depths of vertices in $B_{x,y}$ are unchanged. We have

$$\operatorname{cost}_w(T^y) - \operatorname{cost}(T, w) = w(B_x) - w(B_y) < \frac{1}{3} - \frac{1}{3} = 0,$$

contradicting the optimality of T.

If Case 1 did not occur, we have $w(B_{x,y}) > \frac{1}{3}$, and in particular, $B_{x,y} \neq \emptyset$. Denote by z the unique child of y in $B_{x,y}$.

Case 2. Assume z is on the path in G between x and y. Let T^z be the search tree obtained by lifting z in T. The depths of vertices in B_x increase by one, since each of these vertices gains z as ancestor. The depths of vertices in B_y are unchanged, since these vertices gain z and lose x as ancestors. The depths of vertices in $B_{x,y}$ are decreased by *at least* one, since each loses at least one ancestor from $\{x, y\}$. We then have

$$\operatorname{cost}_w(T^z) - \operatorname{cost}(T, w) \leq w(B_x) - w(B_{x,y}) < \frac{1}{3} - \frac{1}{3} = 0,$$

again, a contradiction.

Case 3. Finally, assume that z is not on the path between x and y. Let $t \in B_{x,y}$ be the unique vertex in G that separates x, y and z. Let T^t be the search tree obtained by lifting t in T.

As in the previous case, depths of vertices in B_x increase by one, depths of vertices in B_y stay the same and depths of vertices $B_{x,y}$ decrease by *at least* one. To see this, observe that each vertex in $B_{x,y}$ gains t as ancestor and loses at least two ancestors from $\{x, y, z\}$. We again have $\operatorname{cost}_w(T^t) < \operatorname{cost}(T, w)$, a contradiction. $\square$

By a similar argument as previously in the chapter, theorem 4.10 implies that a vertex of weight $w(v) > 0$ has depth at most $\log_{3/2}(w(G)/w(v))$. Thus, the height of an optimal tree is at most $\log_{3/2}(w(G))$ if all weights are positive.

In the following, we expand this height guarantee to trees with zero-weight vertices. Because of the reduction proved earlier (theorem 3.10), we may ignore zero-weight vertices of degree at most two.

Theorem 4.37. *Let (G, w) be a weighted tree on n vertices with weights in $\{0\} \cup [1, \infty)$, such that all zero-weight vertices have degree at least three. Then, all optimal search trees on (G, w) have height at most $2\log_{3/2} w(G) - 2$, and there exists an optimal search tree with height at most $\log_{3/2} w(G) + \log\log w(G) + \mathcal{O}(1)$.*

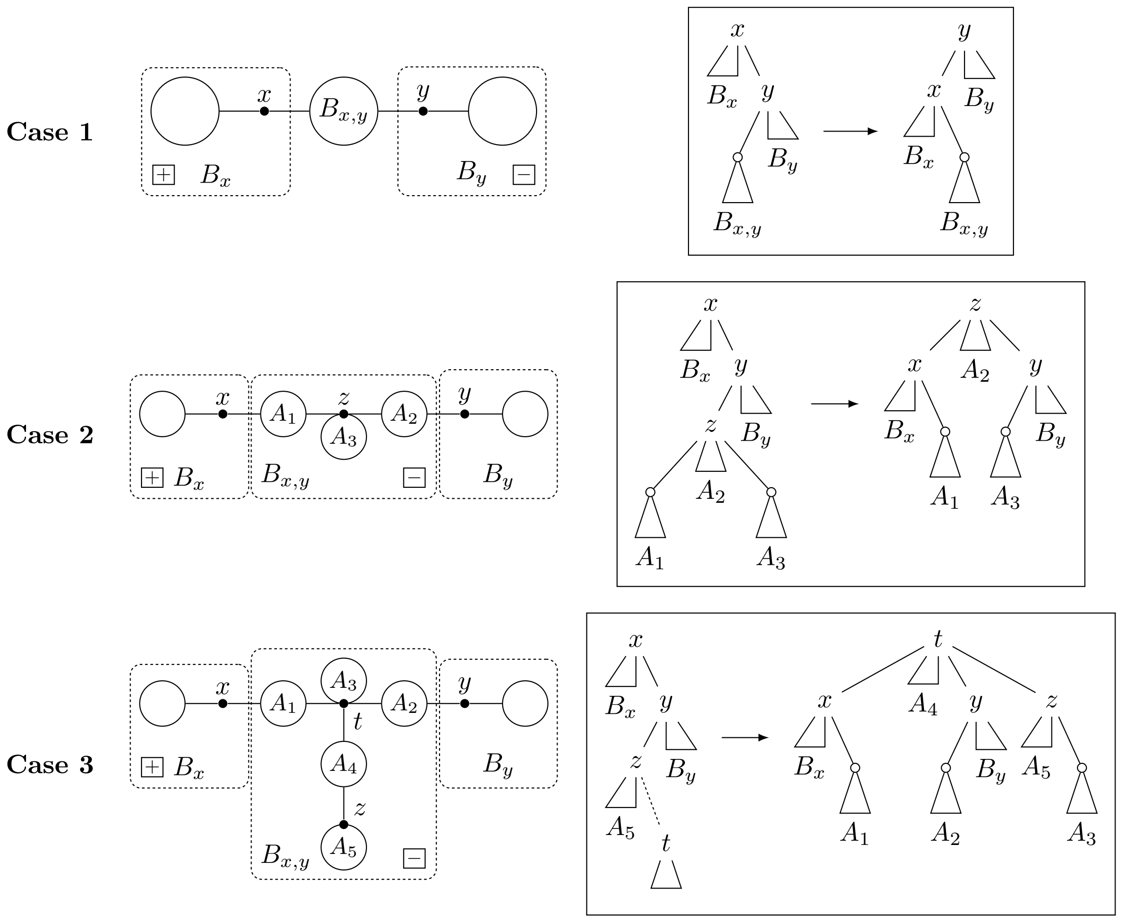


Figure 4.5.: Illustration of the proof of theorem 4.10. A sketch of the underlying graph is shown to the left and the search tree transformation to the right.

Proof. Let T be an optimal search tree on (G, w). As discussed above, all positive-weight vertices have depth at most $d = \log_{3/2} w(G)$. The same is obviously true for all zero-weight vertices that have a positive-weight descendant. Now consider a maximal rooted subtree T_v with only zero-weight vertices. Let $k = |V(T_v)|$ and let p denote the parent of v.

Since all vertices in $V(T_v)$ have degree at least three, we have $|\partial(T_v)| \geq k + 2$. Since all boundary vertices are ancestors of v (corollary 2.8), we have $\mathrm{depth}_T(p) \geq k + 2$. By maximality of T_v, either p has positive weight, or p has a descendant with positive weight. Hence, we have $\mathrm{depth}_T(p) \leq d$, implying $k \leq d - 2$.

In the worst case, the height of T_v is precisely k. Still, each node in T_v has depth at most $d + k \leq 2d - 2$, yielding the general upper bound. On the other hand, we can replace T_v by an unweighted centroid tree on $G[T_v]$, with height at most $\log k + 1$. This yields the second upper bound. □

Remark. A variant of theorem 4.37 can easily be shown for centroid trees instead of optimal search trees (cf. lemma 4.34). This is another way of solving the issue of unbalanced centroid trees in section 4.7.2. However, this does *not* solve the issue that the standard top tree interface does not allow zero weights; hence, we use a different approach in section 4.7.2.

5. Dynamic programming on k-cut search trees

In this chapter, we discuss a dynamic programming approach based on Knuth's algorithm for optimal static BSTs [Knu71]. Our main result is a polynomial-time approximation scheme (PTAS) for optimal static search trees on graphs with bounded tree-width. In other words, given a weighted graph (G, w) with bounded tree-width and an arbitrarily small $\varepsilon > 0$, our algorithm finds a search tree T on G with $\mathrm{cost}(T, w) \leq (1 + \varepsilon) \cdot \mathrm{StOPT}(G, w)$, in time $\mathcal{O}(n^{f(\varepsilon)})$ for some function f.

Naive dynamic programming. As a warm-up, let us describe a simple extension of Knuth's algorithm to search trees on general graphs. For this, we essentially need two observations.

First, by observation 3.11, the cost of a search tree can be recursively described as follows.

Observation 5.1. *Let (G, w) be a weighted graph and let T be a search tree on G. Let r be the root of T and K be the set of children of r. Then*

$$\mathrm{cost}(T, w) = w(T) + \sum_{c \in K} \mathrm{cost}(T_c, w).$$

Second, *subtree-optimality* holds, i.e., a subtree of an optimal STG is also optimal. This is an immediate consequence of observation 5.1: Any non-optimal subtree can be swapped with an optimal one to reduce the cost of the overall tree.

Let (G, w) be a connected weighted graph, and let $C[G'] = \mathrm{StOPT}(G', w)$ for each connected induced subgraph G' of G. The following recursion holds by the observations above:

$$C[G'] = \min_{r \in V(G')} w(G') + \sum_{H \in \mathbb{C}(G' - r)} C[H]. \tag{5.1}$$

This suggests a dynamic programming approach, which we sketch now. First, compute all connected induced subgraphs of G and store them in an array A. We then sort the graphs in A by number of vertices, in increasing order. Additionally, to ensure we can efficiently retrieve graphs from A, we fix an arbitrary order on the vertices of G, and sort the subgraphs in A with the same number of vertices according to lexicographic order of their sorted vertex sets.

Now, for each G' in A in order, compute $C[G']$ and store it in A. Whenever we need the value $C[H]$ in the recursion (5.1), we do a binary search for H in A. Since H has less vertices than G', we know that $C[H]$ has already been computed and stored in A.

This algorithm, which we call DP-Static, runs in time $\mathcal{O}(n^3 \cdot N \log N)$, where $n = |V(G)|$, and N is the number of connected induced subgraphs of G. It is not hard to compute the list of subgraphs in $\mathcal{O}(n \cdot N)$ time [Wer06b, KS21]. Comparing two subgraphs according to our order can be done in $\mathcal{O}(n)$ time, so each binary search takes $\mathcal{O}(n \log N)$ time, and the sorting takes $\mathcal{O}(n \cdot N \log N)$ time. Finally, the recursive formula 5.1 checks up to n possible roots r, computes $\mathbb{C}(G - r)$ in time $\mathcal{O}(m)$, where $m = |E(G)|$, and does up to $\mathcal{O}(n)$ searches in A for each r. Overall, the running time is

$$\mathcal{O}(n \cdot N + n \cdot N \log N + N \cdot n \cdot (m + n \cdot n \cdot \log N)) = \mathcal{O}(n^3 \cdot N \log N).$$

The algorithm can easily be modified to compute an actual optimal search tree (instead of only its cost). We thus have:

Theorem 3.3. *Given a weighted graph (G, w) with n vertices, we can find an optimal static STG on (G, w) in time $\mathcal{O}(n^3 \cdot N \log N)$, where $N \leq 2^n$ is the number of connected induced subgraphs of G.*

If G is a path, then $N < n^2$, so DP-Static runs in time $\mathcal{O}(n^5 \log n)$. But since every connected subgraph of G can be interpreted as an interval $[i, j]$, we can index subgraphs in constant time and likewise compute $\mathbb{C}(G' - r)$ in constant time, reducing the running time to $\mathcal{O}(n^3)$. Knuth [Knu71] improves this to $\mathcal{O}(n^2)$ with a certain trick that, unfortunately, seems to be specific to paths and certain very restricted trees [Vod23].

As mentioned in chapter 3, the number N may be exponential (even in the simple case of a *star*). However, since $N \leq 2^n$, the running time is always $\mathcal{O}(2^n \cdot n^4)$. This is already much better than the brute-force approach of trying every possible search tree: There are roughly 4^n search trees on a path [Sta15], there are $\Theta((n-1)!)$ search trees on a star, and $n!$ search trees on a clique (see section 2.2.5).

For trees with few leaves, the trivial bound on N can be improved considerably.

Proposition 5.2. *Let G be a tree with n vertices and with d leaves. Then G has $\mathcal{O}(n^d)$ connected induced subgraphs.*

We will prove proposition 5.2 in section 5.1. It follows that DP-Static runs in $\mathcal{O}(d \cdot n^{d+3} \log n)$ time if the input graph is a tree with d leaves.

A slightly weaker version of proposition 5.2 was observed by Høgemo, Bergougnoux, Brandes, Paul, and Telle [HBB$^+$21]. They also gave a further improvement of DP-Static if the weight function is uniform. It is based on the following observation: Each tree G admits a search tree T that minimizes $\operatorname{cost}(T, \mathbb{1})$ where all leaves of G are leaves in T. Thus, to find an optimal unit-cost search tree on G, it suffices to compute an optimal search tree T' on the search tree (G', w') obtained by contracting all leaves (transferring their weight to their neighbor), and then attaching all leaves of G as leaves to T', at the appropriate positions. We have:

Proposition 5.3. *Let G be a tree. Then $\operatorname{cost}(G, \mathbb{1})$ can be computed in time $\mathcal{O}(d \cdot n^{d+3} \log n)$, where d is the number of leaves in the graph that we obtain when removing all leaves from G.*

Approximation with k-cut search trees. In the following sections, we describe our PTAS. The main idea is to compute the exact optimum only on a subset of search trees; namely, on k-cut search trees (see section 2.2.1 for definitions). By definition, every rooted subtree of a k-cut search tree induces a connected k-cut subgraph.

It is not hard to adapt the dynamic programming algorithm described above to compute an optimal k-cut search tree on each connected induced k-cut subgraph, although there are some subtleties. The harder part is showing that k-cut trees are a good approximation to the general case. We now summarize the results in this chapter. In the following, a connected induced k-cut subgraph is called a *k-admissible subgraph.*

k-admissible subgraph

In section 5.1, we study the number of k-admissible subgraphs (and thus, subproblems to compute). We show:

Theorem 5.4. *Let G be a graph with n vertices and m edges. For each $k \in \mathbb{N}_+$, the number of k-admissible subgraphs of G is $\mathcal{O}(m \cdot n^{k-1})$.*

In particular, recall that graphs of tree-width t have $\mathcal{O}(t \cdot n)$ edges (lemma 2.37), and thus have at most $\mathcal{O}(t \cdot n^k)$ many k-admissible subgraphs.

In section 5.2 we describe a simple data structure that efficiently indexes subproblems, without requiring binary search. This data structure is used in our algorithm, which we describe in section 5.3.

Theorem 5.5. *Let (G, w) be a connected weighted graph with n vertices, and let $k \geq 2$ be an integer. We can compute an optimal k-cut search tree on (G, w) in time $\mathcal{O}(k^3 \cdot n^{k+2})$, using $\mathcal{O}(n^k)$ space. If no k-cut search tree exists (i.e., $\operatorname{tw}(G) > k$), then the algorithm reports that instead.*

If the underlying graph is a tree, we can improve the running time somewhat.

Theorem 5.6. *Let (G, w) be a weighted tree with n vertices, and let $k \geq 2$ be an integer. We can compute an optimal k-cut search tree on (G, w) in time $\mathcal{O}(k \cdot n^{k+1})$, using $\mathcal{O}(n^k)$ space.*

Finally, in section 5.4, we bound the approximation factor. Again, the tree case allows some improvement.

Theorem 5.7. *Let (G, w) be a weighted tree, let T be a search tree on G, and let $k \geq 2$. Then, there exists a k-cut search tree T' on G such that*

$$\operatorname{cost}(T', W) \leq \left(1 + \frac{1}{\lfloor k/2 \rfloor}\right) \cdot \operatorname{cost}(T, W).$$

Theorem 5.8. *Let (G, w) be a connected weighted graph of tree-width t, let T be a search tree on G, and let $k \geq 3t + 1$. Then, there exists a k-cut search tree T' on G such that*

$$\operatorname{cost}(T', W) \leq \left(1 + \frac{2t+2}{k - 3t}\right) \cdot \operatorname{cost}(T, W).$$

Combining these results, we obtain our two main theorems.

Theorem 3.4. *Given a weighted tree (G, w) and an integer $k \geq 2$, we can compute a search tree T on G in time $\mathcal{O}(k \cdot n^{k+1})$ such that*

$$\operatorname{cost}(T, w) \leq \left(1 + \tfrac{1}{\lfloor k/2 \rfloor}\right) \cdot \operatorname{StOPT}(G, w).$$

Theorem 3.5. *Given a weighted graph (G, w) with tree-width t and an integer $k \geq 3t + 1$, we can compute a search tree T on G in time $\mathcal{O}(k^3 \cdot n^{k+3})$ such that*

$$\operatorname{cost}(T, w) \leq \left(1 + \tfrac{2t+2}{k-3t}\right) \cdot \operatorname{StOPT}(G, w).$$

In contrast to the algorithms in chapter 4, these algorithms have no particular problems with handling zero-weight vertices. Lemma 3.14 can be directly applied to improve the running times as follows: if there are m vertices with positive weight, then the running times become $\mathcal{O}(k \cdot m^{k+1} + n)$ and $\mathcal{O}(k^3 \cdot m^{k+3} + n)$.

Discussion. The approximation algorithms in this chapter underscore the usefulness of k-cut search trees, and their connection to tree-width. Dynamic programming on k-cut search trees has been used for related problems [LK23], and further applications are to be expected.

The most direct avenue to improve our algorithms would be to improve the approximation factor. Especially for non-trees, the bound seems rather loose.

Open question 5.1. Can theorem 5.8 be improved or extended to cases $t \leq k \leq 3t$?

k-admissible root

Notation. If H is a k-admissible subgraph of a graph G, we also occasionally call the vertex set $V(H)$ *k-admissible* for brevity. For $v \in V(H)$, if all components of $H - v$ are k-admissible, then we call v a *k-admissible root* for H (or $V(H)$).

5.1. Counting k-cut subgraphs

In this section we show how to compactly store and efficiently access the list of k-admissible subgraphs, i.e., the list of subproblems. We start with bounding the length of that list.

Theorem 5.4. *Let G be a graph with n vertices and m edges. For each $k \in \mathbb{N}_+$, the number of k-admissible subgraphs of G is $\mathcal{O}(m \cdot n^{k-1})$.*

Proof. We give an encoding of each k-admissible subgraph. Let (u, v) be a tuple of adjacent vertices in G (like a *directed* edge) and let $B \subseteq V(G)$ be a set of $k - 1$ vertices. Observe that there is at most one k-admissible set $A \subseteq V(G)$ such that $v \in A$ and $\partial(A) = B \cup \{u\}$. On the other hand, each k-admissible set admits some encoding (u, v, B).

Observe that there are $2m$ choices for (u, v) and less than $\binom{n}{k-1}$ choices for B. Thus, the number of k-admissible subgraphs is at most $2 \cdot m \cdot \binom{n}{k-1} \in \mathcal{O}(m \cdot n^{k-1})$. □

Observe that theorem 5.4 is tight already for *stars*: Removing any k leaves yields a k-admissible subgraph. Thus, the number of k-admissible subgraphs of a star with n vertices is at least $\binom{n-1}{k}$.

We now make a slight detour and prove proposition 5.2, using theorem 5.4.

Proposition 5.2. *Let G be a tree with n vertices and with d leaves. Then G has $\mathcal{O}(n^d)$ connected induced subgraphs.*

Proof. Let H be a connected subgraph of G. We show that $|\partial_G(H)| \leq d$, which implies the claim by theorem 5.4.

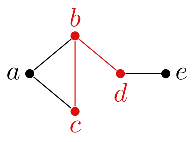
Each boundary vertex $b \in \partial_G(H)$ is contained in some connected component G_b of $G - H$, and no two boundary vertices are contained in the same component (otherwise, there would be a cycle). Further, each component G_b has at least two leaves (one of which may be b). Hence, G has at least $|\partial_G(H)|$ leaves, so $|\partial_G(H)| \leq d$. □

5.2. Indexing k-cut subgraphs

In this section, we show how to compactly store and access the list of k-admissible subgraphs, i.e., the list of subproblems. We will use an encoding similar to the one in the proof of theorem 5.4.

Fix a connected graph G and some total order on $V(G)$. A *representation* of a connected induced subgraph H of G a set $R \in V(G)^2$ of tuples (u, v). For each $(u, v) \in R$, we have $u \in \partial_G(H)$, $v \in V(H)$, and $\{u, v\} \in E(G)$, and each vertex $u \in \partial_G(H)$ occurs exactly once as the first element of a tuple in R. Shown below is an example of a 2-cut subgraph (red) with two possible representations R_1 and R_2.

representation



$$R_1 = \{(a, b), (e, d)\}$$
$$R_2 = \{(a, c), (e, d)\}$$

In the following, a representation is always stored as a list that is sorted by the first component of each tuple.

canonical representation

Recall that we fixed a total order on $V(H)$. We call a representation R of H *canonical* if for each $(u, v) \in R$, the vertex v is the minimal neighbor of u that is contained in $V(H)$. Observe that every k-admissible subgraph has a unique canonical representation of size k. Further observe that if G is a tree and $k \geq 2$, then *every* representation of size at least k is canonical, since each boundary vertex u has only one valid neighbor. This implies that all k-admissible subgraphs of trees are identified already by their boundary.

We now proceed with describing our data structure. First, the following easy lemma will be helpful.

Lemma 5.9. *Let $n, k \in \mathbb{N}_+$. There is a data structure that stores some value x_U for each subset $U \subseteq [n]$ of size at most k, using $\mathcal{O}(n^k)$ space. The data structures requires $\mathcal{O}(n^k)$ time to set up, with $x_H = \bot$ for all H initially.*

Given a set $U \subseteq [n]$ of size at most k as a sorted list, we can access (read or write) x_U in time $\mathcal{O}(k)$.

Proof. Maintain arrays $A_0, A_1, \ldots, A_k$. The array A_i has size precisely n^i and is indexed by tuples in $[n]^i$. Interpreting a given set U as a tuple lets us access the associated value. □

We are now ready to prove the main lemma of this section.

Lemma 5.10. *Let G be a connected graph, and let $k \in \mathbb{N}, k \geq 2$.*

There is a data structure that stores some value x_H for each k-admissible subgraph H of G, using $\mathcal{O}(n^k)$ space. The data structures requires $\mathcal{O}(n^k)$ time to set up, with $x_H = \bot$ for all H initially.

Given the canonical representation of a k-admissible subgraph H of G, we can access (read or write) x_H in time $\mathcal{O}(k)$ if G is a tree, or time $\mathcal{O}(k + \log n)$ otherwise.

Proof. Fix some order on $V(G)$. The basic idea is to *group* k-admissible graphs by boundary. We use lemma 5.9 to maintain a data structure A that allows accessing values associated with each set of at most k vertices of G. The values in A are initially empty *binary search trees*. Keys in the BSTs will be vertices in G. The running time of the initialization step is clearly $\mathcal{O}(n^k)$.

Suppose we access x_H with the canonical representation R of a k-admissible subgraph H. Let $i = |R|$ and suppose for now that $i \neq 1$.

Let $B = \partial_G(H)$. We can easily compute B and store it as a sorted list in $\mathcal{O}(i) \subseteq \mathcal{O}(k)$ time, since $B = \{u \mid (u, v) \in R\}$. We first find the binary search tree T in A that is associated to B.

Let u be the minimal vertex in B and let $v \in V(H)$ such that $(u, v) \in R$. Observe that v uniquely identifies H among induced subgraphs with boundary B.

We search for the key v in T. If the search is successful, we either return the attached value or change it, as required. If not, the we either return $\bot$ (read access) or insert u with the given value attached (write access). This clearly takes $\mathcal{O}(\log n')$ time, where n' is the number of items in T. We clearly have $n' \leq n$, so the access time is $\mathcal{O}(k + \log n)$.

If G is a tree, we can improve the access time somewhat. Recall that for each $B \subseteq V(G)$, since $|B| \neq 1$, there is at most one connected subgraph with boundary exactly B. Thus, every BST in the data structure has size at most one, so the overall access time is $\mathcal{O}(k)$.

It remains to consider the case $i = 1$. Let $\{(u, v)\} = R$. We simply maintain an array A' of size n^2, indexed by tuples in $[n]^2$. The value x_H is stored at position (u, v) in A'. The access time is clearly $\mathcal{O}(1)$. □

5.3. Finding an optimal k-cut search tree

We now describe our main algorithm. Fix a weighted graph (G, w) and a positive integer k. For each k-admissible subgraph H, we compute a search tree $S[H]$ on H such that $|\partial_G(S[H]_v)| \leq k$ for each $v \in V(S[H])$, and $\mathrm{cost}(S[H], w)$ is minimal for such a search tree. Write $S[H] = \bot$ if no such search tree exists or H is not k-admissible.

Observe that $S[H]$ is not necessarily an optimal k-cut search tree on H. This would only require all rooted subtrees of $S[H]$ to be k-cut in the subgraph H; instead, we require them to be k-cut in the whole graph G.

Let $A[H] = \mathrm{cost}(S[H], w)$, or $A[H] = \infty$ if $S[H] = \bot$. In the following, we focus on computing $A[H]$; our algorithm can easily be modified to compute $S[H]$. If R is the canonical representation of H, we write $A[R] = A[H]$ in the following.

The recursion from the general case (eq. (5.1)) clearly also holds here:

Observation 5.11. *The following recursive formula holds.*

$$\begin{aligned} A[H] &= \min_{r \in V(H)} w(H) + \sum_{C \in \mathbb{C}(H-r)} A[C] && \textit{if } H \textit{ is } k\textit{-cut;} \\ A[H] &= \infty && \textit{otherwise.} \end{aligned}$$

Algorithm 5.1 is a straight-forward implementation of the recursive formula. We store A using the data structure of lemma 5.10 as a cache. Accessing $A[R]$ works as follows. If $|R| > k$, then clearly $A[R] = \infty$. If $|R| \leq k$, we first check if R is stored in the data structure. If yes, we return the cached result. Otherwise, we compute Opt-k-cut-STG(R) and insert the result into the data structure.

The initial call, which computes the optimum cost of a k-cut search tree on G, is Opt-k-cut-STG$(\emptyset)$. Observe that it returns $\bot$ if there is no k-cut search tree on G, i.e., if the tree-width of G is strictly greater than k.

Line 3 of algorithm 5.1 deserves a more thorough explanation. We compute canonical representations of the components of $H - r$ as follows. Let $B = \partial_G(H) = \{u \mid (u, v) \in R\}$. For each neighbor u of r, perform a depth-first search starting at u that does not traverse edges towards any of the vertices $B \cup \{v\}$. This means we traverse the whole component C_u of $H - v$ that contains u. Computing the canonical representation of C_u is easily done during the traversal.

To avoid traversing components twice, we simply keep a global set of already visited vertices and ignore a neighbor u if it has been visited before.

This concludes the algorithm description. We proceed with the running time analysis. Let $n = |V(G)|$ and $m = |E(G)|$. By theorem 5.4, we solve a total of $m \cdot n^{k-1}$ subproblems.

Line 3 is executed up to n times, and each execution runs in time $\mathcal{O}(m)$. The running time of line 4 is dominated by the accesses to $A[\cdot,\cdot]$. We claim that the total number of such accesses over all iterations is at most $2m$. Indeed, each component C represented by R_i can be identified by the current root r and some neighbor $v \in V(C)$ of r. Hence, each edge corresponds to at most two accesses.

One access to A costs time $\mathcal{O}(k \log n)$ by lemma 5.10. Overall, we get a running time of $\mathcal{O}(n \cdot m + m \cdot k \log n) = \mathcal{O}(m(n + k \log n))$ for a single call to Opt-k-cut-STG(R). Using theorem 5.4 to bound the total number of calls, we have a total running time of $\mathcal{O}(n^{k-1} \cdot m^2 \cdot (n + k \log n)) \subseteq \mathcal{O}(k \cdot m^2 \cdot n^k)$.

Recall that if some k-cut search tree exists, then $\operatorname{tw}(G) \leq k$ and thus $m \leq k \cdot n$ by lemma 2.37. If we modify the algorithm to first check whether $m \leq k \cdot n$ and abort if not, we can bound the running time by $\mathcal{O}(k^3 \cdot n^k)$. Thus, we have:

Theorem 5.5. *Let (G, w) be a connected weighted graph with n vertices, and let $k \geq 2$ be an integer. We can compute an optimal k-cut search tree on (G, w) in time $\mathcal{O}(k^3 \cdot n^{k+2})$, using $\mathcal{O}(n^k)$ space. If no k-cut search tree exists (i.e., $\operatorname{tw}(G) > k$), then the algorithm reports that instead.*

Improvement for trees. If G is a tree, we can reduce the running time by computing line 3 of algorithm 5.1 more efficiently, as follows.

Lemma 5.12. *Let G be a graph, and let H be a k-admissible subgraph of G, given as its canonical representation R. Then, for all $r \in V(H)$, we can compute a list of canonical representations of the subgraphs $\mathbb{C}(H - r)$ in total time $\mathcal{O}(k \cdot n)$*

Algorithm 5.1 Finding an optimal k-cut STG

```
    Input: A weighted graph $(G, w)$ and the canonical representation $R$ of a connected induced subgraph $H$ of $G$.
    Output: $A[R] = A[H]$.
1: procedure Opt-$k$-cut-STG($R$)
2:     for $r \in V(H)$ do
3:         Compute canonical representations $R_1, R_2, \dots, R_\ell$ of components $\mathbb{C}(H - r)$
4:         $c_r = w(H) + \sum_{i=1}^{\ell} A[R_i]$
5:     return $\min_r c_r$
```

Proof. Start by constructing a rooting S of G at an arbitrary vertex. For each node $v \in V(S)$, compute $R_v = \{(u, v) \in R \mid v \in V(S_v)\}$. Store R_v as a sorted list. This can be done in time $\mathcal{O}(k \cdot n)$ in a bottom-up fashion (recall that R is sorted).

Now take some vertex v. The components of $H - r$ are the following:

- For each child c of r in S, the subgraph $H[S_c]$. The canonical representation of $H[S_c]$ is $R_c \cup \{(v, c)\}$ and can be computed in $\mathcal{O}(k)$ time (since R_c is sorted).
- If v has a parent p, then there is one more component $G - V(S_v)$, and its canonical representation is $R \setminus R_v \cup \{(v, p)\}$, which again can be computed in $\mathcal{O}(k)$ time.

Hence, the running time for each v is $\mathcal{O}(k \cdot (n_v + 1))$, where n_v is the number of children of v, adding up to $\mathcal{O}(k \cdot n)$ overall. □

In the modified algorithm, the running time of line 3 is thus $\mathcal{O}(k \cdot n)$ over all iterations in a single call of Opt-k-cut-STG. Further, by lemma 5.10 each access to A needs only $\mathcal{O}(k)$ time. This works out to a total running time of $\mathcal{O}(1 + k \cdot n + n \cdot k) = \mathcal{O}(n \cdot k)$ per subproblem. By theorem 5.4, we have $\mathcal{O}(m \cdot n^{k-1}) = \mathcal{O}(n^k)$ subproblems. Thus:

Theorem 5.6. *Let (G, w) be a weighted tree with n vertices, and let $k \geq 2$ be an integer. We can compute an optimal k-cut search tree on (G, w) in time $\mathcal{O}(k \cdot n^{k+1})$, using $\mathcal{O}(n^k)$ space.*

5.4. k-cut search trees approximate depth

In this section, we show theorems 5.7 and 5.8. Our proof is algorithmic, i.e., we give an algorithm that transforms an arbitrary search tree T on G into a k-cut search tree T', such that the cost increases only by a small factor $1 + \varepsilon$. It is enough to show that our algorithm increases the depth of each node by no more than $1 + \varepsilon$. Indeed, observe that then $\operatorname{cost}(T', w) \leq (1 + \varepsilon) \cdot \operatorname{cost}(T, w)$.

We now give an informal description of the special case where G is a tree. Our algorithm performs a top-down traversal of the search tree T'. Whenever we encounter a node x that is *not* a k-admissible root of $V(T_x)$, we choose a k-admissible root $v \in V(T_x)$, and replace x with v by rotating v to the top of T_x (i.e., we *lift* v in T_x, see section 3.1). We then recurse on the children of x (or v, if we performed the replacement).

As we prove later, every k-admissible set of vertices in a tree has a k-admissible root. Thus, our algorithm indeed produces a k-cut search tree. To bound the cost, we need to ensure that the root replacement only happens every $\approx k/2$ steps on any given root path in the search tree. This means that the root path of every node v in T' has at most $\approx \text{depth}_T(v)/(k/2)$ "new" nodes, implying that its depth increases by no more than a factor of roughly $1 + 2/k$.

The key to avoid frequent root replacements is to carefully choose the k-admissible root v. Essentially, if $V(T_x)$ has boundary size k, then there exists a node v that "splits" the boundary into parts of size at most $k/2$ (see figure 5.1). This means that v is actually a $(k/2 + 1)$-admissible root. Further observe that, with every step from parent to child, the boundary size increases by at most one (by observation 2.7). Hence, only after $k/2 - 1$ more steps can the boundary size be k again and we may need to do a replacement step.

The algorithm sketched above gives a slightly worse approximation factor then stated in theorem 5.7, so some additional ideas are required. We give an improved version of the algorithm in section 5.4.1.

To extend the algorithm to graphs with bounded tree-width, there may not be a single suitable node v to replace x with. Instead, we may need multiple nodes to split the boundary of $V(T_x)$. That algorithm is presented in section 5.4.2.

5.4.1. Trees

In this section, we prove:

Theorem 5.7. *Let (G, w) be a weighted tree, let T be a search tree on G, and let $k \geq 2$. Then, there exists a k-cut search tree T' on G such that*

$$\text{cost}(T', W) \leq \left(1 + \frac{1}{\lfloor k/2 \rfloor}\right) \cdot \text{cost}(T, W).$$

We first need some definitions and technical lemmas. Let G be a tree and let $A \subseteq V(G)$ induce a connected subgraph of G. A vertex $v \in A$ is a *boundary centroid* of A (or $G[A]$) if for each component C of $G - v$, we have $|\partial(A) \cap V(C)| \leq |\partial(A)|/2$. See figure 5.1 for an example.

boundary centroid

A related, more general concept is that of *α-separators* [BGHK95, Bod98]. Let B be a (not necessarily connected) subset of vertices of G. A vertex $v \in V(G)$ is an *α-separator for B in G* if each component $C \in \mathbb{C}(G - v)$ contains at most $\alpha \cdot |B|$ many vertices of B.

α-separator

Note that a $\frac{1}{2}$-separator v for $B = V(G)$ in G is a classical centroid (see chapter 4). Further, if $B = \partial(A)$ for some connected subset $A \subseteq V(G)$ and $v \in A$, then v is a boundary centroid of A.

Lemma 5.13 ([BGHK95, Lemma 3.4])**.** *Let G be a tree and $B \subseteq V(G)$. Then there is a $\frac{1}{2}$-separator for B in G.*[1]

Lemma 5.13 is not quite sufficient for our purposes, since for boundary centroids we additionally require the property $v \in A$. We now show that it holds in all relevant cases.

[1]This also follows from the existence of weighted centroids: A $\frac{1}{2}$-separator for B is the same as a centroid of (G, w), where $w(v) = 1$ if $v \in B$ and $w(v) = 0$ otherwise.

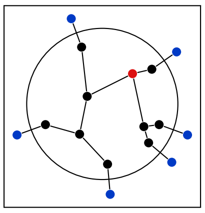


Figure 5.1.: A vertex (red) splits the $k = 6$ boundary vertices (blue) into parts of size at most $k/2 = 3$.

Lemma 5.14. *Let G be a tree and let $A \subseteq V(G)$ induce a connected subgraph of G. If $|\partial(A)| \geq 3$, then each $\frac{1}{2}$-separator for $\partial(A)$ in G is contained in A.*

Proof. Suppose $v \notin A$ is a $\frac{1}{2}$-separator for $\partial(A)$ in G. Since $G[A \cup \partial(A)]$ is connected, all vertices in $\partial(A)$, except possibly v (if $v \in \partial(A)$) lie in a single component of $G - v$. Since $|\partial(A)| - 1 > |\partial(A)|/2$ by assumption, this implies that v is not a $\frac{1}{2}$-separator for $\partial(A)$ in G. □

If $|\partial(A)| = 2$, this is not true; each of the two boundary vertices is also a $\frac{1}{2}$-separator. However, every other vertex in the convex hull $\mathrm{ch}(\partial(A))$ is a boundary centroid. Hence, we have

Corollary 5.15. *Let G be a tree and let $A \subseteq V(G)$ induce a connected subgraph of G. If $|\partial(A)| \geq 2$, then A has a boundary centroid.*

Algorithm 5.2 shows our procedure to make a search tree k-cut. The initial call is $\textsc{Fix}(T, k, x)$ with $x = \mathrm{root}(T)$. We remark that in line 3, we find a boundary centroid of $V(T_c)$ instead of $V(T_x)$. The latter would make the correctness proof easier, but would yield a worse cost approximation factor in the end.

The following lemma is essential for proving correctness and the approximation guarantee.

Lemma 5.16. *Let S be a tree, let $k \geq 2$, let $A \subseteq V(G)$ be k-admissible, and let $x \in A$.*

Suppose that there is a component C of $G[A] - x$ that is not *k-admissible and let v be a boundary centroid of C. Then,*

(i) $|\partial(A)| = k$.

(ii) v *is a k-admissible root for A.*

Algorithm 5.2 Transforming an arbitrary STT into a k-cut STT.

```
    Input: search tree $T$ on a tree $G$, constant $k \geq 2$, node $x \in V(T)$.
1:  procedure Fix($T, k, x$)
2:      if $|\partial(T_c)| > k$ for some child $c$ of $x$ then    ▷ Check if $x$ is a $k$-admissible root
3:          $v \leftarrow$ boundary separator of $V(T_c)$
4:          Lift $v$ in $T_x$
5:          $x \leftarrow v$
6:      for each child $c$ of $x$ do
7:          Fix($T, k, c$)
```

(iii) The component of $G[A] - v$ that contains x has boundary size at most $\lceil k/2 \rceil$.

Proof. Part (i) follows from observation 2.6 and the assumptions that $|\partial(A)| \leq k$ and $|\partial(C)| > k$. More precisely, we have $\partial(C) = \partial(A) \cup \{x\}$, which will be useful soon.

For parts (ii) and (iii), consider a component C' of $G[A] - v$. Figure 5.2 sketches the situation. We need to show that $|\partial(C')| \leq k$, and $|\partial(C')| \leq \lceil k/2 \rceil$ if $x \in V(C')$. By observation 2.6, we have $\partial(C') \subseteq \partial(A) \cup \{v\}$. We bound the size of $U = \partial(C') \setminus \{v\}$.

Observe that $U \subseteq \partial(A) \subseteq \partial(C)$. Further, clearly $V(C') \cup U$ induces a connected subgraph of G, and thus is contained in a single component C'' of $G - v$, so $U \subseteq \partial(C) \cap V(C'')$. Now the fact that v is a boundary centroid of C implies that

$$|U| \leq |\partial(C) \cap V(C'')| \leq \lfloor |\partial(C)|/2 \rfloor = \lfloor \tfrac{k+1}{2} \rfloor = \lceil \tfrac{k}{2} \rceil \leq k - 1.$$

Since $|\partial(C')| = |U| + 1$, this implies (ii).

Now suppose $x \in V(C')$. Then $x \in V(C'') \setminus U$. Since also $x \in \partial(C)$ by definition, using the same argument as before, we have $|U \cup \{x\}| \leq \lceil \frac{k}{2} \rceil$ and thus $|U| \leq \lceil \frac{k}{2} \rceil - 1$, implying $|\partial(C')| \leq \lceil \frac{k}{2} \rceil$. □

For the remainder of this section, consider a call $\textsc{Fix}(T, k, \mathrm{root}(T))$ with input search tree T. Let T^* denote the output, i.e., the search tree after the call.

Observe that each call $\textsc{Fix}(T, k, x)$ only affects nodes within T_x, though it does depend on nodes outside of T_x, namely $\partial(T_x)$.

Correctness. The following proves that algorithm 5.2 produces a k-cut search tree.

Lemma 5.17. *At the start of each recursive call* $\textsc{Fix}(T', k, x)$*, the set* $V(T'_x)$ *is* k*-admissible.*

Proof. We proceed by induction. For the initial call, we have $x = \mathrm{root}(T')$, so T'_x is 0-admissible.

Now take some recursive call $\textsc{Fix}(T', k, x)$, and assume that $V(T'_x)$ is k-admissible. If $V(T_c)$ is k-admissible for each child c of x in T', then the claim holds for all recursive calls in line 7.

Now suppose that x has some child c with $|\partial(T_c)| > k$, and let v be the boundary centroid of $V(T_c)$. Then v is a k-admissible root for $V(T_c)$ by lemma 5.16. Since we rotate v to the top of T_x, all recursive calls in line 7 are valid. □

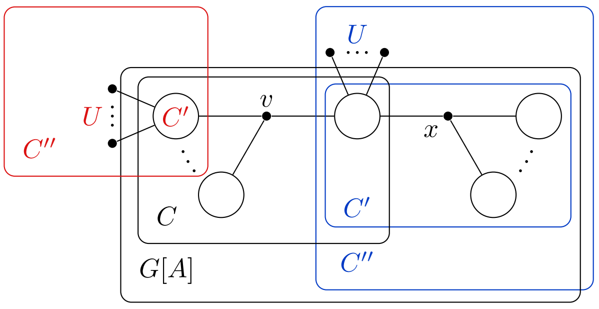


Figure 5.2.: Illustration of the proof of lemma 5.16. The case $x \notin V(C')$ is labeled in red, the case $x \in V(C')$ is labeled in blue.

Cost increase. We say a node x is *marked* if there is a recursive call $\textsc{Fix}(\,\cdot\,, x)$ where lines 3 to 5 were executed. We also say that the call $\textsc{Fix}(\,\cdot\,, x)$ *marks* x. We next show that, essentially, a node x is not touched again after being marked.

Lemma 5.18. *Consider a call* $\textsc{Fix}(T', x)$ *that marks* x*, and let* T'' *be the search tree after executing line 5. Then* x *has a parent* p *in* T^**, and we have* $V(T^*_p) = V(T'_x)$ *and* $V(T^*_x) = V(T''_x)$*.*

Proof. Observe that $\textsc{Fix}(T', x)$ makes the computed boundary centroid v a parent of x, and that $V(T''_v) = V(T'_x)$. By the recursive structure of the algorithm, there will be no later calls $\textsc{Fix}(\,\cdot\,, v)$. This already implies that $V(T^*_v) = V(T''_v) = V(T'_x)$.

Since x is a child of v in T'', there will be another call $\textsc{Fix}(\,\cdot\,, x)$. However, lemma 5.16 implies that $|\partial(T''_x)| < k$, so x must be a k-admissible root of $G[T''_x]$ (by observation 2.7), and therefore lines 3 to 5 are not executed. Thus, x stays a child of v, and will not be involved in further recursive calls. We conclude $V(T^*_x) = V(T''_x)$ and $p = v$, the latter of which implies $V(T^*_p) = V(T'_x)$. □

With this in mind, we prove the following two technical lemmas. First, we show that a root path cannot contain many marked nodes.

Lemma 5.19. *Let* x, y *be distinct marked nodes such that* x *is an ancestor of* y *in* T^**. Then,* $\operatorname{depth}_{T^*}(x) - \operatorname{depth}_{T^*}(y) \geq \lfloor k/2 \rfloor + 1$*.*

Proof. Let p be the parent of y in T^*. By lemma 5.18, we have $V(T^*_p) = V(T'_y)$, where T' is the tree at the beginning of the call $\textsc{Fix}(T', y)$ that marks y. Lemma 5.16 implies that $|\delta(T^*_p)| = k$.

Moreover, observe that $|\delta(T^*_x)| \leq \lceil |\delta(T^*_p)|/2 \rceil = \lceil k/2 \rceil$; this follows from lemmas 5.16 and 5.18. Finally, observation 2.7 implies that $\operatorname{depth}_{T^*}(p) - \operatorname{depth}_{T^*}(x) \geq |\delta(T^*_x)| - |\delta(T^*_p)| \geq \lfloor k/2 \rfloor$. Since $\operatorname{depth}_{T^*}(y) = \operatorname{depth}_{T^*}(p) + 1$, this concludes the proof. □

Now, we show how increasing depths are linked to marked nodes.

Lemma 5.20. *Let* $u \in V(T)$*, and let* m *be the number of marked nodes that are ancestors of* u *in* T^**. Then* $\operatorname{depth}_{T^*}(u) \leq \operatorname{depth}_T(u) + m$*.*

Proof. The depth of a node can only increase when some recursive call $\textsc{Fix}(T', x)$ executes lines 3 to 5. Let T'' be the search tree at line 5.

Observe that T''_v is obtained by lifting v in T_x. Hence, by observation 3.12, lines 3 to 5 can only increase the depth of the nodes in the component C of $G - v$ that contains x, and only by one. Clearly, we have $V(C) = V(T''_x)$, since x is a child of v in T''_v. Observe that x is a marked node and that $V(T''_x) = V(T^*_x)$ by lemma 5.18. Thus, the depth of a node $u \in V(T^*)$ increases at most once for each marked ancestor of u. □

By lemma 5.19, the root path of a node u in T^* contains at most

$$\frac{1}{\lfloor k/2 \rfloor + 1} \cdot \operatorname{depth}_{T^*}(u)$$

marked nodes. By lemma 5.20, we thus have

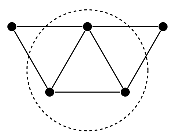
$$\begin{aligned}\operatorname{depth}_{T^*}(u) &\le \operatorname{depth}_T(u) + \frac{1}{\lfloor k/2\rfloor + 1}\cdot \operatorname{depth}_{T^*}(u)\\ \implies \operatorname{depth}_{T^*}(u) &\le \left(1+\frac{1}{\lfloor k/2\rfloor}\right)\cdot \operatorname{depth}_T(u).\end{aligned}$$

This concludes the proof of theorem 5.7.

Remark. It is tempting to try extending the approximation algorithm (with some ratio $(1+\varepsilon) > 2$) to the the easiest $k = 1$ case, i.e., when the STT is a rooted version of G. Unfortunately, 1-cut trees cannot give an $o(n/\log n)$-approximation of the STT optimum. To see this, take G to be a path, and observe that every rooting of G has average depth $\Omega(n)$, whereas a balanced BST on G (which is, in particular, a 2-cut tree) has maximum depth $O(\log n)$.

5.4.2. Graphs with bounded tree-width

We now turn to graphs with bounded tree-width. Unfortunately, a boundary centroid is no longer guaranteed to exist; in fact, there can even be k-admissible subsets without a k-admissible root. For example, in the picture below, the circled subgraph is 2-admissible, but has no 2-admissible root.



The solution is to use *sets* of vertices in place of a single boundary centroid. For this, we introduce the following concept.

Let G be a graph and let $A \subseteq V(G)$ induce a connected subgraph of G. A set $S \subseteq A$ is a *t-boundary centroid set* of A (or $G[A]$) if $|S| \le t+1$ and each component of $G[A \setminus S]$ has boundary size at most $\frac{1}{2}|\partial(A)| + t$.

t-boundary centroid set

It is not hard to show that if G is a tree and c is a boundary centroid for A in G, then $\{c\}$ is a 1-boundary centroid set for A in G. We now show the existence of small boundary centroid sets in graphs with bounded tree-width.

Lemma 5.21. *Let G be a connected graph of tree-width t, and let $A \subseteq V(G)$ induce a connected subgraph of G. Then there exists a t-boundary centroid set for A in G.*

Proof. Since $\operatorname{tw}(G) \le t$, there exists a t-cut search tree T on G. By lemma 5.13, the unrooting of T has a $\frac{1}{2}$-separator s for $\partial(A)$. We claim that $S = (\partial(T_s) \cup \{s\}) \cap A$ is a t-boundary centroid for A in G.

Clearly, $|S| \le t+1$. Let H be a connected component of $G[A] - S$. We need to show that $\partial(H) \le \frac{1}{2}|\partial(A)| + t$.

First, consider the case that $V(H) \subseteq V(T_s)$. Since either $s \notin A$ or $s \in S$, we have $s \notin V(H)$, and thus actually $V(H) \subseteq V(T_c)$ for some child c of s in T. This implies

$\partial(H) \subseteq V(T_c) \cup \partial(T_c)$. Since also $\partial(H) \subseteq S \cup \partial(A)$ by observation 2.6 and $V(T_c) \cap S = \emptyset$, we have

$$|\partial(H)| \leq |V(T_c) \cap \partial(A)| + |\partial(T_c)| \leq \tfrac{1}{2}|\partial(A)| + t$$

where the second inequality holds because s is a $\frac{1}{2}$-separator of $\partial(A)$ in the unrooting of T and T is t-cut.

Second, consider the case that $V(H)$ is disjoint from $V(T_s)$. We claim that $\partial(H)$ is then also disjoint from $V(T_s)$. Indeed, suppose that $b \in \partial(H) \cap V(T_s)$, and let v be a neighbor of b in $V(H)$. Then we have $v \in \partial(T_s)$. Since $v \in V(H) \subseteq A$, this implies $v \in S$, which contradicts the definition of H.

Using disjointness of $\partial(H)$ and $V(T_s)$ together with the previous observation $\partial(H) \subseteq S \cup \partial(A)$, we have

$$|\partial(H)| \leq |(S \cup \partial(A)) \setminus V(T_s)| \leq |S \setminus \{s\}| + |\partial(A) \setminus V(T_s)| \leq t + \tfrac{1}{2}|\partial(A)|.$$

The second inequality again holds because s is a $\frac{1}{2}$-separator of $\partial(A)$ in the unrooting of T.

Finally, suppose $V(H)$ contains both a vertex $u \in V(T_s)$ and a vertex $v \notin V(T_s)$. Since H is connected, w.l.o.g., there is an edge between u and v. But then $v \in \partial(T_s)$, and since $v \in A$, this implies $v \in S$, contradicting the definition of H. Thus, this third case cannot occur and we are done. □

We are now ready to consider algorithm 5.3, which serves as an existence proof of well-approximating k-cut trees. It is based on a simplified version of algorithm 5.2. The initial call $\textsc{Fix}(T, t, k, \mathrm{root}(T))$ transforms T into a k-cut search tree.

Fix $k \geq 3t + 1$ and consider a call $\textsc{Fix}(T, t, k, x)$ where lines 3 and 4 are executed. We then say this call is a *marking* call and the vertices in S are *marked* by it. Note the difference to section 5.4.1, where x was considered marked instead of S. Let T' be search tree before the recursion (after line 4).

Since lines 3 and 4 are a bit more complicated then their equivalent in algorithm 5.2, we study their effect in more detail. First, since T_x is replaced by a tree whose root is in S, we have:

Observation 5.22. *Let $s^* = \mathrm{LCA}_{T'}(S)$. Then $s^* \in S$ and $V(T'_{s^*}) = V(T_x)$, and thus $|\partial(T'_{s^*})| = k - t$.*

Algorithm 5.3 Transforming an STG into a k-cut STG.

```
   Input: search tree $T$ on a graph $G$, constant $t \geq \mathrm{tw}(G)$, constant
   $k \geq 3t + 1$, node $x \in V(T)$.
1: procedure $\textsc{Fix}(T, t, k, x)$
2:     if $|\partial(T_x)| = k - t$ then
3:         $S \leftarrow t$-boundary centroid set of $V(T_x)$
4:         Lift $S$ in $T_x$                              ▷ See section 3.1.
5:     for each child $c \notin S$ of some $s \in S$ do
6:         $\textsc{Fix}(T, k, c)$
```

Second, the boundary sizes of nodes in S and their children can be bounded as follows.

Lemma 5.23. *We have*

(i) $|\partial(T'_s)| \leq k$ *for each* $s \in S$*; and*

(ii) $|\partial(T'_c)| \leq \frac{k+t}{2}$ *for each child* $c \notin S$ *of some* $s \in S$ *in* T'*.*

Proof. Let $s^* \in S$ such that $V(T'_{s^*}) = V(T_x)$, as established in observation 5.22. We have $|\partial(T'_{s^*})| = |\partial(T_x)| = k - t$. Now take some $s \in S \setminus \{s^*\}$. We have $s^* \prec_{T'} s$ and the path from s^* to s in T' contains only nodes in S, since S is a prefix of T'_{s^*}. Thus, $\partial(T'_s) \subseteq \partial(T'_{s^*}) \cup S \setminus \{s\}$ by observation 2.7, implying $|\partial(T'_s)| \leq k - t + t = k$.

Now let c be a child of some $s \in S$. Clearly $V(T_c)$ corresponds to a component of $G[V(T'_{s^*})] - S$, and thus has boundary size at most $\frac{1}{2}|\partial(T'_{s^*})| + t = \frac{k+t}{2}$, since S is a t-boundary centroid set. □

From lemma 5.23 and observation 2.7, and the fact $\partial(T) = \emptyset$, it follows:

Lemma 5.24. *If* $t \leq k$*, and if* T *is a search tree on a graph of tree-width at most* t*, then* $\textsc{Fix}(T, t, k, \mathrm{root}(T))$ *transforms* T *into a* k*-cut search tree.*

It remains to show the cost-approximation ratio of algorithm 5.3. From now on, let T^* be the final tree produced by $\textsc{Fix}(T, t, k, \mathrm{root}(T))$. Clearly, every node is marked at most once, and not touched again afterwards; hence, the facts established in observation 5.22 and lemma 5.23 for marked nodes also hold in T^*.

Lemma 5.25. *Let* s, s' *be nodes that both marked, but not by the same call, and let* $s \prec_{T^*} s'$*. Then* $\mathrm{depth}_{T^*}(s') - \mathrm{depth}_{T^*}(s) \geq 1 + \frac{1}{2}(k - 3t)$*.*

Proof. Lemma 5.23 implies that there must be a node x on the path between s and s' in T^* that is the child of some marked node and satisfies $|\partial(T^*_x)| \leq \frac{k+t}{2}$. Let s^* be the lowest-depth node in T^* that was marked by the same call as s'.

We have $s \prec_{T^*} x \preceq_{T^*} s^* \preceq_{T^*} s'$, and we have $|\partial(T^*_{s^*})| = k - t$ by observation 5.22. On the other hand, we have $|\partial(T^*_x)| \leq \frac{k+t}{2} < k - t$ by assumption (recall that $k \geq 3t + 1$). This means that $x \neq s^*$ and thus $x \prec_{T^*} s^*$. With this in mind, we get

$$\begin{aligned} \mathrm{depth}_{T^*}(s') - \mathrm{depth}_{T^*}(s) &\geq \mathrm{depth}_{T^*}(s^*) + 1 - \mathrm{depth}_{T^*}(x) \\ &\geq (k - t) + 1 - \tfrac{1}{2}(k + t) = 1 + \tfrac{1}{2}(k - 3t). \end{aligned}$$ □

The following lemma finishes the proof of theorem 5.8.

Lemma 5.26. *For each* $v \in V(G)$*, we have*

$$\mathrm{depth}_{T^*}(v) \leq \left(1 + \frac{2t + 2}{k - 3t}\right) \cdot \mathrm{depth}_T(v).$$

Proof. Since only marked nodes are lifted, the difference $d = \mathrm{depth}_{T^*}(v) - \mathrm{depth}_T(v)$ is at most the number of marked nodes on the root path P of v in T^*. Suppose these marked nodes are marked in p different calls. Then $d \leq p \cdot (t + 1)$.

Lemma 5.25 implies that P has $\frac{1}{2}(k - 3t)$ unmarked nodes between any two nodes that were marked in different calls. Since the topmost marked node in P has boundary size

$k-t$ by observation 5.22, it must have at least $k-t-1 \geq k-3t$ (unmarked) ancestors. Thus, we have

$$\begin{aligned}
& & \operatorname{depth}_{T^*}(v) = |P| &\geq (k-t-1) + (p-1)\cdot \tfrac{1}{2}(k-3t) + d & \\
& & &\geq p \cdot \tfrac{1}{2}(k-3t) + d & \\
&\Longrightarrow & \operatorname{depth}_T(v) &\geq p \cdot \tfrac{1}{2}(k-3t) & \text{by def. of } d \\
& & &\geq \tfrac{d}{t+1} \cdot \tfrac{1}{2}(k-3t) & \text{since } d \leq p\cdot(t+1) \\
&\Longrightarrow & d &\leq \frac{2t+2}{k-3t}\operatorname{depth}_T(v). & \square
\end{aligned}$$

6. An FPTAS for trees

In this section, we give a fully-polynomial-time approximation scheme (FPTAS) for the optimal static STT problem (note that the underlying graph is always *tree* in this chapter). That is, we give a $(1+\varepsilon)$-approximation algorithm with running time $\operatorname{poly}(1/\varepsilon) \cdot \operatorname{poly}(n)$, where n is the number of vertices in the input tree. The main ingredient is the following algorithmic result (section 6.1).

Theorem 6.1. *Given a weighted tree (G, w) on n vertices and $h \in \mathbb{N}_+$, the minimum-cost search tree on G with height at most h can be computed in $\mathcal{O}(4^h \cdot n)$ time, if such a search tree exists (i.e., $\operatorname{td}(G) \leq h$).*

Theorem 6.1 implies that the problem is *fixed-parameter tractable* (FPT) for the parameter "height of the optimal search tree".

To obtain an FPTAS, we need two more steps. First, we have shown earlier that the height of the (unrestricted) optimal search tree is logarithmic in the *spread* of the non-zero weights (section 4.9). This gives us a *pseudo-polynomial* algorithm (section 6.2).

Theorem 6.2. *Let (G, w) be a given weighted tree with n vertices with weights in $\{0\} \cup [1, \infty)$, and let m be the number of positive-weight vertices. We can compute an optimal search tree on (G, w) in time*

$$\mathcal{O}(w(G)^{2/\log(3/2)} \cdot m \cdot \log^2 w(G) + n) \subseteq \mathcal{O}(w(G)^{3.42} \cdot m + n).$$

Observe that theorem 3.7 from chapter 3 is a special case of theorem 6.2, since $w(G) \leq W \cdot n$ if all weights are at most W. Finally, standard techniques [IK75] allow transforming the pseudo-polynomial algorithm into an FPTAS (section 6.3).

Theorem 3.8. *For each $\varepsilon > 0$, there exists an algorithm that computes a search tree T on a given weighted tree (G, w) such that $\operatorname{cost}(T, w) \leq (1+\varepsilon) \cdot \operatorname{StOPT}(G, w)$, in time*

$$\mathcal{O}\left((\tfrac{1}{\varepsilon})^{2/\log(2/3)} \cdot n^{1+4/\log(2/3)} \cdot \log^2 \tfrac{n}{\varepsilon} \right) \subseteq \mathcal{O}\left((\tfrac{1}{\varepsilon})^{3.42} \cdot n^{7.84} \right).$$

The contents of this section are heavily based on the work of Cicalese, Jacobs, Laber, and Molinaro [CJLM14]. They present an FPTAS for *edge-query trees* (EQTs; see section 1.3) on trees with bounded degree with the three steps outlined above. Naturally, transferring their result to STTs necessitates some changes. Perhaps the most notable difference is that they require the underlying tree to have bounded degree; this is necessary to bound the height of the optimal EQT. Our algorithm needs no such assumption, and we get the height bound from our previous observations on centroid trees (see section 4.9).

Note that our algorithm is presented in a self-contained way; no knowledge of edge-query trees is necessary.

6.1. Computing bounded-height optimal search trees

In this section, we show:

Theorem 6.1. *Given a weighted tree (G, w) on n vertices and $h \in \mathbb{N}_+$, the minimum-cost search tree on G with height at most h can be computed in $\mathcal{O}(4^h \cdot n)$ time, if such a search tree exists (i.e.,* $\operatorname{td}(G) \leq h$*).*

Recall that the straight-forward dynamic programming approach presented in chapter 5 builds an optimal STG by progressively building optimal search trees on k-admissible subgraphs, which then serve as rooted subtrees of the overall solution.

Here, we again use dynamic programming, but the subproblems are quite different. We only consider a small number of subgraphs. For each of those subgraphs, we build all possible corresponding *parts* of a final STT, which may not even be a connected subtree. To represent these parts, we use *generalized STTs*, defined below.

Generalized STTs. A *generalized search tree* (GSTT) T on a tree G is a rooted tree with $V(T) = V(G) \cup N$, where N is a set of so-called *null nodes* that is disjoint from $V(G)$. We denote by $\tilde{V}(T_v)$ the set of non-null nodes in T_v, i.e., $\tilde{V}(T_v) = V(T_v) \cap V(G)$. A GSTT has the following properties.

(i) For each $v \in V(T)$, we have $\tilde{V}(T_v) \neq \emptyset$ and the subgraph of G induced by $\tilde{V}(T_v)$ is connected.

(ii) For each edge $\{u, v\} \in E(G)$, either u is an ancestor of v in T, or v is an ancestor of u in T.

Observe that every STT is a GSTT without any null nodes, by lemma 2.3. Intuitively, it is clear that adding null nodes cannot improve the height or cost of a search tree. We formalize this idea with the following lemma.

Lemma 6.3. *Let T be a generalized search tree on at tree G. Then there is a (non-generalized) search tree T' on G such that for each $v \in V(G)$, we have* $\operatorname{depth}_{T'}(v) \leq \operatorname{depth}_T(v)$.

Proof. We describe a way to progressively remove all null nodes from T, while retaining the GSTT properties and without increasing the depth of any node.

Let v be a null node in T. If v has no children, we can simply remove v. Otherwise, let $c_1, c_2, \ldots, c_k$ be the children of v. We modify the tree as follows. First, delete v, which splits off the subtrees $T_{c_1}, T_{c_2}, \ldots, T_{c_k}$. Put T_{c_1} at the place where v previously was, i.e., make c_1 the root if v was the root, or otherwise make c_1 a child of the previous parent of v. Then make $c_2, c_3, \ldots, c_k$ each a child of c_1. Call the new tree T'

Observe that (i) this operation does not increase the depth of any node, (ii) any ancestor-descendant pair of non-null nodes in T is still present in T', and (iii) all rooted subtrees are preserved except T_{c_1}, but we have $\tilde{V}(T'_{c_1}) = \tilde{V}(T_v)$, which induces a connected subgraph of G by definition. Hence, the result is a valid GSTT and we eventually obtain an STT. □

GSTT cost

If (G, w) is a weighted tree and T is a GSTT on G, we write $\operatorname{cost}(T, w) = \operatorname{cost}(T, w')$, where $w'(v) = w(v)$ for each $v \in V(G)$, and $w'(v) = 0$ for each null node of T. Lemma 6.3 implies that for each GSTT T on G, there is a search tree T' with $\operatorname{cost}(T', w) \leq \operatorname{cost}(T, w)$ and $\operatorname{height}(T') \leq \operatorname{height}(T)$. This implies:

Corollary 6.4. *Let (G, w) be a weighted tree and $h \in \mathbb{N}_+$. If G admits a search tree of height at most h (i.e., $\operatorname{td}(G) \leq h$), then the minimum cost of a search tree on G with height at most h is equal to the minimum cost of a GSTT on G with height at most h.*

Combining GSTTs. Let v be a node in a GSTT T and let $(u_1, u_2, \ldots, u_k = v)$ be root path of v. Define $\mathrm{P}(T, v) = \{i \in [k] \mid u_i \notin N\}$, i.e., $\mathrm{P}(T, v)$ is the set of depths of non-null nodes on the root path of v. We now show how two "compatible" GSTTs can be combined into a larger one.

Lemma 6.5. *Let G be a tree, let $e = \{x, y\}$ be an edge in G, and let G_x, G_y be the two components in $G - e$, such that $x \in V(G_x)$ and $y \in V(G_y)$. Let T^x be a generalized search tree on G_x and let T^y be a generalized search tree on G_y, such that $\mathrm{P}(T^x, x)$ and $\mathrm{P}(T^y, y)$ are disjoint. Then, there exists a generalized search tree T on G such that*

(i) For each $v \in V(G_x)$, we have $\operatorname{depth}_T(v) = \operatorname{depth}_{T^x}(v)$.

(ii) For each $v \in V(G_y)$, we have $\operatorname{depth}_T(v) = \operatorname{depth}_{T^y}(v)$.

(iii) $\mathrm{P}(T, x) = (\mathrm{P}(T^x, x) \cup \mathrm{P}(T^y, y)) \cap [\operatorname{depth}_T(x)]$.

Proof. The idea is to "merge" T^x and T^y along the root path of v. See figure 6.1 for an example.

Let $k = \operatorname{depth}_{T^x}(x)$ and $\ell = \operatorname{depth}_{T^y}(y)$, and let $m = \max(k, \ell)$. Let $(x_1, x_2, \ldots, x_k = x)$ be the root path of x in T^x, and let $(y_1, y_2, \ldots, y_\ell = y)$ be the path from the root to y in T^y.

We build T as follows. Start with a degenerate tree $z_1, z_2, \ldots, z_m$ of null nodes. Then, for each $i \in [m]$,

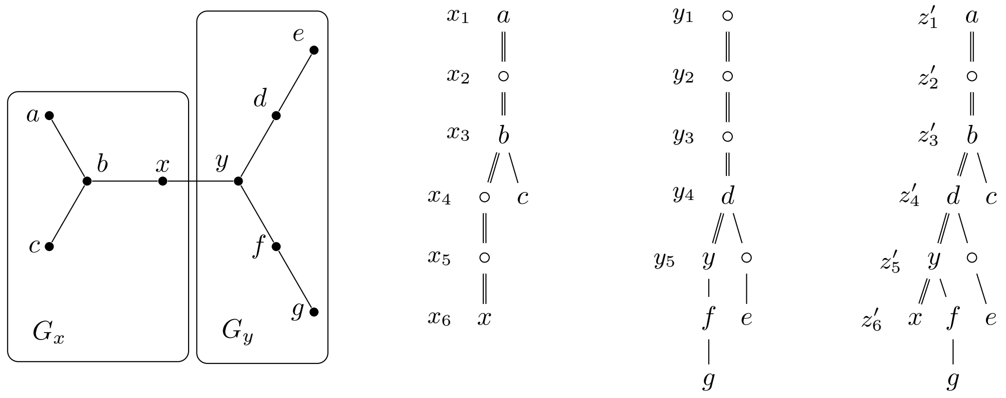
- if x_i exists and is not a null node, replace z_i with x_i;
- if y_i exists and is not a null node, replace z_i with y_i;
- for each child c of x_i in T^x, attach a copy of T^x_c to z_i; and
- for each child c of y_i in T^y, attach a copy of T^y_c to z_i.

Observe that for each $i \in [m]$, either x_i or y_i must be a null node by assumption that $\mathrm{P}(T^x, x)$ and $\mathrm{P}(T^y, y)$ are disjoint. Denote by z'_i the node z_i in T if it is still present, or otherwise the node that replaced z_i.

Parts (i) to (iii) of our claim are immediate from construction. It remains to show that T is indeed a valid GSTT.

To prove GSTT property (ii), first observe that each ancestor-descendant relation between non-null nodes in T^x or T^y is also present in T. It remains to handle the edge $\{x, y\}$. Observe that when constructing T, we replaced z_k with x and z_ℓ with y, hence $x \prec_T y$ if $k < \ell$ and $y \prec_T x$ if $\ell < k$. We cannot have $k = \ell$ since $P(T^x, x)$ and $P(T^y, y)$ are disjoint. Thus, GSTT property (ii) holds.

For GSTT property (i), take any node $v \in V(T)$. If $v \notin \{z'_1, z'_2, \ldots, z'_m\}$, then the whole subtree T_v was copied from T^x or T^y, and thus $\tilde{V}(T_v)$ induces a connected subgraph of


a
b
x
y
c
d
e
f
g
G_x
G_y
x_1 x_2 x_3 x_4 x_5 x_6
y_1 y_2 y_3 y_4 y_5
z'_1 z'_2 z'_3 z'_4 z'_5 z'_6

(a) Underlying tree G (b) T^x (c) T^y (d) Merged T

Figure 6.1.: Illustration of lemma 6.5. Null nodes are represented as circles ($\circ$). The double lines in the search trees indicate the paths $\mathrm{P}(T^x, x)$, $\mathrm{P}(T^y, y)$, and $\mathrm{P}(T, x)$.

either G_x or G_y by assumption. Similarly, if $v = z'_i$ for some $i > \ell$ or $i > k$, then T_v was copied from T^x or T^y, respectively.

Otherwise, we have $v = z'_i$ for some $i \le \min(k, \ell)$. Let $p, q \ge i$ be minimal such that $z'_p \in V(G_x)$ and $z'_q \in V(G_y)$. Observe that $x \in \tilde{V}(T^x_{z'_p})$, $y \in \tilde{V}(T^y_{z'_q})$, and $\tilde{V}(T_v) = \tilde{V}(T^x_{z'_p}) \cup \tilde{V}(T^y_{z'_q})$. Since there is an edge between x and y, this implies that $\tilde{V}(T_v)$ induces a connected subgraph of G. $\square$

Dynamic Programming algorithm. We now present an algorithm that computes the optimal GSTT with at most a given height.

Fix a weighted tree (G, w), and an arbitrary rooting S of G. For each connected subgraph G' of G, write $\mathrm{LCA}(G') = \mathrm{LCA}_S(V(G'))$. Recall that $\mathrm{LCA}(G') \in V(G')$ by lemma 2.4.

Let G' be a connected subgraph of G, let $h \in \mathbb{N}_+$, and let $A \subset [h]$. Then $\mathrm{C}_h[G', A]$ denotes the minimum cost of a GSTT T on (G', w) such that

- $\mathrm{height}(T) \le h$.
- $\mathrm{P}(T, \mathrm{LCA}(G')) \subseteq A$. In other words, each non-null node v on the root path of $\mathrm{LCA}(G')$ must satisfy $\mathrm{depth}_T(v) \in A$.

We now give a recursive formula for $\mathrm{C}_h[G', A]$. First, observe that if G' consists of precisely one node v, then an optimal GSTT can be constructed by taking a degenerate tree of $\min(A) - 1$ null nodes and leaf v, with cost $\min(A) \cdot w(v)$. Otherwise, we use the following recurrence.

Lemma 6.6. *Let H_1, H_2 be disjoint connected subgraphs of G such that $v_1 = \mathrm{LCA}(H_1)$ is the parent of $v_2 = \mathrm{LCA}(H_2)$ in S. Let $H = G[V(H_1) \cup V(H_2)]$. Then*

$$\mathrm{C}_h[H, A] = \min_{B \subseteq A} \mathrm{C}_h[H_1, B] + \mathrm{C}_h[H_2, A \setminus B].$$

Proof. Observe that $\mathrm{LCA}(H) = v_1$. We start with the "$\geq$" direction. Consider an optimal GSTT T on H with $\mathrm{height}(T) \leq h$ and $P(T, v_1) \subseteq A$. Obtain rooted trees T^1 (resp. T^2) from T by replacing each node in H_2 (resp., in H_1) with a null node, and then repeatedly removing all leaves that are null nodes, as long as there are any. Clearly T^i is an GSTT on H_i and $\mathrm{height}(T^i) \leq h$, for $i \in \{1, 2\}$. Let $B = P(T^1, v_1)$. Since either $v_1 \prec_T v_2$ or vice versa, we have $P(T^2, v_2) \subseteq A \setminus B$. By definition of $C_h[\,\cdot\,,\cdot\,]$, we have

$$\mathrm{C}_h[H, A] = \mathrm{cost}(T, w) = \mathrm{cost}(T^1, w) + \mathrm{cost}(T^2, w) \geq \mathrm{C}_h[H_1, B] + \mathrm{C}_h[H_2, A \setminus B].$$

We now prove the "$\leq$" direction. Fix some $B \subseteq A$ and let T^1, T^2 be optimal GSTTs on H_1, resp. H_2, such that $\mathrm{height}(T^1), \mathrm{height}(T^2) \leq h$ and $P(T^1, v_1) \subseteq B$, $P(T^2, v_2) \subseteq A \setminus B$. By lemma 6.5, there exists a GSTT T on H such that $\mathrm{cost}(T, w) = \mathrm{cost}(T^1, w) + \mathrm{cost}(T^2, w)$, $\mathrm{height}(T) \leq h$, and $P(T, v_1) \subseteq B \cup A \setminus B = A$. Hence, we have

$$\mathrm{C}_h[H, A] \leq \mathrm{cost}(T, w) = \mathrm{cost}(T^1, w) + \mathrm{cost}(T^2, w) = \mathrm{C}_h[H_1, B] + \mathrm{C}_h[H_2, A \setminus B]. \quad \square$$

In order to limit the number of subproblems to solve, we fix an arbitrary order on the children of each node in S. We only consider subgraphs of the following form. If $v \in V(G)$ and $c_1, c_2, \ldots, c_k$ are the children of v in S (in order), then let $G_{v,i}$ be the subgraph induced by $\{v\} \cup V(S_{c_1}) \cup V(S_{c_2}) \cup \cdots \cup V(S_{c_i})$, for $i \in \{0, 1, \ldots, k\}$. Let $G_v = G_{v,k}$ be the subgraph induced by $V(S_v)$. We compute $\mathrm{C}_h[G_{v,i}, A]$ for each $A \subseteq [h]$ and each such $G_{v,i}$. By lemma 6.6, we have

$$\begin{aligned} \mathrm{C}_h[G_{v,0}, A] &= \min(A) \cdot w(v), \text{ and} \\ \mathrm{C}_h[G_{v,i}, A] &= \min_{B \in [h] \setminus A} \mathrm{C}_h[G_{v,i-1}, B] + \mathrm{C}_h[G_{c_i}, [h] \setminus A \setminus B], \end{aligned}$$

Note that $G = G_{\mathrm{root}(S)}$. By corollary 6.4, the minimum cost of an STT of height at most h is thus $\mathrm{C}_h[G_{\mathrm{root}(S)}, \emptyset]$.

Overall, we solve $2^h \cdot (2n - 1)$ subproblems, and each needs $\mathcal{O}(2^h)$ time. This concludes the proof of theorem 6.1.

6.2. A pseudo-polynomial algorithm

The next step towards the FPTAS is using theorem 6.1 to obtain a pseudo-polynomial algorithm, which we do now.

Theorem 6.2. *Let (G, w) be a given weighted tree with n vertices with weights in $\{0\} \cup [1, \infty)$, and let m be the number of positive-weight vertices. We can compute an optimal search tree on (G, w) in time*

$$\mathcal{O}(w(G)^{2/\log(3/2)} \cdot m \cdot \log^2 w(G) + n) \subseteq \mathcal{O}(w(G)^{3.42} \cdot m + n).$$

Proof. First apply theorem 3.10 to remove all vertices with weight zero and degree at most two, ending up with at most $2m$ vertices. By theorem 4.37, there exists an optimal search tree with height at most $h = \log_{3/2} w(G) + \log\log w(G) + \mathcal{O}(1)$. Applying theorem 6.1 with this value for h computes an optimal search tree in the desired running time. $\square$

If all weights are in $\{0\} \cup [1, W]$, then $w(G) \leq m \cdot W$. Thus, we get the following result, which implies theorem 3.7 from chapter 3 as a special case.

Corollary 6.7. *Let (G, w) be a given weighted tree with n vertices such that $w\colon V(G) \to \{0\} \cup [1, W]$ for some $W \in \mathbb{R}$, and let m be the number of positive-weight vertices. We can compute an optimal search tree on (G, w) in time*

$$\mathcal{O}(W^{2/\log(3/2)} \cdot m^{1+2/\log(3/2)} \cdot \log^2(Wm) + n) \subseteq \mathcal{O}(W^{3.42} \cdot m^{4.42} + n).$$

6.3. Approximating optimal search trees

Let (G, w) be a weighted tree. To transform our pseudo-polynomial algorithm into an FPTAS, we follow standard techniques [IK75, CJLM14] and modify the weight function as follows.

Without loss of generality, we have $w(v) \in \{0\} \cup [1, \infty)$ for all $v \in V(G)$. Let $m \geq 1$ be the number of positive-weight vertices. Let $K = \varepsilon \cdot w(G)/m^2$ and let $w'(v) = \lceil w(v)/K \rceil$ for all $v \in V(G)$. We use theorem 6.2 to compute a minimum-cost search tree T' on (G, w').

We first bound the running time. Clearly, we have $w'(G) \leq w(G)/K + m \leq \frac{m^2}{\varepsilon} + m \leq 2 \cdot \frac{m^2}{\varepsilon}$. Hence, the running time is

$$\mathcal{O}((1/\varepsilon)^{\alpha} \cdot m^{2+\alpha} \cdot \log^2(m/\varepsilon) + n)$$

for $\alpha = 2/\log(3/2) < 1.71$.

We now show the approximation ratio. Let T be an optimal search tree on (G, w) and let T' be the optimal search tree on (G, w'), which is computed by our algorithm. We show that $\operatorname{cost}(T', w) \leq (1 + \varepsilon) \operatorname{cost}(T, w) = (1 + \varepsilon) \operatorname{StOPT}(G, w)$.

We have $\operatorname{cost}(T', w) \leq K \cdot \operatorname{cost}(T', w') \leq K \cdot \operatorname{cost}(T, w')$ by definition of w' and optimality of T'. Now define $\tilde{w}(v) = 1$ if $w(v) > 0$ and $\tilde{w}(v) = 0$ otherwise. Observe that $w'(v) \leq \frac{1}{K} w(v) + \tilde{w}(v)$. Then

$$\begin{aligned} K \cdot \operatorname{cost}(T, w') &\leq \operatorname{cost}(T, w) + \operatorname{cost}(T, \tilde{w}) \cdot K \\ &\leq \operatorname{cost}(T, w) + m^2 \cdot K \leq \operatorname{cost}(T, w) + \varepsilon \cdot w(G). \end{aligned}$$

Since $\operatorname{cost}(T, w) \geq w(G)$, we have $\operatorname{cost}(T, w) + \varepsilon \cdot w(G) \leq (1 + \varepsilon) \cdot \operatorname{cost}(T, w)$. Thus, we conclude:

Theorem 6.8. *For each $\varepsilon > 0$, there exists an algorithm that computes a search tree T on a given weighted tree (G, w) with m positive-weight vertices such that $\operatorname{cost}(T, w) \leq (1 + \varepsilon) \cdot \operatorname{StOPT}(G, w)$, in time*

$$\mathcal{O}\left((\tfrac{1}{\varepsilon})^{2/\log(2/3)} \cdot m^{1+4/\log(2/3)} \cdot \log^2 \tfrac{m}{\varepsilon} + n \right) \subseteq \mathcal{O}\left((\tfrac{1}{\varepsilon})^{3.42} \cdot m^{7.84} + n \right).$$

Theorem 3.8 follows as a special case.

7. Hardness

In this chapter, we prove the two hardness results: Theorem 3.6 in section 7.1, and theorem 3.9 in section 7.2

7.1. Edge-query trees and line graphs

In this section, we show that exactly computing optimal search trees is NP-hard already on graphs with bounded tree-width. The proof is a reduction to the problem of computing optimal *edge-query trees* (EQTs; see section 1.3). We start with a formal definition.

An edge-query tree (EQT) on a tree G is a rooted tree T with $V(T) = V(G) \cup E(G)$, such that $V(G)$ is precisely the set of leaves of T. For each node v, let G_v denote the subgraph of G induced by the set of leaves of T_v. The following holds:

G_v

- Each inner node $v \in V(T)$ has two children c and d, such that G_c and G_d are the two components obtained by removing the edge v from G_v.

Observe that G_v is a connected subgraph for each node v.

The optimal static STG problem is easily adapted to the EQT case. If $w\colon V(G) \to \mathbb{R}_{\geq 0}$ is a weight function on the underlying tree, then for an EQT T, we write $\operatorname{cost}(T, w) = \sum_{v \in V(G)} \operatorname{depth}_T(v) \cdot w(v)$. An *optimal static EQT* on (G, w) is an EQT on G that minimizes $\operatorname{cost}(T, w)$. We denote the cost of an optimal static EQT by $\operatorname{StOPT}^{\mathrm{EQT}}(G, w)$.

Theorem 7.1 (Cicalese, Jacobs, Laber, and Molinaro [CJLM11, §2.2])**.** *Computing an optimal static EQT on a given weighted tree (G, w) is NP-hard, even if G is a tree with maximum degree 16 and all non-leaves in G have weight zero.*

To transfer this result to STGs, we show a connection between EQT and STGs on *line graphs*. The *line graph* $L(G)$ of a graph G is the graph H with $V(H) = E(G)$, where two $x, y \in V(H)$ are adjacent in H if and only if the corresponding edges in G share a common endpoint.

line graph

Lemma 7.2. *Let G be a tree. Then, there exists a bijection between EQTs on G and search trees on $L(G)$, such that, for each EQT T and associated search tree S, the following holds.*

(i) For each $x \in E(G) = V(L(G))$, we have $\operatorname{depth}_T(x) = \operatorname{depth}_S(x)$.

(ii) For each $v \in V(G)$, we have $\operatorname{depth}_T(v) = 1 + \max_{u \in U_v} \operatorname{depth}_S(u)$, where U_v is the set of edges incident to the vertex v in G.

Proof. Write $H = L(G)$. We proceed by induction on $|V(G)|$. For convenience, let us say there is an empty search tree on the empty graph, and that $\max_{u \in U} \operatorname{depth}_T(u) = 0$ if U is empty. With this, the case $|V(G)| = 1$ is immediate.

Now suppose $|V(G)| > 1$. Let $x \in E(G) = V(H)$. It is easy to see that $G - x$ has two components C_1 and C_2, and $H - x$ has also two components C_1' and C_2', such that $C_1' = L(C_1)$ and $C_2' = L(C_2)$.

Consider an EQT T on G with root x. By induction, there are EQTs T^i on C_i and search trees S^i on C_i' such that T^i, S^i satisfy the lemma, for $i \in \{1, 2\}$. Build a search tree S on H by taking x as the root and S^1 and S^2 as child subtrees of x. Observe that we can reverse this construction, hence we have a bijection. It remains to show that T and S satisfy conditions (i) and (ii).

Clearly, condition (i) holds for T and S by induction. We proceed to show condition (ii). Let v be a vertex of G. W.l.o.g., assume $v \in V(C_1)$. Let $y \in E(G)$ be the edge incident to v with largest depth in S.

If $y \in E(C_1)$, then $y \in V(C_1') = V(S^1)$, and thus

$$\mathrm{depth}_T(v) = 1 + \mathrm{depth}_{T^1}(v) \overset{\text{(ind.)}}{=} 2 + \mathrm{depth}_{S^1}(y) = 1 + \mathrm{depth}_S(y),$$

as desired.

If $y \notin E(C_1)$, then $y = x$, and v is a leaf of G. This means that v is the only node in $V(C_1) = V(T^1)$, so $\mathrm{depth}_T(v) = 2 = 1 + \mathrm{depth}_S(y)$, as desired. □

We now compare the cost of the associated EQT T and search tree S of lemma 7.2. This is relatively straight-forward, except that the term $\max_{u \in U_v} \mathrm{depth}_S(u)$ cannot really be expressed by our cost model. However, it works if we assume that U_v is a singleton for all relevant v.

Lemma 7.3. *Let G be a graph and $w \colon V(G) \to \mathbb{R}_{\geq 0}$ be a weight function, such that every node $v \in V(G)$ with degree larger than one has weight zero. Let $w' \colon E(G) \to \mathbb{R}_{\geq 0}$ be given by $w'(x) = w(u) + w(v)$, where u, v are the two endpoints of the edge x in G.*

Then $\mathrm{StOPT}^{\mathrm{EQT}}(G, w) = \mathrm{StOPT}(L(G), w') + w(G)$.

Proof. Let T be an EQT on G and let S be the associated search tree on $L(G)$, as per lemma 7.2. We show that $\mathrm{cost}(T, w) = \mathrm{cost}(S, w) + w(G)$, which implies the claim.

Let U be the set of vertices of G with positive weight, and let $u \in U$. Then u is a leaf of G, hence it has a unique incident edge x_u. By lemma 7.2, we have $\mathrm{depth}_T(u) = 1 + \mathrm{depth}_S(x_u)$. Thus,

$$\begin{aligned}\mathrm{cost}(T, w) &= \sum_{u \in U} w(u) \cdot (1 + \mathrm{depth}_S(x_u)) = w(G) + \sum_{u \in U} w(u) \cdot \mathrm{depth}_S(x_u) \\ &= w(G) + \sum_{x \in E(G)} w'(x) \cdot \mathrm{depth}_S(x) = w(G) + \mathrm{cost}(S, w').\end{aligned}$$

□

Observe that the line graph of a tree with maximum degree d is a graph consisting of cliques of size at most d which are linked in a tree-like fashion via single vertices. The tree-width of the line graph is clearly at most $d - 1$. Thus, combining theorem 7.1 and lemma 7.3, we obtain:

Theorem 3.6. *Computing the optimal static search tree on a given weighted graph is NP-hard, even if the graph has tree-width at most 15.*

7.2. Unit-cost optima

In this section, we show that computing $\mathrm{StOPT}(G, \mathbb{1})$ is equivalent to the *trivially perfect completion* problem, as observed by [HBB+21]. This implies the former problem is NP-hard [Yan81]. See section 2.3 for the relevant definitions.

We start with a general lemma.

Lemma 7.4. *Let T be a rooted tree and let $G = \mathrm{cl}(T)$. Then $\mathrm{cost}(T, \mathbb{1}) = |V(G)| + |E(G)|$.*

Proof. The edges of G are precisely the ancestor-descendant pairs in T, so

$$|E(G)| = \sum_{v \in V(T)} (\mathrm{depth}_T(v) - 1) = \mathrm{cost}(T, \mathbb{1}) - |V(G)|. \qquad \square$$

We now show equivalence of the two problems. Recall that $\mathrm{TPC}(G)$ is the minimum number of edges in a trivially perfect supergraph of G.

Theorem 7.5 ([HBB+21])**.** *For each graph G with n vertices, we have $\mathrm{TPC}(G) = \mathrm{StOPT}(G, \mathbb{1}) - n$.*

Proof. By lemma 7.4, for each search tree T on G, we have $|E(\mathrm{cl}(T))| = \mathrm{cost}(T, \mathbb{1}) - n$. Since $\mathrm{TPC}(G)$ minimizes $|E(\mathrm{cl}(T))|$ over all search trees on G (by corollary 2.28), and $\mathrm{StOPT}(G, \mathbb{1})$ minimizes $\mathrm{cost}(T, \mathbb{1})$ over all search trees on G (by definition), our claim follows. $\square$

The trivially perfect completion problem can be shown to be NP-hard via a simple modification of a reduction by Yannakakis [Yan81]. We thus have:

Theorem 3.9 ([HBB+21])**.** *Computing $\mathrm{StOPT}(G, \mathbb{1})$ for a given graph G is NP-hard.*

Before moving on to the next chapter, we take a brief detour to show the following fact about trivially perfect graphs, which will be useful in chapter 11.

Lemma 7.6. *Let G be a trivially perfect graph with n vertices and m edges. Then $\mathrm{StOPT}(G, \mathbb{1}) = m + n$.*

Proof. We start with showing $\mathrm{StOPT}(G, \mathbb{1}) \leq m + n$. By definition, there exists a rooted tree T with $V(T) = V(G)$ such that $G = \mathrm{cl}(T)$. By lemma 2.25, we have that T is a search tree on G. Thus $\mathrm{StOPT}(G, \mathbb{1}) \leq \mathrm{cost}(T, \mathbb{1}) \leq m + n$ by lemma 7.4.

We now show $\mathrm{StOPT}(G, \mathbb{1}) \geq m + n$. Let T be a search tree on G. For each vertex v, denote by a_v the number of ancestors of v (in T) that are adjacent to v (in G). Observe that $a_v \leq \mathrm{depth}_T(v) - 1$. Since $m = \sum_{v \in V(G)} a_v$, we have $m \leq \mathrm{cost}(T, \mathbb{1}) - n$. This implies the statement. $\square$

Part II.

Dynamic search trees

8. Adaptive searching in trees

In this chapter, we study the dynamic search tree model for STGs discussed in the introduction (section 1.2). We start by formally describing our model, which is based on previous BST and STT models [Wil89, DHIP07, BCI+20]. Then, we discuss some caveats and alternate models.

In the rest of the chapter, we discuss various *adaptivity properties* of dynamic STG algorithms (section 8.1) and then study some simple STG algorithms with respect to these properties.

A problem instance consists of a connected graph G and a sequence $X \in V(G)^m$ of *accesses*. Note that individual vertices may be accessed zero or multiple times. An access corresponds to what we called a *search* in the introduction.

access

We now define what it means to *serve* an access $x \in V(G)$ in a search tree T on G. We start with a pointer at the root and repeatedly perform one of the following steps:

- Move the pointer from the current node to one of its children.
- Move the pointer from the current node to its parent.
- Rotate the current node with its parent.

At some point during the sequence of steps, the accessed vertex x must be visited. We call a valid sequence of steps S a *serve sequence* for x in T. The *cost* of S is the number of steps, plus one.[1] If T is the search tree at the start and T' is the search tree after executing S, we write $T' = \text{apply}(S, T)$.

serve sequence

cost

apply(S, T)

As mentioned in section 1.3, there are two variants of the dynamic search tree model. In the *offline* model, the whole access sequence X are known from the start. In the *online* model, the $(i+1)$-th access is only revealed after the i-th access is served. We now formally define the two variants.

Offline variant. An *offline dynamic search tree algorithm* is an algorithm that takes a connected graph G and an access sequence $X = (x_1, x_2, \dots, x_m) \in V(G)^m$ as input. It outputs an *initial* search tree T^0 and serve sequences $S_1, S_2, \dots, S_m$, such that

offline algorithm

(i) Each S_i is a serve sequence for x_i; and

(ii) There exist search trees $T^1, T^2, \dots, T^m$ such that $T^i = \text{apply}(S_i, T^{i-1})$.

The *cost* of the algorithm for (G, X) is the sum of the costs of $S_1, S_2, \dots, S_m$. We define DynOPT(G, X) as the minimum cost of an offline algorithm to serve X.

DynOPT$(\dots)$

online algorithm

Online variant. An *online dynamic search tree algorithm* consists of two parts. The first part is an algorithm that computes an initial search tree T^0 on a given graph G. The second part is an algorithm that, given a search tree T on a graph G and a vertex $x \in V(G)$, computes a serve sequence $S(G, T, x)$ for x in T. Here x is given as a pointer to the vertex x in G *and* as a pointer to the node x in T.

Consider a connected graph G and an access sequence $X = (x_1, x_2, \dots, x_m) \in V(G)^m$. Intuitively, the algorithm starts with T^0, and whenever an access x_i is revealed, it executes $S(G, T, x_i)$, where T is the current search tree.

More formally, let $T^1, T^2, \dots T^m$ be inductively defined as $T^i = \text{apply}(S_i, T^{i-1})$, with $S_i = S(G, T^{i-1}, x_i)$. Then the *cost* of the algorithm for (G, X) is the sum of the costs of $S_1, S_2, \dots, S_m$.

Discussion. We defined our models with a focus on simplicity; this comes with a number of caveats.

First, note that the actual running time of the algorithm to compute the initial tree and serve sequences is ignored, since we only consider the *cost.* However, each major algorithm presented in this thesis can be implemented with running time that is linear in its cost.

Second, for each access x, the location of the node x in the current search tree T is given. Hence, actually *searching* for x in T is not necessary. However, observe that serving x in T always has cost at least $\text{depth}_T(x)$: The pointer must be moved from the root to x, visiting all vertices in $\text{Path}_T(x)$ at some point. Thus, the cost dominates the number of queries that would be necessary for finding x.

Third, an online algorithm may need to maintain some data at each node in the tree; consider, for example, the *AVL tree* and *red-black tree* BST algorithms [CLRS01]. There are no explicit provisions for this in our online model; however, amending it accordingly is easy. The algorithms studied in this thesis work purely on the search tree structure and do not require additional data.

Fourth, observe that BSTs are easily implemented in a way such that each *step* in a serve sequence, rotations in particular, can be executed in constant time. It is not quite obvious how to implement a general dynamic search tree data structure with this feature. The main problem is each node may have an unbounded number of children, and a rotation may transfer any number of them from one node to another. For trees, we observed in section 2.2.2 that only one child can switch parents, but it is still not immediately clear how to identify that child.

In chapter 10, we show how to implement *k-cut* search trees with constant-time rotations. In particular, we give a compact and efficient implementation of *2-cut* search trees on *trees.*

The author does not know whether it is possible to implement rotations in general search trees in constant time. Note that if the underlying graph has bounded degree, then each node has a bounded number of children. This means a more brute-force approach may be possible, but we still need to identify the correct children to transfer based on the structure of the graph.

[1]This extra "virtual" step makes the cost of every access positive, which will be useful. It can be interpreted as the time needed to output the found node.

Fifth, if we implement a data structure for searching, it obviously needs to be *searchable*. Let us focus on the tree case, and recall that a query to a vertex v with input x reveals whether $x = v$, or otherwise returns the first edge on the path from v to x.[2]

In a *static* search tree T, for each vertex v, we can simply map edges incident to v in G to children of v in T. Thus, we can find x in T with a sequence of $\mathrm{depth}_T(x)$ queries, moving down from the root, with only $\mathcal{O}(1)$ time overhead per query. In a *dynamic* search tree, this approach means that the mapping has to be maintained under rotations. We sketch a way to do this in section 10.10.

Note that an STG implementation does not need to be searchable if it is only used as a *decomposition* of the underlying tree. This is the case for our main practical application (see chapter 10). Here, it is actually an advantage of our model that it disregards the details of vertex search.

The prefix model. Above we discussed some caveats of our models, in particular the online model. Even for BSTs, there are several reasonable models with subtle differences; we refer to Kozma [Koz16, section 2.2] for more details.

There is one *offline* model which is worth studying in more detail. In the BST setting, it is equivalent to our model, but this equivalency may not hold for general graphs.

prefix model

The *prefix model* is defined as our model above, except that *serving* a node x means choosing a prefix P of T that contains x, and replacing P with an arbitrary rooted tree on $V(P)$ such that the result is a valid search tree on G. The cost to serve x is taken as the number of vertices in P. Define $\mathrm{DynOPT}'(G, T, X)$ to be the total cost of serving an input sequence X with initial tree T in the prefix model.

Let us call the first model the *pointer model*. As mentioned above, it is known that the two models are essentially equivalent for BSTs, i.e., when G is a path. To see this, observe that when serving x_i in the pointer model, the nodes touched by the pointer form a prefix P of T. All rotations take place within P, and to visit x_i, we must have $x_i \in V(P)$. The number of pointer moves is clearly at least $|V(P)| - 1$, so the cost of serving x_i in the pointer model is at least $|V(P)|$. Thus, we have $\mathrm{DynOPT}'(G, T, X) \leq \mathrm{DynOPT}(G, T, X)$.

The above argument actually holds for arbitrary underlying graphs. For the other direction, we need the following fact.

Observation 8.1. *Any BST on n elements can be transformed into any other BST in the pointer model using $\mathcal{O}(n)$ pointer-moves and rotations at the pointer.*

Observation 8.1 is well-known, and we will discuss it and generalizations to STGs in part III. Now, say a prefix-model algorithm serves x_i by touching a prefix P and transforming it to P'. In the pointer model, we can touch all elements of P with a simple traversal with $2|V(P)|$ pointer moves. Further, observation 8.1 lets us transform P into P' with $\mathcal{O}(|V(P)|)$ steps. Thus, $\mathrm{DynOPT}(G, T, X) \in \mathcal{O}(\mathrm{DynOPT}'(G, T, X))$.

This argument unfortunately does not work in general, even for trees, for two reasons. First, observation 8.1 does not hold for arbitrary trees [CLPL18]. Second, transforming the prefix P into P' involves search trees on $\mathrm{torso}_G(V(P))$ (see section 2.2.6), which may be an even more general graph than a tree.

[2]Recall that vertex queries in general graphs are harder to define; see section 1.1.

We partially solve the first problem in chapter 13 by showing that observation 8.1 holds for arbitrary trees when restricted to *k-cut search trees.* Note that most of the algorithms presented in this thesis operate on k-cut search trees. Still, the question whether the two models are equivalent even in this restricted case is left open.

Open question 8.1. Are the pointer model and the prefix model equivalent when restricted to k-cut search trees on trees?

Dynamic vs. static search trees. Every algorithm that computes a static search tree can be seen as a simple offline dynamic algorithm: Just output the computed search tree as the initial tree, and then serve the each access without any rotations, by moving the pointer down the root path. The cost of this algorithm for an access sequence X is clearly $\text{cost}(T, w_X)$, where $w_X \colon V(G) \to \mathbb{N}_0$ is the function counting the number of occurrences of each node in X. Since we are free to compute the optimal static search tree, even if it takes exponential time, we have:

Theorem 8.2. *Let G be a graph and let $X = (x_1, x_2, \ldots, x_m) \in V(G)^m$ be an access sequence. Then* $\text{DynOPT}(G, X) \leq \text{StOPT}(G, w_X)$*, where $w_X(v) \in \mathbb{N}_0$ is the number of times v occurs in X.*

8.1. Adaptive bounds for STGs

The holy grail of dynamic BSTs (and, by extension, dynamic STGs) is two find an *online* algorithm that matches the running time of the *offline* optimum. This is known as the *dynamic optimality conjecture.*

We start by discussing the BST case. The two online algorithms generally considered the most likely candidates for dynamic optimality are Splay [ST85b] and Greedy [DHI$^+$09]. Splay is among the oldest dynamic BST algorithms and is based on a relatively simple procedure that moves the accessed node to the root for quick future access. We extensively discuss generalizations of Splay to the STG setting in chapter 9. Greedy is an online variant of the *offline* GreedyFuture algorithm [Luc88, Mun00], which transforms the search path of the accessed node into a search tree that optimizes future searches.

While the dynamic optimality conjecture remains unsolved, some dynamic BST algorithms (including Splay and Greedy) are known to *adapt* well to certain input sequences, i.e., they efficiently serve sequences that are *easy* in some well-defined way. On the other hand, there also exist *hard* sequences which are costly for any algorithm to serve [Wil89].

Adaptivity is expressed with a set of *upper bounds* on $\text{DynOPT}(G, X)$, depending on G and X. In the following, we list a few of these bounds (following the survey of Chalermsook, Goswami, Kozma, Mehlhorn, and Saranurak [CGK$^+$16]) and discuss ways of generalizing them to STGs.

STG bound

Formally, an *STG bound* is a function $f(G, X)$ depending on a graph G and an access sequence $X \in V(G)^m$. In the following, we consider an STG bound to be *achieved* by an algorithm if its cost for X is $\mathcal{O}(f(G, X) + g(G))$ for any function g. The additive term is necessary for online algorithms.

The BST variants of all bounds presented below are achieved by both Splay and Greedy [CGK$^+$16].

Static optimality. The *static optimality* bound is $\mathrm{StOPT}(G, w_X)$, where w_X is the function that counts the number of occurrences of each node in X. We already showed that the dynamic optimum achieves static optimality (theorem 8.2). The main result of chapter 9 is a Splay-based online algorithm for search trees on *trees*, called SplayTT, that we show to achieve static optimality.

static optimality

For BSTs, the static optimum cost $\mathrm{StOPT}(G, w)$ is asymptotically equal to the *entropy* of the input sequence, defined as

$$\sum_{v \in V(G)} w(v) \cdot \log(m/w(V)).$$

This quantity is known as the *entropy bound* [CGK+16]. In contrast to the BST setting, the entropy bound cannot be achieved in the general STG case, even for offline algorithms (see section 8.3). However, if G is a tree, the entropy bound is strictly weaker than static optimality (see section 4.4).

entropy bound

Finally, we note that static optimality implies an optimal worst-case $\mathcal{O}(m \log n)$ bound in the BST case, and a worst-case $\mathcal{O}(m \cdot \mathrm{td}(G))$ bound in general.

Working set. Consider an access sequence $X = (x_1, x_2, \ldots, x_m)$. For each $i \in [m]$, we say the *working set* W_i of i is the set of distinct accesses since the last access to x_i (or since the beginning, if x_i was not accessed before). More formally, the working set is the set of elements contained in the maximal contiguous subsequence of X ending at index i that contains x_i no more than once. The *working-set* bound for X is $\sum_{i=1}^{m} \log |W_i|$.

working set

This definition is easily generalized to STGs, but cannot be achieved in general (see section 8.3). If G is a tree, we show that our Splay-based algorithms do achieve this bound (see section 9.3), by an extension of the respective argument for the Splay BST algorithm.

We propose two variants of the working-set bound. The *linear working-set* bound is $\sum_{i=1}^{m} |W_i|$. This property is achieved by a simple algorithm (see section 8.2) and is tight for cliques (see section 8.3).

linear working set

Even the classical (logarithmic) working set bound seems very loose if the underlying graph is a tree. It would be interesting if we can somehow use the "topology" of the working set W_i in the graph to formulate a stronger property. We propose the following.

The *torso-tree-depth working-set* bound is $\sum_{i=1}^{m} \mathrm{td}(\mathrm{torso}_G(W_i))$.[3] Observe that the torso-tree-depth working-set property is equivalent to the logarithmic working set bound for paths, and can be stronger on trees (e.g., the bound is $\mathcal{O}(1)$ if $\mathrm{torso}_G(W_i)$ is a star).

torso-tree-depth working set

We now sketch our main motivation for the torso-tree-depth working-set bound. Say we access x, then a set U of vertices, and then x again. Let T be the search tree just before the last access. Suppose U induces a prefix of T and x is a child of some $u \in U$ (this holds or "almost" holds for many online algorithms). Recall that such a prefix must be a search tree on $\mathrm{torso}_G(U)$ (theorem 2.21). To achieve the torso-tree-depth working-set bound, this prefix must have (approximately) *optimal height*.

Conjecture 8.2. The offline optimum achieves the torso-tree-depth working-set bound.

[3]See section 2.2.6 for the definition of $\mathrm{torso}_G(W_i)$.

The working-set bound and its variants are primarily concerned with *locality in time*, i.e., repeating requests shortly after each other can be served quickly. The next two bounds conversely are about *locality in the search space.*

static finger

Static finger. Fix a graph G and let $d(u, v)$ denote the distance between u and v in G. The *fixed finger* bound w.r.t. a *finger vertex* $v \in V(G)$ for an access sequence $X = (x_1, x_2, \ldots, x_m)$ is $\sum_{i=1}^{m} \log d(x_i, v)$. The *static finger* bound is the minimum fixed-finger bound over all possible finger vertices.

For BSTs, achieving the static finger bound follows from achieving static optimality in relatively straight-forward way: It is not hard to construct a static search tree for each $v \in V(G)$ where the depth of each node u is $\mathcal{O}(\log d(v, u))$. An algorithm with static optimality matches the cost of all these trees, and thus achieves the static-finger bound.

For STGs in general, the static-finger bound as stated is clearly unachievable. For example, in a clique, all distances are one, so the static-finger bound is zero, while $\mathrm{DynOPT}(G, X)$ can be as large as $\Theta(mn)$ (see section 8.3). Interestingly, the bound is unachievable already for trees. Indeed, if G is a binary tree, then the static finger bound is $\mathcal{O}(m \log\log n)$, since the diameter of G is $\mathcal{O}(\log n)$. We will show in chapter 12 (proposition 12.38) that there exist sequences X with $\mathrm{DynOPT}(G, X) \in \Omega(m \log n)$, so no algorithm can match the static-finger bound (even offline).

One intuitive way to understand the problem is that a binary tree G has asymptotically equal tree-depth and diameter (both are $\Theta(\log n)$), whereas if G is a path, then the tree-depth is logarithmic in the diameter. This suggests that the tree-depth should play a role in an improved variant of the static finger bound.

linear static finger

But first, let us define the *linear static-finger* bound (analogously to the linear working-set bound), as $\min_v \sum_{i=1}^{m} d(x_i, v)$. If G is a clique, the linear static-finger bound is m and again cannot be achieved. However, for trees, it trivially implied by static optimality: For each $v \in V(G)$, rooting G at v yields a tree T with $\mathrm{cost}(T, w_X) \le \sum_{i=1}^{m} d(x_i, v)$.

We now define a tree-depth-based generalization for the case when G is a tree. For each vertex pair $u, v \in V(G)$, let $C_{u,v}$ be the unique component of $G - \{u, v\}$ that is adjacent to both u and v (or the empty graph, if u and v are themselves adjacent). Let the *tree-depth fixed-finger* bound w.r.t. a finger vertex $v \in V(G)$ be $\sum_{i=1}^{m}(1 + \mathrm{td}(C_{x_i,v}))$. (The tree-depth of the empty graph is zero.) The *tree-depth static-finger* bound is the minimum tree-depth fixed-finger bound over all possible finger vertices.

tree-depth static finger

Observe that if G is a path, then the tree-depth static-finger bound is asymptotically equal to the (logarithmic) static-finger bound. Perhaps the tree-depth static-finger bound can be achieved in the same way as the static-finger bound, via a collection of static trees.

Conjecture 8.3. For each graph G, input sequence X, and finger vertex v, there exists a static search tree T on G that achieves the tree-depth fixed-finger bound w.r.t. v.

If conjecture 8.3 is true, then every algorithm with static optimality (such as our SplayTT) also achieves the tree-depth static-finger bound.

dynamic finger

Dynamic finger. The *dynamic-finger* BST bound is a variant of fixed/static finger where the “finger” always moves to the last accessed elements. Formally, the dynamic-finger bound is $\sum_{i=2}^{m} \log d(x_{i-1}, x_i)$. It is perhaps a more appropriate measure of *space locality* then the static-finger bound.

The dynamic-finger bound, as stated, is unachievable for general STGs, or even STTs, for the same reasons as the static finger bound. The *linear dynamic-finger* and *tree-depth dynamic-finger* bounds can be derived analogously.

The linear dynamic-finger bound is achieved by a simple *STT* algorithm presented in section 8.2. Since the BST dynamic-finger bound is already very hard to prove for Splay [CMSS00, Col00], we do not attempt to prove the tree-depth dynamic-finger bound for our algorithms.

Lower bounds. Besides the upper bounds described above, there are several known *lower* bounds on DynOPT(G, X) [Wil89, DHI+09]. Here, by "lower bound" we mean a function $f(G, X)$ such that DynOPT$(G, X) \in \Omega(f(G, X))$ for each connected graph G and each access sequence $X \in V(G)^m$.

We focus on a bound due to Wilber [Wil89] that is commonly called the *interleave* lower bound [Koz16]. Denote it by $\Lambda(G, X)$. The interleave bound has been prominently used as an ingredient of the Tango algorithm, which has amortized running time $\mathcal{O}(\Lambda(G, X) \cdot \log\log n)$, and thus is $\mathcal{O}(\log\log n)$ competitive with the dynamic optimum [DHIP07]. Tango was the first algorithm to achieve a competitive ratio better than the trivial $\mathcal{O}(\log n)$.

The interleave lower bound and Tango were generalized to STTs by Bose, Cardinal, Iacono, Koumoutsos, and Langerman [BCI+20]. We define it in part III of the thesis (section 12.2) and show that it holds in a much more general setting, where it serves as a lower bound for the *rotation distance* between search trees.

Summary. It turns out that generalizing BST bounds to the STT and STG settings can be quite subtle. Above, we suggested some possibilities. In the rest of this chapter, we will study some of the simpler STG bounds and see whether they can be achieved or not.

In chapter 9, we present and analyze a generalization of Splay to the tree case, called SplayTT. We will show that SplayTT achieves static optimality and the logarithmic working set bound. Another promising approach to studying STG bounds would be a generalization of the Greedy BST algorithm.

Open question 8.4. Is there a generalization of Greedy or GreedyFuture to the STT or STG setting? Which bounds can be achieved by such a generalization?

8.2. Simple dynamic algorithms

The arguably simplest adaptive BST algorithm is MoveToRoot. After finding a node v, this algorithm repeatedly rotates v with its parent until v is the root. Each such rotation reduces the depth of v by one, so v indeed ends up at the root.

It is well-known that for MoveToRoot, there is a request sequence that forces a linear number of rotations per request [Koz16]. However, Allen and Munro [AM78] showed that MoveToRoot achieves a weaker form of static optimality: If requests are sampled randomly from an arbitrary distribution, then the cost of MoveToRoot is at most the cost of an optimal static search tree (up to a constant and after enough requests).

MoveToRoot easily generalizes to search trees on arbitrary graphs (observe that a rotation at x always reduces the depth of x by one). Note that in our model, an algorithm has to choose an initial tree; since there is no obvious choice for MoveToRoot, we prove the following results for any initial tree.

We first show that MoveToRoot achieves the linear working-set bound described in section 8.1.

Proposition 8.3. *For each graph G with n vertices and each input sequence $X = (x_1, x_2, \dots, x_m)$,* MoveToRoot *serves X with cost at most*

$$\mathcal{O}\left(\operatorname{cost}(T^0, \mathbb{1}) + \sum_{i=1}^{m} |W_i|\right),$$

where T^0 is the initial tree and W_i is the working set at time i.

Proof. Let T be the search tree before some access to $v \in V(G)$, and let W_v be the current working set of v. The crucial observation is that a node u can only become an ancestor of another node when u is accessed. From this, it is easy to see that

- If v has been accessed before, then all ancestors of v in T have been accessed since the last access to v (i.e., are in W_v).
- Otherwise, all ancestors of v in T have been either accessed before, or are ancestors of v in T^0.

This implies that the access cost for v is $\operatorname{depth}_{T^0}(v) + |W_v| + 1$ on the first access, and $|W_v| + 1$ on each further access. Hence, the linear working-set bound is achieved up to an additive term of $\sum_{v \in V(G)} \operatorname{depth}_{T^0}(v) = \operatorname{cost}(T^0, \mathbb{1})$. □

Recall that MoveToRoot on paths achieves static optimality in a random setting [AM78]. We now show that this does *not* hold for MoveToRoot on general STGs, even when the underlying graph is a star. We use parts of Allen and Munro's [AM78] analysis.

Proposition 8.4. *The expected cost of* MoveToRoot *on a star with n vertices, for $m \geq n$ uniformly distributed requests, is $\Omega(m \cdot n)$.*

Proof. Let S_n be the star with central vertex c and $n - 1$ leaves. We consider a tree T after k requests.

Let u, v be two leaves of S_n. We lower bound the probability $p_{u,v}$ that u is an ancestor of v in T. Consider the last request made to u, v, or c. The probability that such a request has occurred at all is

$$1 - \left(\tfrac{n-3}{n}\right)^k = 1 - \left(1 - \tfrac{3}{n}\right)^k \geq 1 - e^{-3k/n}.$$

Assume such a request has occurred, then u is an ancestor of v in T if and only if the request was for u, which happens with probability $\frac{1}{3}$. (Recall the structure of search trees on graphs as described in section 2.2.5.) Hence, we have

$$p_{u,v} \geq \tfrac{1}{3}(1 - e^{-3k/n}).$$

Clearly, the expected depth of a node v in T is $\sum_{u \neq v} p_{u,v}$, and thus the expected cost of request $k+1$ is

$$\tfrac{1}{n} \sum_{v} \sum_{u \neq v} p_{u,v} = \tfrac{(n-1)(n-2)}{3n}(1 - e^{-3k/n}) \geq \tfrac{n}{12}(1 - e^{-3k/n}),$$

if $n \geq 4$. For all $k \geq \frac{1}{3}n$, this is at least $\frac{n}{12}(1 - \frac{1}{e}) \geq \frac{1}{24}n$, for a total cost of at least $(m - \frac{n}{3}) \cdot \frac{1}{24}n \geq \frac{2}{3}m \cdot \frac{1}{24}n \in \Omega(m \cdot n)$. □

Finally, we discuss a different algorithm that achieves the *linear dynamic-finger bound* (defined in section 8.1) when the underlying graph G is a tree. This algorithm starts with am arbitrary 1-cut search tree (i.e., a *rooting* of G). When accessing v, if v is not the root, we identify the unique child c of the root that is an ancestor of v, and rotate c with r. Observe that the result is a rooting of G at c. Repeat this until v becomes the root.

The number of rotations to bring v to the root is clearly the distance between v and the previous root r in G. In the online setting, the first access can take up to $\mathcal{O}(n)$ time. Hence, we obtain:

Proposition 8.5. *There exists an online dynamic STT algorithm that serves an access sequence $(x_1, x_2, \ldots, x_m) \in [n]^m$ with cost $\sum_{i=2}^{m} d(x_{i-1}, x_i) + \mathcal{O}(n)$, where $d(\,\cdot\,,\cdot\,)$ denotes the distance between two vertices in the underlying tree G.*

In other words, the algorithm achieves the linear dynamic-finger bound.

8.3. List-update and dynamic search trees on cliques

We now consider dynamic search trees on *cliques*, which serve as a counterexample to some of the bounds discussed above.

Recall that search trees on cliques are always degenerate (section 2.2.5), and a rotation simply swaps two elements. These search trees can also be interpreted as *linked lists*. *Searching* the list means performing a linear search from the start, and rotations swap two adjacent elements in the list. This is essentially the *list-update model* of Sleator and Tarjan [ST85a].[4]

MoveToRoot on cliques corresponds to Sleator and Tarjan's MoveToFront list-update algorithm, which is known to be dynamically optimal. Thus, the dynamic optimality conjecture is true for cliques.

It is easy to see that the cost of MoveToFront is at least $\binom{n}{2}$ if X is a permutation of $V(G)$, i.e., if every vertex is accessed precisely once. This means that also $\mathrm{DynOPT}(G, X) \in \Theta(n^2)$. Observe that the entropy bound, the logarithmic working-set bound, and the various logarithmic finger bounds are all $\mathcal{O}(n \log n)$ for a permutation. Thus, these bounds cannot be achieved in the general STG setting.

[4]In the list-update model, certain rotations/swaps are free, but this does not affect the asymptotic running time.

9. Splay trees on trees

In this section, we discuss a generalization of Splay to search trees on trees. In fact, we discuss multiple variants based on the same idea. Note that the underlying graph will always be a *tree* in this chapter.

The algorithms in this chapter are restricted to *2-cut STTs*. This has two advantages. First, 2-cut STTs turn out to *locally* behave much like BSTs; we exploit this in our Splay generalization. Second, 2-cut STTs can be efficiently implemented, and thus are well-suited in practical applications (see chapter 10).

We start with some useful definitions and observations related to 2-cut STTs.

Separator nodes and partitions. Let T be a 2-cut search tree on a tree G, and let v be a node in T. We call v a *separator node* if $|\partial(T_v)| = 2$. Observe that $|\partial(T_v)| \leq 2$ by definition, so if v is *not* a separator node, then T_v has boundary size zero or one.

separator node

Lemma 9.1. *Let v be a separator node in a 2-cut search tree on a tree G. Then v separates the two nodes $\{a, b\} = \partial(T_v)$.*

Proof. Consider the components $\mathbb{C}(G[T_v] - v)$. If every such component is adjacent to at most one node of a and b, then v separates a and b. Otherwise, there is a component C with $\partial(C) = \{a, b, v\}$. But there is a child c of v with $V(T_c) = V(C)$, so T is not 2-cut. □

We now present a simple tool that will be useful when dealing with separator nodes. Fix a tree G and two distinct vertices $x, y \in V(G)$, and let $\mathcal{C} = \mathbb{C}(G - \{x, y\})$. The *$(x, y)$-partition* of G is the triple (U_x, U', U_y) of vertex sets where:

(x, y)-partition

- U_x consist of x and each component $C \in \mathcal{C}$ adjacent to x, but not y.
- U' consist the unique component $C \in \mathcal{C}$ adjacent to both x and y.
- U_x consist of y and each component $C \in \mathcal{C}$ adjacent to y, but not x.

Clearly, if (U_x, U', U_y) is the (x, y)-partition of G, then the reverse triple (U_y, U', U_x) is the (y, x)-partition of G.

Lemma 9.2. *Let T be 2-cut STT and let p be the parent of v. Let (U_p, U', U_v) be the (p, v)-partition of $G[T_p]$. Then no vertex in $\delta(T_p)$ is adjacent to U', and at most one vertex in $\delta(T_p)$ is adjacent to U_v.*

Proof. See figure 9.1 for an illustration. First, if U' is nonempty, then there exists a child c of v with $U' = V(T_c)$. Since $\{p, v\} \subseteq \partial(T_c)$ and T is 2-cut, no further vertices are contained in $\partial(T_c)$, so only p and v are adjacent to U'.

Second, observe that $V(T_v) = U' \cup U_v$ and $p \in \partial(T_v) = \partial(U' \cup U_v)$. By the same argument as above, at most one vertex outside of $V(T_p)$ can be adjacent to $U' \cup U_v$, and hence to U_v. □

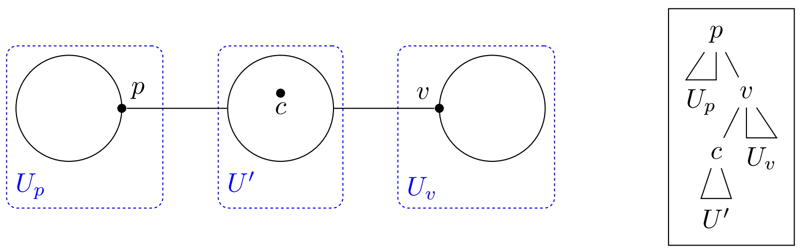


Figure 9.1.: (p, v)-partition of $G[T_p]$ (left) and the search tree (right) for lemma 9.2.

Allowed rotations. It is easy to see that some rotations can destroy the 2-cut property. We now precisely characterize the allowed rotations.

Lemma 9.3. *Let T be an STT and let $v \in V(G)$ with parent $p \in V(G)$. Rotating v with p maintains the 2-cut property if and only if v is a separator node or p is not a separator node.*

Proof. Let (U_p, U', U_v) be the (p, v)-partition of $G[T_p]$. By lemma 2.10, we have $V(T'_v) = V(T_p)$ and $V(T'_p) = U_p \cup U'$. Clearly, this means that $|\partial(T'_v)| \leq 2$. For all nodes $x \in V(T) \setminus \{v, p\}$, we have $V(T_x) = V(T'_x)$. Thus, the only node that may not be 2-cut in T' is p.

Suppose first that p is not a separator node in T. Then $|\partial(T'_p)| \leq |\partial(T'_v)| + 1 = |\partial(T_p)| + 1 \leq 2$ by observation 2.7, so the rotation produces a 2-cut search tree.

Now suppose that p is a separator node in T with $\{a, b\} = \partial(T_p)$. We claim that then T' is 2-cut if and only if v is a separator node, as required. By lemma 9.2, there are two cases (up to symmetry):

- If a and b are both adjacent to U_p, then we have $\partial(T_v) = \partial(U' \cup U_v) = \{p\}$, so v is not a separator. Further, we have $\partial(T'_p) = \partial(U_p \cup U') = \{a, b, v\}$, so T' is not 2-cut.
- If a is adjacent to U_p and b is adjacent to U_v, then $\partial(T_v) = \{p, b\}$. This means that v is a separator in T, and $\partial(T'_p) = \partial(U_p \cup U') = \{a, v\}$, so T' is 2-cut. □

(in)direct separator node

Direct and indirect separator nodes. Let T be a search tree on a tree G, and let v be a separator node with parent p and grandparent g. We call v a *direct separator node* if $\partial(T_v) = \{p, g\}$, or an *indirect separator node* otherwise.

Observe that if we rotate a node v with its parent p, by definition (see section 2.2.2), precisely the direct separator children of v change parent from v to p. We already observed that at most one such child can exist. We now prove uniqueness for both direct and indirect separator children, starting with a technical lemma.

Lemma 9.4. *Let p be a node in a search tree T on a tree G, and let u, v be children of p. Then $\partial(T_u) \cap \partial(T_v) = \{p\}$.*

Proof. By observation 2.7, we have $p \in \partial(T_u) \cap \partial(T_v)$. Suppose $a \in \partial(T_u) \cap \partial(T_v)$, for some vertex $a \neq p$. Clearly, $U = V(T_u) \cup \{p\} \cup V(T_v)$ induces a connected subgraph of G. Moreover, a is adjacent to both $V(T_u)$ and $V(T_v)$, so there are two edges from a to U. But then G is has a cycle, a contradiction. □

Lemma 9.5. *Each node in a 2-cut STT has at most one child that is a direct separator node, and at most one child that is an indirect separator node.*

Proof. Suppose a node u has two direct separator children v, v'. Then u has a parent p and $\partial(T_v) = \partial(T_{v'}) = \{u, p\}$. But $\partial(T_v) \cap \partial(T_{v'}) = \{u\}$ by lemma 9.4, a contradiction.

Now suppose u has two indirect separator children v, v'. Then u has a parent p, and there are distinct ancestors a, b of p such that $\partial(T_v) = \{a, u\}$ and $\partial(T_{v'}) = \{b, u\}$ by lemma 9.4. But then $\{p, a, b\} \subseteq \partial(T_u)$ contradicting that T is 2-cut. □

2-cut rotations vs. BST rotations. As mentioned above, intuitively, 2-cut STTs "locally" behave like BSTs (i.e., search trees on *paths*), which allows applying familiar BST techniques. The following two lemmas make this idea precise.

Lemma 9.6. *Let v be a node in a 2-cut search tree T on a tree G. Let p be the parent of v and let $a \in \partial(T_p)$. Then v, p, a must lie on a common path in G (though not necessarily in that order).*

Proof. Let (U_p, U', U_v) be the (p, v)-partition of $G[T_p]$. Lemma 9.2 implies that a is adjacent to U_p or U_v, which means that v, p, a are on a common path. □

Lemma 9.7. *Let v be a non-root node in a 2-cut search tree T on a tree, and assume rotating v with its parent p yields a 2-cut search tree T'. Let C be the set of separator children of v and p in T. Then the nodes $\{v, p\} \cup \partial(T_p) \cup C$ must lie on a common path in G.*

Proof. Let (U_p, U', U_v) be the (p, v)-partition of $G[T_p]$. Using lemma 9.2 on (U_p, U', U_v) with T and on its reverse (U_v, U', U_p) with T', we get that U_p and U_v both are adjacent to at most one vertex in $\partial(T_p)$, and U' is adjacent to none. This implies that all vertices in $\partial(T_p)$ must be leaves of a path containing $\{v, p\} \cup \partial(T_p)$.

Now take some separator child c of v or p. Then $\partial(T_c) \subseteq \{v, p\} \cup \partial(T_p)$, so c separates two vertices from this set (lemma 9.1), and hence lies on the path. □

Using lemma 9.7 along with our earlier observations, we can characterize all allowed rotations and their effect on direct/indirect separators. See figure 9.2. We will not use this sketch in later proofs, but the reader may want to reference it for illustration.

Finally, we observe how a rotation changes node depths. The following is immediate from lemma 2.10.

Observation 9.8. *Let T be a 2-cut STT, let $v \in V(T)$, let p be the parent of v in T, and let T' be the search tree obtained by rotating v with p.*

Let $D = V(T_d)$ if v has a direct separator child d, and let $D = \emptyset$ otherwise. Then:

- $\mathrm{depth}_{T'}(u) = \mathrm{depth}_{T'}(u) + 1$ *for all $u \in V(T_p) \setminus V(T_v)$.*
- $\mathrm{depth}_{T'}(u) = \mathrm{depth}_{T'}(u) - 1$ *for all $u \in V(T_v) \setminus D$.*
- $\mathrm{depth}_{T'}(u) = \mathrm{depth}_{T'}(u)$ *for all remaining nodes.*

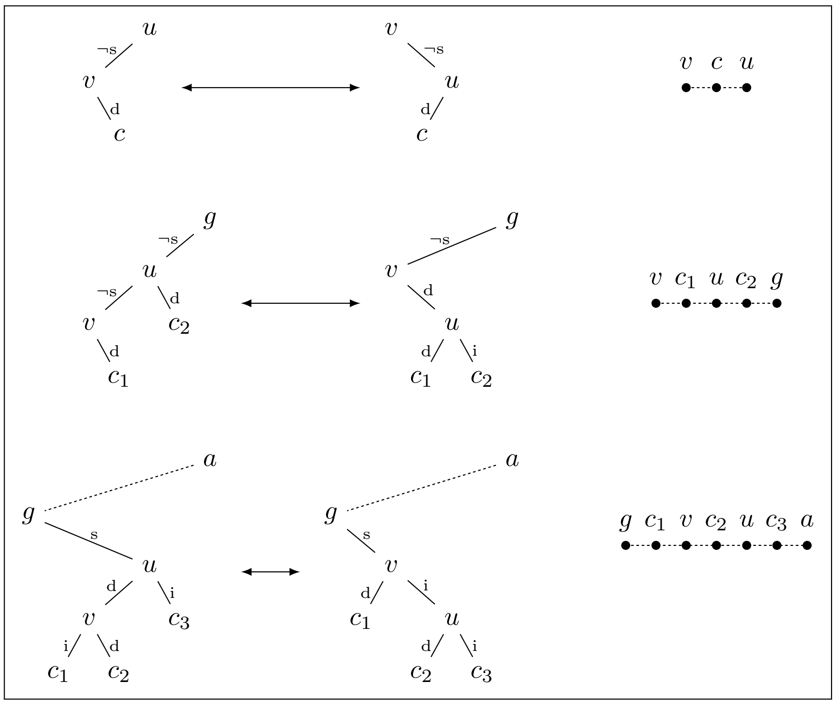


Figure 9.2.: All possible valid rotations in a 2-cut STT, up to symmetry. On the right, the order of vertices in the underlying tree is shown. A small "s" (resp. "d", "i") indicates an edge to a separator (resp. direct/indirect separator) child, and "¬s" (resp. "¬d", "¬i") indicates the child cannot have the respective property. Dotted edges represent an ancestor-descendant path of arbitrary length.

9.1. 2-cut Move-to-root

As a warm-up, we describe a variant of the MoveToRoot algorithm of section 8.2. The idea is very simple: To move v to the root, we keep trying to rotate v with its parent. If that is not possible (without destroying the 2-cut property), we instead rotate its parent with its grandparent. We call this algorithm MoveToRootTT; see algorithm 9.4 for pseudocode.

It is easy to see that this algorithm only performs allowed rotations. If a rotation of v with its parent p is not allowed, then v is not a separator and p is a separator by lemma 9.3. This implies that p is not the root and, in particular, we may rotate p with its parent (again by lemma 9.3). By observation 9.8, even when rotating at p, the depth of v is reduced by one (since v is not a separator). Thus, eventually v becomes the root.

Observe that lower bound of proposition 8.4 does not apply to MoveToRootTT. Indeed, if G is a star, then a 2-cut search tree T on G has height at most four. Otherwise, the center c of G would have at least three ancestors (see section 2.2.5), all of which are in $\partial(T_c)$. Thus, MoveToRootTT requires constant time per search on a star.

On the other hand, MoveToRootTT on a path is exactly the same as the BST MoveToRoot algorithm, since then all rotations are allowed. Hence, MoveToRootTT does require $\Theta(n)$ time per access in the worst case.

Experimental analysis indicates that MoveToRootTT performs well for uniformly distributed searches (see section 10.9). It would be interesting to study whether the randomized static optimality property of the original BST MoveToRoot algorithm [AM78] holds for MoveToRootTT.

Open question 9.1. Let G be a tree, and let $X \in V(G)^m$ be a sequence of $m \geq |V(G)|$ searches that are sampled independently from a distribution p on $V(G)$. Does MoveToRootTT serve X in expected running time $\mathcal{O}(\mathrm{StOPT}(G,p) + f(G))$, for some function f?

9.2. Splaying on 2-cut STTs

We now turn to our generalization of the Splay algorithm to 2-cut STTs.

We first describe Sleator and Tarjan's [ST85b] original Splay algorithm for BSTs. It can be seen as a slightly more sophisticated version of the MoveToRoot algorithm. After finding a node v, it is brought to the root by a series of calls to the procedure

Algorithm 9.4 The 2-cut variant of the MoveToRoot algorithm.

```
1: procedure MoveToRootTT($v$)
2:     while $v$ has parent $p$ do
3:         if can_rotate($v$) then
4:             rotate($v$)
5:         else
6:             rotate($p$)
```

`splay_step`(v) (see figure 9.3). If v has no grandparent, then `splay_step`(v) simply rotates v with its parent p (this is called a ZIG step). If the value of v is between the values of p and its grandparent g, then `splay_step`(v) rotates twice at v (ZIG-ZAG step). Finally, if the value of v is smaller or larger than both values of p and g, then `splay_step`(v) rotates first at p and then at v (ZIG-ZIG step). Afterwards, v is an ancestor of both p and g, so v is eventually brought to the root.

It is far from obvious that Splay performs well. We described many of its adaptivity properties in section 8.1, and direct a reader interested in a more in-depth discussion of Splay to the relevant literature [ST85b, CGK+15, Koz16, LT19]. For us, the most important properties of Splay are *static optimality* and the *working-set bound*; we will show that both also hold for our generalization.

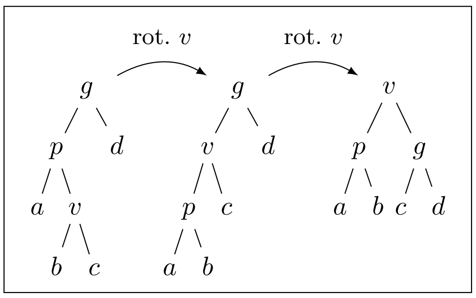
In 2-cut STTs, `splay_step`(v) can be applied basically as-is, since lemma 9.6 implies that v, p, and g are on a path. If v is between p and g, then we execute a ZIG-ZAG step; otherwise, we execute a ZIG-ZIG step (see algorithm 9.5). However, simply applying `splay_step` repeatedly may destroy the 2-cut property, which may make it impossible to apply `splay_step` later. Thus, our algorithms need to be more careful. The following lemma characterizes situations where `splay_step`(v) is allowed; see figure 9.4 for a sketch of allowed `splay_step`s.

Lemma 9.9. *Let v be a node in a 2-cut search tree T on a tree. If v is a child of the root of T, then* `splay_step`(v) *preserves the 2-cut property. If v has a parent p and a grandparent g, then* `splay_step`(v) *preserves the 2-cut property if and only if g is not a separator or both v and p are separators.*

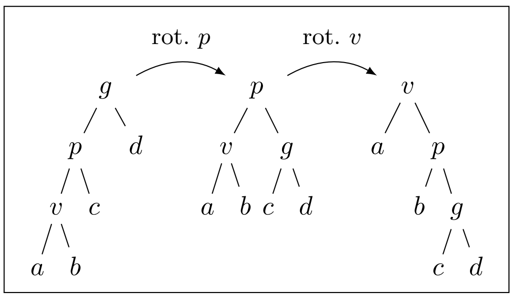
Proof. If v is the child of the root r, then `splay_step`(v) performs a single rotation, which is allowed (i.e., preserves the 2-cut property) by lemma 9.3, since r trivially is not a separator.

Suppose v is not the child of the root, and we apply `splay_step`(v). Let T' be the search tree after the first rotation, and T'' be the search tree after the second rotation.

If `splay_step`(v) executes a ZIG-ZIG step, then it first rotates p with g, and then v with p. The first rotation is disallowed iff $|\partial(T_p)| = 1$ and $|\partial(T_g)| = 2$. The second rotation is disallowed iff $|\partial(T'_v)| = |\partial(T_v)| = 1$ and $|\partial(T'_p)| = |\partial(T_g)| = 2$. So the ZIG-ZIG step is disallowed if g is a separator in T and at least one of v and p is not. This is precisely the negation of the condition stated in the lemma.



(a) ZIG-ZAG step

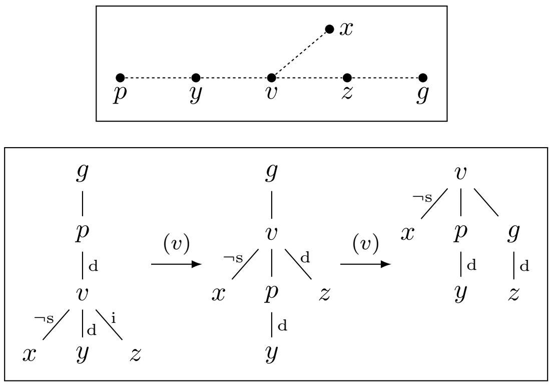


(b) ZIG-ZIG step

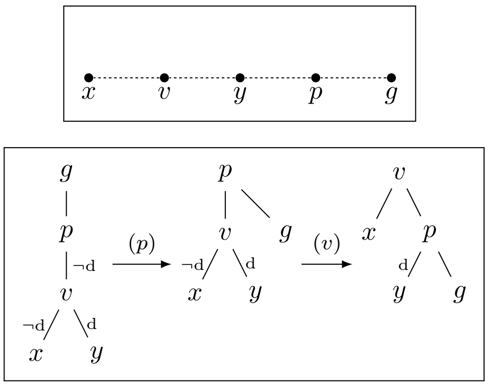
Figure 9.3.: `splay_step` in binary search trees.

Algorithm 9.5 The `splay_step` procedure.

```
Input: Non-root node $v$
procedure splay_step($v$)
    if $v$ has no grandparent then                ▷ ZIG
        rotate($v$)
    else
        if $v$ is a direct separator then         ▷ ZIG-ZAG
            rotate($v$)
            rotate($v$)
        else                                      ▷ ZIG-ZIG
            rotate(parent($v$))
            rotate($v$)
```



(a) ZIG-ZAG (rotate twice at v)



(b) ZIG-ZIG (rotate at p, then v)

Figure 9.4.: Sketches of the two cases in a `splay_step`(v), including the behavior of all possible types of children of v. Edge labels mean the same as in figure 9.2. The relevant part of the underlying graph is shown at the top.

If `splay_step`(v) executes a ZIG-ZAG step, then it rotates twice at v. The first rotation is disallowed if and only if $|\partial(T_v)| = 1$ and $|\partial(T_p)| = 2$. This can never happen, since we only execute a ZIG-ZAG step if v is a separator. The second rotation is disallowed if and only if $|\partial(T'_v)| = |\partial(T_p)| = 1$ and $|\partial(T'_g)| = |\partial(T_g)| = 2$. Since v is a separator in T, this is again the negation of the stated condition. □

In the following, we present two different adaptations of the Splay algorithm to 2-cut STTs. In both cases, we will show correctness of the algorithm immediately. The cost analysis is similar for both variants and is deferred to section 9.3.

Greedy SplayTT. Our first algorithm is similar to MoveToRootTT and is inspired by a *dynamic forest* data structure of Holm, Rotenberg, and Ryhl [HRR23] (see chapter 10 for more information on the dynamic forest problem). GreedySplayTT brings a node v to the root by repeatedly trying to execute `splay_step` on v, its parent, and its grandparent, in that order. See algorithm 9.6 for pseudocode. The function `can_splay_step` checks the conditions of lemma 9.9.

The following lemma implies that GreedySplayTT is always allowed to execute at least one of the three possible `splay_step`s.

Lemma 9.10. *Let v be a node in a 2-cut STT T with parent p and grandparent g. Then one of* `splay_step`(v)*,* `splay_step`(p)*, and* `splay_step`(g) *can be executed while maintaining the 2-cut property.*

Proof. Suppose the first two calls are disallowed. We use lemma 9.9 to show that then the third call is allowed. Observe that g must have a parent h; otherwise, g is not a separator, so `splay_step`(v) is allowed. Since `splay_step`(v) and `splay_step`(p) are disallowed, the nodes g and h must be separators, so `splay_step`(g) is allowed. □

It remains to show that GreedySplayTT actually brings the given node v to the root. Observe that we only execute `splay_step`(x) for ancestors x of v (including v itself). We show that a ZIG-ZIG or ZIG-ZAG step at x decreases the depth of each descendant of x

Algorithm 9.6 GreedySplayTT

```
procedure GreedySplayTT($v$)
    while $v$ has parent $p$ do
        if $p$ has parent $g$ then
            if can_splay_step($v$) then
                splay_step($v$)
            else if can_splay_step($p$) then
                splay_step($p$)
            else
                splay_step($g$)                ▷ Must be possible
        else                                   ▷ $p$ is the root
            rotate($v$)
```

(including x itself) by at least one in the following. Note that observation 9.8 will be used extensively.

Lemma 9.11. *Let x be a node in a 2-cut search tree T on a tree, and let T' be the search tree produced by executing* `splay_step`(x)*. Assume that* $\mathrm{depth}_T(x) \geq 3$ *and T' is 2-cut. Then, for each $u \in V(T_x)$, we have* $\mathrm{depth}_{T'}(u) < \mathrm{depth}_T(u)$.

Proof. Let p, g be the parent and grandparent of x. If x is not a direct separator, we execute a ZIG-ZIG step. The first rotation at p reduces the depth of all nodes in $V(T_x)$ (since x is not a direct separator), and all those nodes stay descendants of x. The following rotation at x cannot increase the depth of any descendant of x.

Now suppose x is a direct separator, so we do a ZIG-ZAG step. Let T'' be the search tree after the first rotation at x. We claim that p is not a direct separator in T''. Indeed, if it is, then p separates x and g in T. But x already separates p and g by assumption, a contradiction.

Let $D = V(T_d)$ if x has a direct separator child d in T, or let $D = \emptyset$ otherwise. The first rotation reduces the depth of each node in $V(T_x) \setminus D$, and these nodes stay descendants of x, so the second rotation does not increase their depth. On the other hand, all nodes in D become descendants of p in T''. The second rotation at x decreases the depth of p and all its descendants (since p is not a direct separator in T''). Thus, the ZIG-ZAG step also reduces the depth of each node in D. $\square$

We now argue that GreedySplayTT(v) executes at most one ZIG step, which must be the last `splay_step` and brings v to the root. Suppose otherwise we execute a ZIG step at the parent p or grandparent g of v. Then the grandparent of v is not a separator, so GreedySplayTT would execute `splay_step`(v) instead, a contradiction.

Combined with lemma 9.11, we have that every `splay_step`(x) in GreedySplayTT(v) decreases the depth of v by at least one, eventually bringing it to the root.

Two-pass SplayTT. We now turn to a different SplayTT variant. The algorithm itself is more complicated, but will be easier to analyze. It has some similarities to another dynamic forest data structure, namely *link-cut trees* [ST83, ST85b] (see also section 10.7), but was developed independently [BK22].

The idea is to first "clean up" the root path of v and remove all nodes that might inhibit rotations. Afterwards, all rotations on the root path are allowed, so we can simply repeat `splay_step`(v) until v is the root.

The algorithm uses the following "generalized splaying" procedure (see algorithm 9.7). Let x be a descendant of a node y. The procedure `splay_to`(x, y) executes `splay_step`(x) until y is the parent or grandparent of x. Then, if y is the grandparent of x, it executes a final rotation (or ZIG step), so x becomes a child of y.

We also want our procedure to be able to bring a node to the root. For this, we define the "parent of the root" to be $\perp$. Calling `splay_to`$(x, \perp)$ brings x to the root.

We now describe the TwoPassSplayTT algorithm. Fix a search tree T and a node $v \in V(T)$. Let $x \in \mathrm{Path}_T(v) \setminus \{v\}$, and let c be the unique child of x that lies on $\mathrm{Path}_T(v)$. We call x a *branching node for v* if $|\partial(T_x)| = 2$ and $|\partial(T_c)| = 1$.[1] Clearly, if x is a branching node for v, then it is a branching node for each descendant of v.

branching node for v

[1]The branching nodes are precisely the nodes on $\mathrm{ch}(\mathrm{Path}_T(v))$ that have degree three, hence the name.

Observe that a rotation at some node $x \in \mathrm{Path}_T(v)$ is allowed if and only if $\mathrm{parent}(x)$ is *not* a branching node. Accordingly, our algorithm removes all branching nodes in a first pass, which works as follows. We first find all branching nodes $b_1, b_2, \dots, b_k$ on the path from v to the root, ordered such that $b_k \prec_T b_{k-1} \prec_T \dots \prec_T b_1$. We then call `splay_to`$(b_i, b_{i+1})$ for each $i \in [k-1]$ in order. Finally, call `splay_to`$(b_k, \bot)$. This concludes the first pass. The second pass simply brings v to the root with `splay_to`$(v, \bot)$. See algorithm 9.8 for pseudocode.

To show correctness, we mainly need to prove that no rotations that destroy the 2-cut property are performed. The first pass is covered by the following lemma.

Lemma 9.12. *Let T be a 2-cut STT, let $v, x \in V(T)$ and $y \in V(T) \cup \{\bot\}$ with $y \prec_T x \prec_T v$, and let $U = \{u \mid y \prec_T u \prec_T x\}$ be the set of nodes strictly between x and y on the root path of v.*

Suppose that x is a branching node for v and U contains no branching nodes for v. Then `splay_to`(x, y) *performs precisely $|U|$ rotations, all of which are allowed. If T' is the search tree produced by* `splay_to`(x, y)*, then* $\mathrm{parent}_{T'}(x) = y$ *and* $\mathrm{Path}_{T'}(v) = \mathrm{Path}_T(v) \setminus U$.

Proof. We perform induction on $|U|$. If $U = \emptyset$, then `splay_to`(x, y) performs no rotations and there is nothing to prove.

Suppose $U \neq \emptyset$, and consider the first `splay_step`(x) (or single rotation at x, if $|U| = 1$) performed by `splay_to`(x, y). We first argue that this operation is allowed. Indeed, we know that $|\partial(T_x)| = 2$, so a single rotation is always allowed (lemma 9.3) and `splay_step`(x) is allowed unless $p = \mathrm{parent}(x)$ is not a separator and $g = \mathrm{parent}(p)$ exists and is a separator (lemma 9.9). But in that case, g is a branching node, so we have $g = y$ and `splay_to` only performs a single rotation instead.

We now show that all nodes in U are removed from the root path of v. This implies $\mathrm{Path}_{T'}(v) = \mathrm{Path}_T(v) \setminus U$, since all rotations involve nodes in U, and we never *add* new nodes to the root path of v.

Let T'' be the tree after the first `splay_step` or rotation. Let c be the child of x in T such that $v \in V(T_c)$. Since x is a branching node, we know that c is not a direct separator and therefore still is a child of x in T''. Thus, all nodes in U that remain on the root path of v are still between x and y. Let $U' = U \cap \mathrm{Path}_{T''}(v)$. We now consider two cases.

Algorithm 9.7 Generalized splaying.

```
   Input: Node $x$ and ancestor $y$
1: procedure splay_to($x, y$)
2:     while parent($x$) $\neq y$ do
3:         if parent(parent($x$)) $= y$ then
4:             rotate at $x$            ▷ Final ZIG
5:         else
6:             splay_step($x$)
```

Algorithm 9.8 TwoPassSplayTT

```
   Input: Node $v$
1: procedure TwoPassSplayTT($v$)
2:     ▷ First pass
3:     Identify branching nodes $b_1, \dots, b_k$
       on the path from $v$ to the root
4:     for each $i \in [k-1]$ do
5:         splay_to($b_i, b_{i+1}$)
6:     splay_to($b_k, \bot$)
7:     ▷ Second pass
8:     splay_to($v, \bot$)
```

- If $|\partial(T''_x)| = 2$, then x is a branching node for v in T'' and our claim follows by induction.
- If$|\partial(T''_x)| = 1$, then $|\partial(T''_u)| = |\partial(T_u)| = 1$ for each $u \in U'$ (otherwise, there is a branching node in U). Any rotation at x or some $u \in U'$ will simply remove the parent of the respective node from the root path of v, maintaining that x and all nodes between x and y are 1-cut. Thus, all rotations performed subsequently are allowed, and ultimately, all nodes in U are removed from the root path of v.

It remains to show that only $|U|$ rotations are performed. For this, simply observe that each rotation performed by `splay_to`(x, y) is either a rotation at x, which reduces its depth by one; or a `splay_step` at x; which reduces its depth by two (see figure 9.4). □

Lemma 9.12 shows that the first pass only performs allowed rotations, and that each rotation removes a node from the root path of v.

We now show that the first pass indeed removes all branching nodes. The nodes between v and b_1 on the root path of v are clearly not touched and thus cannot become branching nodes. Now consider a call `splay_to`(b_i, b_{i+1}) for $i \in [k-1]$ (line 5). Let T, T' be the search trees before and after that call, and let c be the child of b_{i+1} in T that lies on the root path of v. We know that c is not a separator, since b_{i+1} is a branching node. Further, we have $V(T'_{b_i}) = V(T_c)$, so b_i is not a separator and thus not a branching node in T'. Similarly, the call `splay_to`$(b_k, \perp)$ (line 6) makes b_k the root, which cannot be a branching node. Thus, the first pass makes $b_1, b_2, \dots, b_k$ non-branching nodes.

We now turn to the second pass. Let T be the tree produced by the first pass. We know that the root path of v contains no branching nodes, thus `splay_to`$(v, \perp)$ only performs allowed rotations (lemma 9.12) and makes v the root.

Finally, observe that TwoPassSplayTT(v) overall performs $\mathrm{depth}_T(v) - 1$ rotations by lemma 9.12. We have:

Lemma 9.13. *Let v be a node in a 2-cut search tree T.* TwoPassSplayTT(v) *brings v to the root with* $\mathrm{depth}_T(v) - 1$ *rotations, all of which are allowed.*

Remark. If the underlying tree is a path, then all rotations are allowed and there are no branching nodes. Hence, both SplayTT variants are identical to classical Splay in that case.

9.3. Adaptivity properties of SplayTT

In this section, we show that GreedySplayTT and TwoPassSplayTT achieve static optimality and the (logarithmic) working-set bound.

Theorem 9.14. *Let G be a tree and let $X \in V(G)^m$ be an access sequence. Let $w_X \colon V(G) \to \mathbb{N}_0$ count the number of occurrences of each element in X. Then the cost of* GreedySplayTT *and* TwoPassSplayTT *for serving X is* $\mathcal{O}(\mathrm{StOPT}(G, w_X) + \mathrm{StOPT}(G, \mathbb{1}))$.

Theorem 9.15. *Let G be a tree and let $(x_1, x_2, \ldots, x_m) \in V(G)^m$ be an access sequence. For each $i \in [m]$, let W_i denote the set of distinct elements in $x_{j+1}, x_{j+2}, \ldots, x_i$, where $j \in [m]$ is maximal such that $j < i$ and $x_j = x_i$, or $j = 0$ if no such index exists. Then the cost of* GreedySplayTT *and* TwoPassSplayTT *for serving X is $\mathcal{O}(n \log n + \sum_{i=1}^{m} \log |W_i|)$.*

We use the potential method [Tar85] with two different potential functions, one for each theorem. The core of the proof is an amortized analysis of the `splay_step` procedure and is common to both SplayTT variants and both potential functions.

In section 9.3.6, we improve the error term $\mathrm{StOPT}(G, \mathbb{1})$ of theorem 9.14 in certain cases. In particular, it vanishes if all vertices of degree two are accessed at least once.

9.3.1. Potential functions

Our first potential function, used to show static optimality, is inspired by a similar one suggested by Thatchaphol Saranurak for the analysis of classical Splay [CGK+16, § 3.2]. Fix a search tree R on the underlying tree G. We will call R the *reference tree.*

reference tree

For each nonempty subset $U \subseteq V(R) = V(G)$, define

$$\phi_R(U) = -\min_{u \in U} \mathrm{depth}_R(u).$$

Let T be a search tree on G. The *node potential* of a node $x \in V(T)$ is $\phi_R(T_x) = \phi_R(V(T_x))$. In words, the negated node potential is the smallest depth (in R) of x or a descendant of x (in T). The *search tree potential* of T is $\Phi_R(T) = \sum_{x \in V(T)} \phi(T_x)$.

Φ_R

Our second potential function is a simple generalization of one used by Sleator and Tarjan for classical Splay [ST85b]. Let T be a search tree on a tree G. Let $w\colon V(G) \to \mathbb{R}_+$ be a positive weight function on $V(G)$. Define $\Phi_w^{\mathrm{ST}}(T) = \sum_{v \in V(T)} \log w(T_v)$.

Φ_w^{ST}

We now study some essential similarities of the two potential functions, which will allow us to re-use most of the proofs. Observe that Φ^{ST} can be written in a similar way as Φ_R, as follows. For a nonempty subset $U \subseteq V(G)$, define $\phi_w^{\mathrm{ST}}(U) = \log w(U)$. Then we can define a *node potential* $\phi_w^{\mathrm{ST}}(T_v) = \phi_w^{\mathrm{ST}}(V(T_v))$ and a the *search tree potential* $\Phi_w^{\mathrm{ST}}(T) = \sum_{v \in V(T)} \phi_w^{\mathrm{ST}}(T_v)$.

In general, a function Φ mapping search trees on G to a real is called an *additive potential function* if there exists a function $\phi\colon 2^{V(G)} \to \mathbb{R}_{\geq 0}$ such that $\Phi(T) = \sum_{v \in V(T)} \phi(V(T_v))$. We call ϕ the *node potential* corresponding to Φ. We further call Φ *monotone* if $\phi(A) \leq \phi(B)$ for each $A \subseteq B \subseteq V(G)$, and we call Φ *strictly pseudo-concave* if $\phi(A) + \phi(B) \leq 2\phi(C) - 1$ for two disjoint nonempty subsets $A, B \subseteq V(G)$ and each superset $C \supseteq A \cup B$ that induces a connected subgraph of G.

additive

node potential

monotone

strictly pseudo-concave

Clearly, both Φ_w^{ST} and Φ_R (for any R) are additive potential functions. We now show that both are also monotone and strictly pseudo-concave.

Lemma 9.16. *Φ_R is monotone for each reference tree R.*

Proof. Let $A \subseteq B \subseteq V(G)$. The minimum in $\phi(B)$ is taken over a larger set, so it is smaller. The negation in ϕ reverses the inequality, so we have $\phi(A) \leq \phi(B)$. □

Lemma 9.17. *Φ_R is strictly pseudo-concave for each reference tree R.*

Proof. Let $A, B \subseteq V(G)$ be disjoint and let $C \supseteq A \cup B$ induce a connected subgraph of G. Let $x = \mathrm{LCA}_R(C)$. Since $G[C]$ is connected, we have $x \in C$ (lemma 2.4). Observe that x is the unique node minimizing $\mathrm{depth}_R(x)$ in C.

We have $\phi(A), \phi(B) \leq \phi(C)$ by monotonicity (lemma 9.16) and $\phi(C) = -\mathrm{depth}_R(x)$ since $x \in V(C)$ by lemma 2.4. If $x \notin A$, then $\phi(A) < \phi(C)$. If $x \notin B$, then $\phi(B) < \phi(C)$. At least one of these cases hold and both imply the claim. □

Lemma 9.18. Φ_w^{ST} *is monotone for each positive weight function* w.

Proof. This follows directly from monotonicity of the logarithm. □

Lemma 9.19. Φ_w^{ST} *is strictly pseudo-concave for each positive weight function* w.

Proof. Let $A, B \subseteq V(G)$ be disjoint and let $C \supseteq A \cup B$. Observe that $w(C) \geq w(A \cup B) = w(A) + w(B)$. Thus, it suffices to show that for all $x, y \in \mathbb{R}_+$, we have $\log x + \log y \leq 2\log(x+y) - 1$. Assume $x \geq y$. Indeed,

$$\begin{aligned}\log x + \log y &= 2\log(x+y) - \log \tfrac{x+y}{y} - \log \tfrac{x+y}{x} \\ &\leq 2\log(x+y) - \log 2 - \log 1 = 2\log(x+y) - 1.\end{aligned}$$ □

9.3.2. Proof outline

Let Φ be a potential function and let S be a sequence rotations that transform a search tree T into a search tree T'. The *amortized number of rotations in S w.r.t.* Φ is defined as $|S| + \Phi(T') - \Phi(T)$.

amortized no. of rotations

Our key result, proved in the following few sections, is:

Lemma 9.20. *Let Φ be a monotone and strictly pseudo-concave potential function, and let ϕ be the corresponding node potential. For both algorithms* TwoPassSplayTT *and* GreedySplayTT, *there exist constants α, β, γ such that the amortized number of rotations w.r.t. $\alpha \cdot \Phi$ performed by the respective algorithm for an access to a node v is at most $\beta(\phi(T'_v) - \phi(T_v)) + \gamma$, where T is a search tree before the access, and T' is the search tree afterwards.*

We now show that lemma 9.20 implies theorems 9.14 and 9.15. First, the number of pointer moves is essentially dominated by the (actual) number of rotations in both algorithms. Indeed, the number of pointer moves to find the accessed node v is $\mathrm{depth}(v)$, and both algorithms bring v to the root, which needs at least $\mathrm{depth}(v) - 1$ rotations. Apart from the initial search, GreedySplayTT clearly performs $\mathcal{O}(1)$ pointer moves for each rotation, and the two passes of TwoPassSplayTT can be implemented with $\mathcal{O}(\mathrm{depth}(v))$ pointer moves each.

The only case where the cost is not dominated by the number of rotations is when none are performed, because v is already the root. In that case, the cost is $\mathcal{O}(1)$ for both algorithms. Thus, the total cost is $\mathcal{O}(r + m)$, where r is the number of rotations, and m is the length of the input sequence.

We now proceed with the proof theorem 9.14. We first need a technical lemma.

Lemma 9.21. *Let R and T be search trees on a tree G and let $x \in V(T)$. Then* $-\mathrm{depth}_R(x) \leq \phi_R(T_x) \leq -1$.

Proof. Since $x \in V(T_x)$, we have $\min_{u \in V(T_x)} \operatorname{depth}_R(u) \leq \operatorname{depth}_R(x)$. Thus, $\phi_R(T_x) \geq -\operatorname{depth}_R(x)$. Further, $\operatorname{depth}_R(u) \geq 1$ for all $u \in V(G)$, so $\phi_R(T_x) \leq -1$. □

Theorem 9.14. *Let G be a tree and let $X \in V(G)^m$ be an access sequence. Let $w_X \colon V(G) \to \mathbb{N}_0$ count the number of occurrences of each element in X. Then the cost of* GreedySplayTT *and* TwoPassSplayTT *for serving X is $\mathcal{O}(\mathrm{StOPT}(G, w_X) + \mathrm{StOPT}(G, \mathbb{1}))$.*

Proof. Let R be a search tree, and let α, β, γ be the constants from lemma 9.20, depending on the algorithm. W.l.o.g., assume $\beta \geq \gamma$. We consider the amortized number of rotations r_{A} w.r.t. to the scaled potential function $\alpha \cdot \Phi_R$.

Consider an access to v, and let T be the tree before the access and T' be the tree afterwards. Since v is the root of T', we have $\phi_R(T'_v) = \phi_R(T') = -1$. Further, we have $-\phi_R(T_v) \leq \operatorname{depth}_R(x)$ by lemma 9.21. Hence, by lemma 9.20, the amortized number of rotations of each access is at most $-\beta + \beta \cdot \operatorname{depth}_R(x) + \gamma \leq \beta \cdot \operatorname{depth}_R(x)$.

Now let $X = (x_1, x_2, \ldots, x_m) \in V(G)^m$ be a sequence of accesses, and let $w_X \colon V(G) \to \mathbb{N}_0$ count the number of occurrences of each vertex in X, as in the statement of the theorem. Summing up the amortized number of rotations of each access gives

$$r_{\mathrm{A}} \leq \sum_{i=1}^{m} \beta \cdot \operatorname{depth}_R(x_i) = \beta \operatorname{cost}(R, w_X).$$

Let r_i denote the *actual* number of rotations in the i-th access, and let r denote the total number of actual rotations. Let T^0 be the initial search tree and let T^i be the search tree after the i-th access. By definition, we have

$$\begin{aligned} r_{\mathrm{A}} &= \sum_{i=1} \left(r_i + \alpha \cdot \Phi_R(T^i) - \alpha \cdot \Phi_R(T^{i-1}) \right) \\ &= r + \alpha \cdot \Phi_R(T^m) - \alpha \cdot \Phi_R(T^0) \\ &\geq r + \alpha \sum_{v \in V(G)} (-\operatorname{depth}_R(v)) - \alpha \sum_{v \in V(G)} (-1) && \text{by lemma 9.21} \\ &= r - \alpha \operatorname{cost}(R, \mathbb{1}) + \alpha \cdot n. \end{aligned}$$

This implies that $r \leq r_{\mathrm{A}} + \alpha \operatorname{cost}(R, \mathbb{1}) \leq \max(\alpha, \beta) \cdot \operatorname{cost}(R, w_X + \mathbb{1})$. As mentioned above, the total cost of the algorithm is dominated by the rotations, up to an additional term of $\mathcal{O}(m) \subseteq \mathcal{O}(\operatorname{cost}(R, w_X + \mathbb{1}))$.

Now let R be an optimal search tree for the weight function $w_X + \mathbb{1}$, so $\operatorname{cost}(R, w_X + \mathbb{1}) = \mathrm{StOPT}(G, w_X + \mathbb{1})$. By lemma 3.13, this is upper bounded by $2 \cdot (\mathrm{StOPT}(G, w_X) + \mathrm{StOPT}(G, \mathbb{1}))$. This concludes the proof. □

We now prove that both SplayTT variants achieve the working-set bound, using the argument of Sleator and Tarjan from the analysis of Splay [ST85b]. We use the potential function Φ_w^{ST}. Its node potential is easily bounded as follows.

Observation 9.22. *Let T be a search tree on a tree G and let w be a positive weight function on G. For each $v \in V(G)$, we have $\min_{v \in V(G)} \log w(v) \leq \phi_w^{\mathrm{ST}}(T_v) \leq \log w(G)$.*

Theorem 9.15. *Let G be a tree and let $(x_1, x_2, \dots, x_m) \in V(G)^m$ be an access sequence. For each $i \in [m]$, let W_i denote the set of distinct elements in $x_{j+1}, x_{j+2}, \dots, x_i$, where $j \in [m]$ is maximal such that $j < i$ and $x_j = x_i$, or $j = 0$ if no such index exists. Then the cost of* GreedySplayTT *and* TwoPassSplayTT *for serving X is $\mathcal{O}(n \log n + \sum_{i=1}^{m} \log |W_i|)$.*

Proof. Let α, β, γ be the constants from lemma 9.20, depending on the algorithm. We use the potential function Φ^{ST}_w with a *changing* weight function w.

Throughout the execution, we maintain a permutation of the nodes.[2] Let π_0 be a permutation of $V(G)$ in order of first access, with never-accessed vertices at the end in arbitrary order. For $i \in [m]$, we inductively construct the permutation π_i from π_{i-1} by simply moving the i-th accessed element x_i to the front.

The weight function w^i is defined as $w^i(x) = \frac{1}{j^2}$, where j is the index of x in π_i. By lemma 9.20, the amortized number of rotations w.r.t. $\alpha \cdot \Phi^{\mathrm{ST}}_{w_{i-1}}$ for accessing x_i is at most $r^{\mathrm{A}}_i := \beta \cdot (\phi^{\mathrm{ST}}_{w_{i-1}}(T'_{x_i}) - \phi^{\mathrm{ST}}_{w_{i-1}}(T_{x_i})) + \gamma$, where T is the search tree before the access, and T' is the search tree afterwards.

Observe that the position of x_i in π^{i-1} is precisely $|W_i|$. Thus,

$$\phi^{\mathrm{ST}}_{w_{i-1}}(T_{x_i}) \geq \phi^{\mathrm{ST}}_{w_{i-1}}(\{x_i\}) = \log w_{i-1}(x_i) = \log 1/|W_i|^2 = -2 \log |W_i|.$$

Moreover, since x_i is the root of T', we have $\phi^{\mathrm{ST}}_{w_{i-1}}(T'_{x_i}) = \phi^{\mathrm{ST}}_{w_{i-1}}(T') = \log \sum_{i=1}^{n} \frac{1}{i^2} \leq \frac{\pi^2}{6}$, implying that

$$r^{\mathrm{A}}_i \leq \beta \cdot \tfrac{\pi^2}{6} + 2\beta \log |W_i| + \gamma \in \mathcal{O}(\log |W_i|).$$

We now bound the potential change caused by modifying the weight function. When changing w_{i-1} to w_i, the weight of every node decreases, except for x_i. Since x_i is the root at this time, the weight of no proper rooted subtree can increase. Finally, the weight of the whole tree $T = T_{x_i}$ is unchanged, since we merely redistribute weights among the nodes. Formally, we thus have $\Phi^{\mathrm{ST}}_{w_i}(T^i) \leq \Phi^{\mathrm{ST}}_{w_{i-1}}(T^i)$, where T^i is the search tree after the i-th access.

Now let r_i be the actual number of rotations performed in the i-th access. We have

$$r^{\mathrm{A}}_i = r_i + \alpha \cdot \Phi^{\mathrm{ST}}_{w_{i-1}}(T^i) - \alpha \cdot \Phi^{\mathrm{ST}}_{w_{i-1}}(T^{i-1}) \geq r_i + \alpha \cdot \Phi^{\mathrm{ST}}_{w_i}(T^i) - \alpha \cdot \Phi^{\mathrm{ST}}_{w_{i-1}}(T^{i-1}). \tag{9.1}$$

The total number of rotations r thus can be bounded as follows:

$$\begin{aligned} r = \sum_{i=1}^{m} r_i &\leq \sum_{i=1}^{m} r^{\mathrm{A}}_i + \alpha \cdot \Phi^{\mathrm{ST}}_{w_{i-1}}(T^{i-1}) - \alpha \cdot \Phi^{\mathrm{ST}}_{w_i}(T^i) && \text{by eq. (9.1)} \\ &= \alpha \cdot \Phi^{\mathrm{ST}}_{w_0}(T^0) - \alpha \cdot \Phi^{\mathrm{ST}}_{w_m}(T^m) + \sum_{i=1}^{m} r^{\mathrm{A}}_i \\ &\leq \alpha \cdot n \log n + \sum_{i=1}^{m} r^{\mathrm{A}}_i && \text{by observation 9.22} \end{aligned}$$

Since $r^{\mathrm{A}}_i \in \mathcal{O}(\log |W_i|)$, we obtain the desired bound. □

[2]This permutation is only used in the analysis, and not maintained explicitly by any of the algorithms.

Observe that the *error term* in theorem 9.15 is $n \log n$, compared to the stronger $\mathrm{StOPT}(G, \mathbb{1})$ in theorem 9.14. (By theorem 4.1, we always have $\mathrm{StOPT}(G, \mathbb{1}) \in \mathcal{O}(n \log n)$.)

The potential function and overall analysis for theorem 9.15 are directly lifted from Sleator and Tarjan's Splay analysis for BSTs [ST85b], so it does not take into account the structure of the underlying tree at all. It seems necessary for the proof to *change* the potential function during operation, which makes a reference-tree based approach difficult. Perhaps a variant of Φ_R can be used where, whenever a node v is accessed, we rotate v to the top of the reference tree R.

Open question 9.2. Does SplayTT achieve the working-set bound with additive error $\mathcal{O}(\mathrm{StOPT}(G, \mathbb{1})$?

In the next few sections, we prove lemma 9.20.

9.3.3. Potential change of rotations and `splay_step`

We now analyze the potential change caused by rotations and calls to `splay_step`. Refer to figures 9.2 and 9.4 for illustration. For the remainder of section 9.3, fix a monotone and strictly pseudo-concave additive potential function Φ, and let ϕ be the associated node potential.

Lemma 9.23. *Let T be an STT, and let T' be produced from T by a single rotation at $v \in V(T)$. Then*

$$\Phi(T') - \Phi(T) = \phi(T'_p) - \phi(T_v) \leq 3(\phi(T'_v) - \phi(T_v)).$$

Proof. Let $p = \mathrm{parent}(v)$. Observe that the node potential only changes at v and p, since all other nodes neither gain nor lose descendants. We thus have

$$\begin{aligned}
\Phi(T') - \Phi(T) &= \phi(T'_p) + \phi(T'_v) - \phi(T_p) - \phi(T_v) \\
&= \phi(T'_p) - \phi(T_v) && \text{since } V(T'_v) = V(T_p) \\
&\leq \phi(T'_v) - \phi(T_v) && \text{by monotonicity, since } V(T'_p) \subseteq V(T'_v) \\
&\leq 3(\phi(T'_v) - \phi(T_v)). && \text{by monotonicity, since } V(T_v) \subseteq V(T'_v)
\end{aligned}$$

□

Lemma 9.24. *Let T be an STT, and let T^* be produced from T by a* `splay_step` *at $v \in V(T)$. Then*

$$\Phi(T^*) - \Phi(T) \leq 3(\phi(T^*_v) - \phi(T_v)) - 1.$$

Proof. Let p, g be parent and grandparent of v in the initial tree T. Let T' be the tree after the first of the two rotations.

ZIG-ZAG. We rotate twice at v, first with p and then with g. By lemma 9.23, the total potential change is

$$\Phi(T^*) - \Phi(T) = \phi(T'_p) - \phi(T_v) + \phi(T^*_g) - \phi(T'_v).$$

Since p is not involved in the second rotation, we have $\phi(T^*_p) = \phi(T'_p)$. Moreover, we know that $V(T^*_p)$ and $V(T^*_g)$ are disjoint subsets of $V(T^*_v)$, since they are both child subtrees of v in T^*. Concavity thus implies that $\phi(T'_p) + \phi(T^*_g) \leq 2\phi(T^*_v) - 1$. Further observe that $\phi(T_v) \leq \phi(T_p) = \phi(T'_v)$ by monotonicity, so

$$\Phi(T^*) - \Phi(T) \leq 2(\phi(T^*_v) - \phi(T_v)) - 1 \leq 3(\phi(T^*_v) - \phi(T_v)) - 1.$$

ZIG-ZIG. We first rotate p with g. Since we perform a ZIG-ZIG step, we know that v is not a direct separator in T, so v and g are both children of p in T'. The potential change is

$$\begin{aligned}
\Phi(T') - \Phi(T) &= \phi(T'_g) - \phi(T_p) && \text{by lemma 9.23}\\
&= \phi(T'_g) + \phi(T'_v) - \phi(T_p) - \phi(T_v) && \text{since } V(T'_v) = V(T_v)\\
&= 2\phi(T'_p) - 1 - \phi(T_p) - \phi(T_v) && \text{by concavity}\\
&\leq 2\phi(T'_p) - 1 - 2\phi(T_v) && \text{by monotonicity, since } p \prec_T v\\
&= 2\phi(T^*_v) - 1 - 2\phi(T_v) && \text{since } V(T^*_v) = V(T'_p)
\end{aligned}$$

The second rotation of v with p changes the potential by at most $\phi(T^*_v) - \phi(T'_v) = \phi(T^*_v) - \phi(T_v)$ (lemma 9.23). Adding up the two potential changes yields the desired bound. □

9.3.4. Potential change of Two-Pass SplayTT

We now analyze TwoPassSplayTT, starting with the `splay_to` procedure.

Lemma 9.25. *Let T be an STT, and let T^* be produced from T by calling* `splay_to`(x, y) *for some $x \in V(T)$ and $y \in V(T) \cup \{\bot\}$, and let $k = \text{depth}_T(x) - \text{depth}_T(y) \geq 1$, with $\text{depth}_T(\bot) = 0$. Then*

$$\Phi(T^*) - \Phi(T) \leq 3(\phi(T^*_x) - \phi(T_x)) - \lfloor \tfrac{k-1}{2} \rfloor$$

Proof. Recall that `splay_to`(x, y) performs ZIG-ZIG or ZIG-ZAG steps as long as the depth difference of x and y is at least three, and each of those steps decreases this depth difference by two. (If $y = \bot$, the depth difference is simply the depth of x.) Hence, `splay_to`(x, y) performs $\ell := \lfloor \frac{k-1}{2} \rfloor$ ZIG-ZIG or ZIG-ZAG steps and then a final ZIG step (single rotation) if k is even.

Let T^i be the search tree after i such steps, with $T^0 = T$. By lemma 9.24, we have

$$\begin{aligned}
\Phi(T^\ell) - \Phi(T) &= \sum_{i=1}^{\ell} \Phi(T^i) - \Phi(T^{i-1})\\
&\leq \sum_{i=1}^{\ell} 3(\phi(T^i_v) - \phi(T^{i-1}_v)) - 1 = 3(\phi(T^\ell_v) - \phi(T_v)) - \ell.
\end{aligned}$$

If k is odd, then $T^* = T^\ell$ and we are done. Otherwise, we have

$$\Phi(T^*) - \Phi(T) = \Phi(T^\ell) - \Phi(T) + \Phi(T^*) - \Phi(T^\ell)$$

and $\Phi(T^*) - \Phi(T^\ell) \leq 3(\phi(T^*_v) - \phi(T^\ell_v))$ by lemma 9.23, thus $\Phi(T^\ell) - \Phi(T) \leq 3(\phi(T^*_v) - \phi(T)) - \ell$, as desired. □

If `splay_to` performs more than one rotation (i.e., $k > 2$), then the term $-\lfloor\frac{k-1}{2}\rfloor$ "pays" for the rotations (when scaling the potential function appropriately). However, TwoPassSplayTT may perform many single-rotation `splay_to`s, e.g., if every other node on the root path of the accessed node is a branching node. The trick is to scale the potential function a little more, so that the second pass pays for the unaccounted rotations of the first pass. We proceed with the formal proof.

Lemma 9.26. *Let T be an STT, and let T^* be produced from T by accessing a node $v \in V(T)$ with* TwoPassSplayTT*. Then*

$$\Phi(T^*) - \Phi(T) \le 6(\phi(T_v^*) - \phi(T_v)) - \tfrac{1}{4}\operatorname{depth}_T(v) + 1.$$

Proof. We start with the first phase. Let $b_1, b_2, \ldots, b_k$ be the branching nodes as in algorithm 9.8 and set $b_{k+1} = \perp$. Let T^i be the search tree after performing the i-th call `splay_to`(b_i, b_{i+1}), and let $T^0 = T$. For convenience, in the following we write $\phi^i(x) = \phi(T_x^i)$ for all $i \in \{0, 1, \ldots, k+1\}$ and $x \in V(T)$.

By lemma 9.25, the potential change of the i-th call to `splay_to` is

$$\Phi(T^i) - \Phi(T^{i-1}) \le 3(\phi^i(b_i) - \phi^{i-1}(b_i)) - \left\lfloor \frac{\operatorname{depth}_{T^{i-1}}(b_i) - \operatorname{depth}_{T^{i-1}}(b_{i+1}) - 1}{2} \right\rfloor,$$

where $\operatorname{depth}(\perp) = 0$ and $\phi^i(\perp) = \phi^i(T)$.

By lemma 9.12, we have $\operatorname{depth}_{T^i}(b_j) = \operatorname{depth}_T(b_j)$ for all $i < j$, since b_j is not touched until the j-th call. Moreover, we know that b_i stays a descendant of b_{i+1} throughout the first phase, so in particular $\phi^i(b_i) \le \phi^i(b_{i+1})$ for all $i \in [k-1]$. In the same vain, observe that $\phi^0(v) \le \phi^0(b_1)$ since v is a descendant of b_1 in T, and $\phi^k(b_k) \le \phi^k(b_{k+1}) = \phi^k(\perp) = \phi(T)$. This implies

$$\begin{aligned}
\Phi(T^k) - \Phi(T) &\le \sum_{i=1}^{k} \Phi(T^i) - \Phi(T^{i-1}) \\
&\le \sum_{i=1}^{k} 3(\phi^i(b_i) - \phi^{i-1}(b_i)) - \left\lfloor \frac{\operatorname{depth}_{T^{i-1}}(b_i) - \operatorname{depth}_{T^{i-1}}(b_{i+1}) - 1}{2} \right\rfloor \\
&\le \sum_{i=1}^{k} 3(\phi^i(b_{i+1}) - \phi^{i-1}(b_i)) - \left\lfloor \frac{\operatorname{depth}_{T}(b_i) - \operatorname{depth}_{T}(b_{i+1}) - 1}{2} \right\rfloor \\
&\le 3(\phi^k(b_{k+1}) - \phi^0(b_1)) - \tfrac{1}{2}(\operatorname{depth}_T(b_1) - \operatorname{depth}_T(b_{k+1})) + k \\
&\le 3(\phi(T) - \phi^0(v)) - \tfrac{1}{2}\operatorname{depth}_T(b_1) + k
\end{aligned}$$

By lemma 9.12, after the first phase, the root path of v consists of precisely v, the nodes between v and b_1 in T, and the branching nodes $b_1, b_2, \ldots, b_k$. Thus, $\operatorname{depth}_{T^k}(v) = (\operatorname{depth}_T(v) - \operatorname{depth}_T(b_1)) + k$.

With this in mind, the search tree potential changes as follows in the second phase, when we call `splay_to`$(v, \perp)$.

$$\begin{aligned}
\Phi(T^*) - \Phi(T^k) &\le 3(\phi(T_v^*) - \phi(T_v^k)) - \left\lfloor \frac{\operatorname{depth}_{T^k}(v) - 1}{2} \right\rfloor \\
&\le 3(\phi(T_v^*) - \phi(T_v^k)) - \tfrac{1}{2}(\operatorname{depth}_T(v) - \operatorname{depth}_T(b_1) + k) + 1.
\end{aligned}$$

Observe that $\phi(T_v^*) = \phi(T^*) = \phi(T)$, since $v = \operatorname{root}(T^*)$, and that $T_v = T_v^k$, since v is not touched by the first phase. Further observe that $k \leq \frac{1}{2}\operatorname{depth}_T(v)$, since two branching nodes cannot be next to each other on the root path of v. The following calculation finishes the proof.

$$\begin{aligned}\Phi(T^*) - \Phi(T) &= \Phi(T^*) - \Phi(T^k) + \Phi(T^k) - \Phi(T)\\ &\leq 3(\phi(T) + \phi(T_v^*) - \phi(T_v) - \phi(T_v^k)) - \tfrac{1}{2}\operatorname{depth}_T(v) + \tfrac{k}{2} + 1.\\ &\leq 6(\phi(T_v^*) - \phi(T_v)) - \tfrac{1}{4}\operatorname{depth}_T(v) + 1. \qquad \square\end{aligned}$$

Since TWOPASSSPLAYTT(v) performs $\operatorname{depth}_T(v) - 1$ rotations (lemma 9.13), scaling the potential function by four lets us use the potential change to "pay" for the rotations. We obtain:

Corollary 9.27. *The amortized number of rotations w.r.t.* $4 \cdot \Phi$ *performed by* TWOPASS-SPLAYTT *when accessing a node* v *is at most* $24(\phi(T_v^*) - \phi(T_v)) + 3$*, where* T *is the search tree before the access, and* T^* *is the search tree afterwards.*

This concludes the proof of the first part of lemma 9.20.

9.3.5. Potential change of Greedy SplayTT

This algorithm does not use `splay_to`. Its analysis is a bit more difficult and technical, mainly due to the fact that some performed `splay_step`s only remove one node from the root path of v, instead of two.

Our analysis uses an additional potential function to amortize over calls to `splay_step` in GREEDYSPLAYTT. Let T be an STT and let $v \in V(T)$. We define $\Psi_v(T) = \phi(T_v) + \phi(T_p) + \phi(T_g)$, where p and g are the parent and grandparent of v in T; either is $\perp$ if that node does not exist, and define $\phi(T_\perp) = \phi(T) + 1$.

Lemma 9.28. *Consider a call* GREEDYSPLAYTT(v)*. Let* T *be the STT before some* `splay_step`*. Let* T' *be the STT after that* `splay_step`*, and let* T'' *be the STT after the following* `splay_step` *(if there is such a step). Then one of the following holds:*

(a) $\Phi(T') - \Phi(T) \leq 3(\Psi_v(T') - \Psi_v(T)) - 1$.

(b) $\Phi(T'') - \Phi(T) \leq 3(\Psi_v(T'') - \Psi_v(T)) - 2$ *(and* T'' *exists).*

Proof. Let p, p', p'', g, g', g'' be the parent and grandparent of v in the respective tree. (All non-existing nodes are $\perp$.) In the following, we write $\Psi = \Psi_v$ for short.

We have $\Psi(T') = \phi(T'_v) + \phi(T'_{p'}) + \phi(T'_{g'})$ and $\Psi(T'') = \phi(T''_v) + \phi(T''_{p''}) + \phi(T''_{g''})$. Consider the following cases.

(i) If p is the root of T, then `splay_step` executes a single rotation. By lemma 9.23, the potential change is $\Phi(T') - \Phi(T) \leq 3(\phi(T'_v) - \phi(T_v))$.

Observe that $g = g' = \perp$, so $\phi(T_g) = \phi(T'_{g'})$. Further, since p is the root of T and $p' = \perp$, we have $\phi(T'_{p'}) = \phi(T_p) + 1$, and thus

$$\phi(T'_v) - \phi(T_v) = \phi(T'_v) - \phi(T_v) + \phi(T'_{p'}) - \phi(T_p) + \phi(T'_{g'}) - \phi(T_g) - 1 = \Psi(T') - \Psi(T) - 1.$$

This proves the claim. From now on, assume p is not the root of T.

(ii) If we call `splay_step`(v), then the nodes p and g are simply removed from the root path of v, so p' and g' are ancestors of g in T (or nonexistent). This implies that $\phi(T_p) \leq \phi(T'_{p'})$ and $\phi(T_g) \leq \phi(T'_{g'})$, so $\phi(T'_v) - \phi(T_v) \leq \Psi(T') - \Psi(T)$.

By lemma 9.24, we have $\Phi(T') - \Phi(T) \leq 3(\phi(T'_v) - \phi(T_v)) - 1$, implying the claim.

(iii) Suppose we call `splay_step`(p). Since `splay_step`(v) was disallowed, we know (by lemma 9.9) that at least one of v and p is not a separator in T, and g is a separator in T. In particular, this implies that $\text{depth}_T(v) = 2 + \text{depth}_T(g) \geq 5$.

Suppose first that v is not a separator in T. Then v stays a child of p (by lemma 2.10), so `splay_step`(p) simply removes g and its parent h from the root path of v, and essentially the same analysis as in (ii) applies.

Now suppose v is a separator, and thus p is not a separator. Then, we have $\partial(v) \subseteq \{p\} \cup \partial(p) = \{p, g\}$, hence v is a *direct separator*. This is the case where we need to look at two consecutive `splay_step`s, since $\phi(T'_v) - \phi(T_v) \leq \Psi(T') - \Psi(T)$ does not necessarily hold.

Figure 9.5 illustrates the situation. The key insight is that g is a separator in T' (after the current ZIG-ZIG step at p). To see this, observe that $\partial(T_g) = \{h, a\}$ for some node a that is a proper ancestor of h. On the other hand, we have $a \notin \partial(T_p) = \{g\}$, since p is not a separator. Further observe that $V(T'_g) \supseteq V(T_g) \setminus V(T_p)$. Thus, we have $a \in \partial(T'_g)$, so g is a separator in T'.

Since v is still a direct separator in T' (note that $\partial(T'_v) = \partial(T_v) = \{p, g\}$), the next `splay_step` will be a ZIG-ZAG at v. Call the resulting tree T'' (see figure 9.5).

We now bound the potential change of the two `splay_step`s, which is

$$\Phi(T'') - \Phi(T) = 3(\phi(T'_p) - \phi(T_p) + \phi(T''_v) - \phi(T'_v)) - 2. \tag{9.2}$$

by lemma 9.24.

We have $V(T_v) = V(T'_v)$, $V(T'_p) = V(T''_v)$, implying

$$\begin{aligned}&\phi(T''_v) - \phi(T'_v) = \phi(T''_v) - \phi(T_v), \text{ and}\\ &\phi(T'_p) - \phi(T_p) = \phi(T''_v) - \phi(T_p) \leq \phi(T''_{p''}) - \phi(T_p).\end{aligned}$$

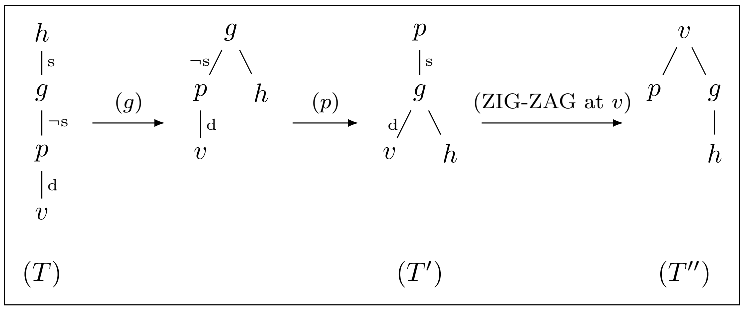


Figure 9.5.: Illustration of the special case in lemma 9.29 (iii), where we first perform a ZIG-ZIG at p, and then a ZIG-ZAG at v. "s", "¬s", and "d" have the same meaning as in figure 9.2.

Moreover, since $V(T_h) = V(T''_v)$, we have $\phi(T_g) \leq \phi(T_h) = \phi(T''_v) \leq \phi(T''_{g''})$ by monotonicity, so $\phi(T''_{g''}) - \phi(T_g) \geq 0$. Thus,

$$\begin{aligned}
&\phi(T'_p) - \phi(T_p) + \phi(T''_v) - \phi(T'_v) \\
&\leq \phi(T''_v) - \phi(T_v) + \phi(T''_{p''}) - \phi(T_p) + \phi(T''_{g''}) - \phi(T_g) \\
&= \Psi(T'') - \Psi(T).
\end{aligned}$$

Together with eq. (9.2), this implies $\Phi(T'') - \Phi(T) = 3(\Psi(T'') - \Psi(T)) - 2$.

(iv) Finally, suppose we call `splay_step`(g). Then `splay_step` at both v and p must be disallowed. The former implies that g is a separator, and the latter implies that p and g cannot both be separators, so p is not a separator (in T). Thus, `splay_step`(g) does not modify the parent of p, and simply removes h and the parent of h from the root path of v, so we have $g = g'$ and $p = p'$ and the potential of v and p does not change. Hence, we have $\Psi(T') - \Psi(T) = \phi(T'_g) - \phi(T_g)$, and the same analysis as in (ii) applies. □

We now analyze a full access.

Lemma 9.29. *Let T be an STT, and let T^* be produced from T by accessing some node $v \in V(T)$ with* GreedySplayTT. *Then*

$$\Phi(T^*) - \Phi(T) \leq 9(\phi(T^*_v) - \phi(T_v)) - \tfrac{1}{2}\operatorname{depth}_T(v) - 5$$

Proof. Say we execute `splay_step` k times. Lemma 9.28 implies

$$\Phi(T^*) - \Phi(T) \leq 3(\Psi_v(T^*) - \Psi_v(T)) - k.$$

Since v is the root of T^*, the parent of grandparent of v are both $\bot$, which means $\Psi_v(T^*) = \phi(T^*_v) + 2 \cdot \phi(T^*_\bot) = 3 \cdot \phi(T^*_v) + 2$ by definition of $\phi(T_\bot)$.

On the other hand, since the node-potential of a parent cannot be lower than the node-potential of its child by monotonicity, we have $\Psi_v(T) \geq 3 \cdot \phi(T_v)$. All in all, we have $\Phi(T^*) - \Phi(T) \leq 9(\phi(T^*_v) - \phi(T_v)) - k - 6$. Since each `splay_step` reduces the depth of v by at most two, we have $\operatorname{depth}_T(v) \leq 2k + 1$. This implies the claim. □

By lemma 9.11, each `splay_step` performed in a call GreedySplayTT(v) decreases the depth of v by at least one. Thus, it performs at most $2 \cdot (\operatorname{depth}(v) - 1)$ rotations, which are paid for by scaling the potential function with four. We have:

Corollary 9.30. *The amortized number of rotations w.r.t. $4 \cdot \Phi$ performed by* Greedy-SplayTT *when accessing a node v is at most $36(\phi(T^*_v) - \phi(T_v)) - 21$, where T is the search tree before the access, and T^* is the search tree afterwards.*

Together with corollary 9.27, this proves lemma 9.20.

We note that with a more careful analysis, the leading constant $\beta = 32$ in corollary 9.27 can be improved. This is because the bound of $2 \cdot (\operatorname{depth}(v) - 1)$ rotations is too pessimistic. Indeed, it is implicit in the proof of lemma 9.28 that each `splay_step` decreases the depth of v be two, except for the situation illustrated in figure 9.5, where two `splay_step`s (with four rotations) decrease the depth by three. Hence, we have at most $\frac{4}{3} \cdot \operatorname{depth}(v)$

rotations, which means we can use the potential function $\frac{8}{3} \cdot \Phi$ and the leading constant in corollary 9.27 becomes $\beta = 24$.

Further following this line of thinking, observe that the transformation of figure 9.5 can be implemented in just three rotations (at g, then twice v), so the algorithm can be modified to use only $\operatorname{depth}(v) - 1$ rotations, yielding $\beta = 18$.

9.3.6. Improved Static Optimality

In this section, we improve the additive $\operatorname{StOPT}(G, \mathbb{1})$ term in theorem 9.14.

Theorem 9.31. *Let G be a tree and let $X \in V(G)^m$ be an access sequence. Let $w_X \colon V(G) \to \mathbb{N}_0$ count the number of occurrences of each element in X, and let w' be the weight function on $V(G)$ with*

$$w'(v) = \begin{cases} 1, & \textit{if } \deg_G(v) = 2 \textit{ and } w_X(v) = 0, \\ 0, & \textit{otherwise.} \end{cases}$$

Then there is an initial search tree T only depending on G such that the cost of GreedySplayTT *and* TwoPassSplayTT *for serving X is $\mathcal{O}(\operatorname{StOPT}(G, w_X) + \operatorname{StOPT}(G, w'))$.*

In particular, we obtain tight static optimality as long as each vertex of degree exactly two is accessed at least once. Observe that this improvement is essentially useless if G is a path (the BST case), since then almost every vertex has degree two, and if every vertex is accessed at least once, then $\operatorname{StOPT}(G, w_X)$ dominates $\operatorname{StOPT}(G, \mathbb{1})$ anyway.

The proof consists of two parts. First, we handle vertices of degree at least three with an analysis of the quantity $\operatorname{StOPT}(G, w_X) + \operatorname{StOPT}(G, \mathbb{1})$.

Lemma 9.32. *Let (G, w) be a connected weighted graph with integral weights. If G has no vertex v with both $\deg_G(v) \leq 2$ and $w(v) = 0$, then $\operatorname{StOPT}(G, w + \mathbb{1}) \leq 10 \cdot \operatorname{StOPT}(G, w)$.*

Lemma 9.32 can be applied to arbitrary dynamic search tree algorithms with guarantees similar to theorem 9.14. It also will be useful in part III of the thesis.

To prove lemma 9.32, we first need some technical lemmas. The concept of *2-cut* search trees is useful once again.

Lemma 9.33. *Let T be a search tree on a tree G, and let $v \in V(T)$. Let N_v be the set of neighbors of v in G. The number of children of v in T is precisely $|N_v \cap V(T_v)|$.*

Proof. First, for each child c of v, we have that $G[T_c]$ is adjacent to v, so there is a node $u \in N_v \cap V(T_c)$. Since the child subtrees are disjoint, the number of children of v is at most $|N_v \cap V(T_v)|$.

Now suppose two neighbors $u, u' \in N_v$ are in the same child subtree T_c. Then u and u' are not connected in $G[T_c]$, since G is a tree and $v \notin V(T_c)$. This contradicts lemma 2.3. □

Lemma 9.34. *Let T be a search tree on a tree G, and let c be a child of a node p in T. Let k_c, k_p be the number of children of c, resp. p. Then*

$$k_c + k_p + |\partial(T_p)| \geq \deg_G(v) + \deg_G(p) - 1.$$

Proof. Let N_c be the set of neighbors of c and N_p be the set of neighbors of p in G. By lemma 9.33, we have $k_c \geq |N_c \cap V(T_c)|$ and $k_p \geq |N_p \cap V(T_p)|$. It thus remains to show that $|\partial(T_p)| \geq |N_c \setminus V(T_c)| + |N_p \setminus V(T_p)| - 1$.

Every node in $u \in N_p$ that is not in T_p must be an ancestor of p, and thus $u \in \partial(T_p)$. Similarly, every node $u \in N_c$ that is not in T_c must be a strict ancestor of c, and thus an ancestor of p, so $u \in \partial(T_p)$ unless $u = p$. We have

$$\begin{aligned} &\partial(T_p) \supseteq (N_c \setminus \{p\} \setminus V(T_c)) \cup (N_p \setminus V(T_p)) \\ \implies &|\partial(T_p)| \geq |N_c \setminus \{p\} \setminus V(T_c)| + |N_p \setminus V(T_p)| - |(N_p \cap N_c) \setminus V(T_p)| \end{aligned} \tag{9.3}$$

If c and p are adjacent in G, then N_c and N_p are disjoint but $p \in N_c$. Thus, eq. (9.3) implies

$$|\partial(T_p)| \geq |N_c \setminus V(T_c)| - 1 + |N_p \setminus V(T_p)|, \tag{9.4}$$

as desired. Otherwise, we still have $|N_c \cap N_p| \leq 1$ since G is a tree, and $p \notin N_c$, so eq. (9.4) again follows from eq. (9.3). □

Lemma 9.35. *Let T be a 2-cut search tree on a tree G and let S be a subtree of T such that for each $v \in V(S)$, we have $\deg_G(v) \geq 3$. Let $\ell(S)$ be the number of nodes in $V(T) \setminus V(S)$ whose parents are in S. Then $\ell(S) \geq \frac{1}{3}|S| + \frac{2}{3}$.*

Proof. We write $|S| = |V(S)|$ in this proof, and proceed by induction on $|S|$. Let $r = \mathrm{root}(S)$. If $|S| = 1$, then one of the at least three neighbors of r must be a descendant of r; otherwise, $|\partial(T_r)| \geq 3$, a contradiction. Thus, $\ell(S) \geq 1 = \frac{1}{3}|S| + \frac{2}{3}$.

Let $|S| \geq 2$. Let K be the set of children of r in S, and let K' be the set of children of r in T that are not in S. Suppose $|K| + |K'| \geq 2$. Then, by induction,

$$\begin{aligned} \ell(S) &\geq |K'| + \sum_{c \in K} \ell(S_c) \geq |K'| + \sum_{c \in K} (\tfrac{1}{3}|S_c| + \tfrac{2}{3}) \\ &\geq \tfrac{1}{3}(|S| - 1) + |K'| + \tfrac{2}{3}|K| \geq \tfrac{1}{3}|S| + 1. \end{aligned}$$

Now suppose $|K| + |K'| \leq 1$. Since $|S| \geq 2$, we have $K \neq \emptyset$, so $|K| = 1$ and $|K'| = 0$. Let $\{c\} = K$, let K_c be the set of children of c in S, and let K_c' be the set of children of c in T that are not in S. By lemma 9.34, and since T is 2-cut and $\deg_G(r), \deg_G(c) \geq 3$, we have $(|K_c| + |K_c'|) + 1 + 2 \geq 5$, implying $|K_c| + |K_c'| \geq 2$. Again using induction, we have

$$\begin{aligned} \ell(S) &\geq |K_c'| + \sum_{g \in K_c} (\tfrac{1}{3}|S_g| + \tfrac{2}{3}) \geq |K_c'| + \tfrac{1}{3}(|S| - 2) + \tfrac{2}{3}|K_c| \\ &\geq \tfrac{1}{3}(|S| - 2) + \tfrac{2}{3}(|K_c'| + |K_c|) \geq \tfrac{1}{3}|S| + \tfrac{2}{3}. \end{aligned}$$ □

We are now ready to prove lemma 9.32.

Lemma 9.32. *Let (G, w) be a connected weighted graph with integral weights. If G has no vertex v with both $\deg_G(v) \leq 2$ and $w(v) = 0$, then $\mathrm{StOPT}(G, w + \mathbb{1}) \leq 10 \cdot \mathrm{StOPT}(G, w)$.*

Proof. Let $w' = w + \mathbb{1}$ and let T be an optimal 2-cut search tree on (G, w). We have $\mathrm{cost}(T, w) \leq 2 \cdot \mathrm{StOPT}(G, w)$ by theorem 5.7. We show that $\mathrm{cost}(T, w') \leq 5 \cdot \mathrm{cost}(T, w)$, which implies our claim.

We construct the weight function w'' from w' as follows. Let P be a maximal subtree of T where $w(v) = 0$ for each $v \in V(P)$. Let L be the set of nodes in $T - P$ whose parents are in P. Take the total weight of all nodes in P (which is precisely $|P|$ by definition of w') and distribute it uniformly among the nodes in L. By lemma 9.35, this increases the weight of each $v \in L$ by at most 3.

Repeatedly apply this process for every such subtree P to obtain w''. Note that there is no overlap in the affected nodes. Since we only move weight from nodes with lower depth to nodes with higher depth, we have $\operatorname{cost}(T, w'') \geq \operatorname{cost}(T, w')$. Moreover, for each node v with $w(v) = 0$, we have $w''(v) = 0$, and for each node v with $w(v) \geq 1$, we have $w''(v) \leq w'(v) + 3 = w(v) + 4 \leq 5 \cdot w(v)$. All in all, we have

$$\operatorname{cost}(T, w') \leq \operatorname{cost}(T, w'') \leq 5 \cdot \operatorname{cost}(T, w). \qquad \square$$

To prove theorem 9.31, it remains to handle *leaves* that are not accessed, which we do by choice of the initial search tree.

Theorem 9.31. *Let G be a tree and let $X \in V(G)^m$ be an access sequence. Let $w_X \colon V(G) \to \mathbb{N}_0$ count the number of occurrences of each element in X, and let w' be the weight function on $V(G)$ with*

$$w'(v) = \begin{cases} 1, & \text{if } \deg_G(v) = 2 \text{ and } w_X(v) = 0, \\ 0, & \text{otherwise.} \end{cases}$$

Then there is an initial search tree T only depending on G such that the cost of GreedySplayTT *and* TwoPassSplayTT *for serving X is $\mathcal{O}(\mathrm{StOPT}(G, w_X) + \mathrm{StOPT}(G, w'))$.*

Proof. If $|V(G)| \leq 2$, the theorem is trivially true. Otherwise, let the initial tree T be a rooting of G at an arbitrary non-leaf vertex. Clearly T is 2-cut and therefore a valid starting tree for SplayTT.

Let L be the set of leaves of G that are never accessed. Crucially, each leaf of G, including each vertex in L, is also a leaf of T. Let T' be the final search tree. Since both SplayTT variants only touch root paths of accessed nodes, no $\ell \in L$ is ever touched, and each leaf in L must still be a leaf of T'.

Let $H = G - L$ and $S = T - L$. By the above discussion, running either SplayTT variant to serve X on G with initial tree T is essentially the same, and has precisely the same cost, as running it to serve X on H with initial tree S. By theorem 9.14, this cost is

$$\begin{aligned}
&\mathrm{StOPT}(H, w_X) + \mathrm{StOPT}(H, \mathbb{1}) \\
&\leq \mathrm{StOPT}(H, w_X + \mathbb{1}) && \text{by lemma 3.13} \\
&\leq \mathrm{StOPT}(H, (w_X + w') + \mathbb{1}) \\
&\leq 10\,\mathrm{StOPT}(H, w_X + w') && \text{by lemma 9.32} \\
&\leq 20(\mathrm{StOPT}(H, w_X) + \mathrm{StOPT}(H, w')) && \text{by lemma 3.13}
\end{aligned}$$

This concludes the proof. $\square$

9.4. More variants of SplayTT

In this section, we discuss further variations of SplayTT. Our goal is not to obtain improved theoretical guarantees of any sort, but rather to come up with an algorithm that is easy to implement and fast in practice.

We consider multiple ideas to improve TwoPassSplayTT. Combining them, we end up with an algorithm that, somewhat surprisingly, is very close to GreedySplayTT.

The first idea for improvement is to combine the two passes of TwoPassSplayTT into a single one. See algorithm 9.9 for pseudocode. Essentially, in each step, we check whether the parent or grandparent of the accessed node v is a branching node. If one of them is, we move it up to the next branching node (or the root) with the procedure SplayBranchingNode. Otherwise, we execute a `splay_step` at v (which then is easily seen to be valid by lemma 9.9).

Observe that this implementation performs exactly the same rotations as the original TwoPassSplayTT, just in a different order. Also, all rotations are performed at v, its parent, its grandparent, or its great-grandparent (the latter only happens if we perform a ZIG-ZIG step within SplayBranchingNode(g)).

We now discuss two further ideas to improve this implementation. First, we could *first* try to execute `splay_step`(v), before checking for branching nodes. If that is not valid, either the parent or grandparent must be a branching node (by lemma 9.9), and we proceed as normal.

This change makes the algorithm "skip" some branching nodes that do not interfere with the splaying of v. Reordering the rotations into two passes would give us an algorithm where we skip some of the calls `splay_to`(b_i, b_{i+1}) present in the original TwoPassSplayTT. Clearly, the analysis transfers to the new variant.

Also observe that we only introduce new calls `splay_step`(v), so every ZIG-ZAG or ZIG-ZIG step still reduces the depth of v by two. In particular, we do not perform unnecessary rotations like in GreedySplayTT.

The second idea is that the calls `splay_to`(b_i, b_{i+1}) in the original TwoPassSplayTT (which are implicit in the variant) are unnecessarily "greedy". In the current tree, let x be the nearest ancestor of b_i that is 1-cut, and let y be the parent of x. Calling `splay_to`(b_i, y) clearly makes b_i 1-cut and thus not a separator node. Since this is the only reason we call `splay_to`(b_i, b_{i+1}) in the first place, we can replace it by `splay_to`(b_i, y). Again, the analysis is not affected.

Algorithm 9.10 (LocalTwoPassSplayTT) combines the two ideas. Observe that the "lookahead" is less than in algorithm 9.9: the latter sometimes needs to check if the *great-great-grandparent* is a separator, while LocalTwoPassSplayTT only checks this for the *great-grandparent*.

Remark. LocalTwoPassSplayTT looks very similar to GreedySplayTT. In fact, it can be seen that the two algorithms differ in only one situation: When p is not a separator, g is a separator, and $g_2 = \text{parent}(g)$ is not a separator. In this case, GreedySplayTT calls `splay_step`(p), while LocalTwoPassSplayTT calls `rotate`(g).

This was exactly the case in the proof of lemma 9.28 that caused problems and necessitated considering two `splay_step`s at once (figure 9.5). As mentioned at the end

Algorithm 9.9 A one-pass implementation of TwoPassSplayTT.

```
procedure TwoPassSplayTT($v$)
    while $v$ has parent $p$ do
        if $p$ has parent $g$ then
            if $p$ is br. node for $v$ then
                SplayBranchingNode($p$)
            else if $g$ is br. node for $v$ then
                SplayBranchingNode($g$)
            else
                splay_step($v$)
        else
            rotate($v$)
```

```
procedure SplayBranchingNode($v$)
    while $v$ has parent $p$ do
        if $p$ has parent $g$ then
            if $p$ is br. node for $v$ then
                return
            else if $g$ is br. node for $v$ then
                rotate($v$)
                return
            else
                splay_step($v$)
        else
            rotate($v$)
```

Algorithm 9.10 The LocalTwoPassSplayTT algorithm, combining our ideas to improve TwoPassSplayTT.

```
procedure LocalTwoPassSplayTT($v$)
    while $v$ has parent $p$ do
        if $p$ has parent $g$ then
            if can_splay_step($v$) then
                splay_step($v$)
            else if $p$ is separator then                ▷ $p$ is br. node
                splay_step($p$)                   ▷ Valid, since $p, g$ are sep.
            else                                         ▷ $g$ is br. node
                $g_2$ ← parent($g$)
                if $g_2$ is sep. then
                    splay_step($g$)              ▷ Valid, since $g, g_2$ are sep.
                else
                    rotate($g$)                       ▷ Makes $g$ not a sep.
        else
            rotate($v$)
```

of section 9.3.5, only three out of the four rotations in the two `splay_step`s are necessary. It turns out that removing that unnecessary rotation is accomplished by replacing `splay_step`(p) with `rotate`(g). Thus, LocalTwoPassSplayTT is precisely to the improvement of GreedySplayTT discussed before. We have

Proposition 9.36. *The amortized number of rotations w.r.t.* $2 \cdot \Phi$ *performed by* LocalTwoPassSplayTT *when accessing a node* v *is at most* $18(\phi(T^*_v) - \phi(T_v)) - 11$*, where* T *is the search tree before the access, and* T^* *is the search tree afterwards.*

10. Dynamic forests with 2-cut STTs

In this chapter, we use the algorithms from chapter 9 to implement a *dynamic forest* data structure. The goal is to maintain a forest (and associated data like edge weights) under edge insertions and deletions.

This is a well-known problem with a forty-year history. Sleator and Tarjan [ST83] first introduced a data structure (commonly called *link-cut trees*) for this task, with worst-case running time $\mathcal{O}(\log n)$ per operation, where n is the number of vertices in the forest. The same authors proposed a simplified amortized variant of the data structure using their Splay trees [ST85b] (see chapter 8). Several alternative data structures have since been proposed, including *topology trees* [Fre85], *ET-trees* [HK99, Tar97], *RC-trees* [ABH$^+$04] and *top trees* [AHLT05, TW05, HRR23]. Top trees are more flexible and thus more widely applicable than link-cut trees (see also section 4.7). However, an experimental evaluation by Tarjan and Werneck [TW10] suggests that link-cut trees, while less flexible, are faster then top trees by a factor of up to four, likely due to their relative simplicity.

Dynamic forest data structures have a large number of applications. Link-cut trees in particular have been used as a key ingredient in algorithms and data structures including, but not limited to: minimum cut [Kar00], maximum flow [ST83, GT88], minimum-cost flow [KN13], online minimum spanning forests [Fre85, EIT$^+$92, EGIN97, CFPI10], online lowest common ancestors [ST83, HT84], online planarity testing [DBT96], and geometric stabbing queries [AAK$^+$12, KMT03].

Our STT-based data structures can serve as a drop-in replacement for link-cut trees. We experimentally compare them with various link-cut tree implementations in section 10.9.

unrooted dynamic forest

We focus on maintaining *unrooted* forests. More precisely, an *unrooted dynamic forest* data structure maintains an edge-weighted forest and supports the following three operations:

- Link(u, v, w) – Adds an edge between the vertices u and v with weight w. Assumes this edge did not exist beforehand.
- Cut(u, v) – Removes the edge between u and v. Assumes this edge existed beforehand.
- PathWeight(u, v) – Returns the sum of weights of edges on the path between u and v, or $\perp$ if u and v are not in the same tree.

We assume that the vertex set is fixed, and initially the forest has no edges. As weights, our implementation allows arbitrary commutative monoids. For example, when using edge weights from $(\mathbb{N}, \max)$, the PathWeight(u, v) method returns the maximum edge weight on the path between u and v. Additional operations like increasing the weight of each edge on a path, or maintaining vertex weights and certain related properties are also possible to implement, but omitted for simplicity.

Rooted vs. unrooted forests. Some applications, like the maximum flow algorithms mentioned above, require maintaining *rooted* forests. In that case, adding arbitrary edges is not allowed; the LINK(u, v) operation requires that u is the root of its tree, and makes v the parent of u. There are several useful queries specific to rooted trees; here, we consider the following ones.

- FINDROOT(v) – Returns the root of the underlying tree containing v.
- LCA(u, v) – Returns the lowest common ancestor of u and v, or $\perp$ if u and v are not in the same tree.
- EVERT(v) – Makes v the root of its tree.

The basic variant of link-cut trees maintains rooted forests. The EVERT(v) is mainly useful to support unrooted forests, since it enables arbitrary LINKs. While asymptotic performance is not affected, EVERT does come with additional bookkeeping that may impact performance in practice.

For our data structures, the opposite is true: they "natively" implement unrooted forests, and maintaining rooted forests is possible only with some overhead. We discuss further similarities and differences between link-cut trees and our data structures in section 10.7.

Dynamic forests using STTs. A common property of our dynamic STT algorithms (MOVETOROOTTT and the three SPLAYTT variants) is that after finding a node, it is brought to the root. This property is very useful to implement self-adjusting dynamic forests. In fact, in this chapter, we describe a modular framework consisting of the following three parts:

(i) a basic implementation of 2-cut STTs, including rotations;

(ii) a routine called ACCESS that brings an STT node to the root via rotations; and

(iii) an implementation of the operations LINK, CUT, and PATHWEIGHT, as well as FINDROOT, LCA, and EVERT, based on ACCESS.

The ACCESS routine will be one of our four dynamic STT algorithms. We require ACCESS to satisfy a certain property called *stability* (essentially, it should not move the previous root too much; see section 10.2 for more details). Nonetheless, we expect that most future algorithms for the dynamic STT model can be adapted for our purposes.

Implementation and evaluation. We implemented our data structures in the *Rust* programming language.[1] The modularity described above is achieved using generics, resulting in an easily extendable library. We will reference the Rust code throughout this chapter.

For comparison, we also implemented the amortized variant of Tarjan and Sleator's link-cut trees [ST83, ST85b], and some very simple linear-time data structures. Furthermore, we implemented a simplified version of our data structure in the *C++* programming language,[2] which is significantly faster. We experimentally compare all our implementations, and some external ones, in section 10.9.

[1]Source code found at `https://github.com/berendsohn/stt-rs`, git tag `thesis`.
[2]Source code found at `https://github.com/berendsohn/stt-cpp`, git tag `thesis`.

Organization of this chapter. In section 10.1, we present a basic data structure that maintains a 2-cut search tree on a *fixed* tree under rotations. In sections 10.2 to 10.4, we show how to implement the dynamic forest operations, using multiple 2-cut STTs and assuming a black-box Access implementation. In section 10.5, we discuss the specifics of the implementation of our four Access algorithms, and in section 10.6, we analyze the overall running time of all operations. In section 10.7, we compare link-cut trees to our algorithms. In section 10.8, we briefly discuss the Rust and C++ implementations, and in section 10.9, we discuss their experimental evaluation.

10.1. Implementing a 2-cut STT data structure

A naive way of maintaining a search tree T (not necessarily 2-cut) is as a rooted tree, where each node v has a pointer to its parent, a pointer to each of its children, and a list of boundary nodes $\partial(T_v)$. To implement a rotation of some node v with its parent p, we need to identify the (possibly non-existent) child node of v that changes parents. Recall that, by definition (section 2.2.2), this is precisely the child c with $p \in \partial(T_c)$. Hence, the information specified above suffices to execute rotations, but searching through the list of child nodes may require up to linear time, just to execute a single rotation.

It turns out we can maintain a 2-cut STT under constant-time rotations with only the following three pointers per node v.

- `parent`(v): The parent node of v, or $\perp$ if v is the root.
- `dsep_child`(v): The unique child of v that is a direct separator node, or $\perp$ if v has no such child.
- `isep_child`(v): The unique child of v that is a indirect separator node, or $\perp$ if v has no such child.

Note that checking `dsep_child`(`parent`$(v)) = v$ allows us to test whether v is a direct separator or not, and similarly for indirect separators. The data structure is found in `stt/src/twocut/basic.rs` in the source code.

10.1.1. Unique representation of the underlying tree

We now show that the three pointers `parent`, `dsep_child`, `isep_child` are sufficient to uniquely represent the underlying tree. The parent pointers tell us the structure of the search tree, and the child pointers tell us which nodes are direct or indirect separators. We first show that this uniquely determines the boundaries of subtrees.

Lemma 10.1. *Given a 2-cut search tree T on a tree and the pointers* `parent`, `dsep_child`, `isep_child` *for each node, we can determine $\partial(T_v)$ for each node v.*

Proof. For the root r, we always have $\partial(T_r) = \emptyset$. If v is not a separator and not the root, then $\partial(T_v)$ contains only the parent of v. If v is a direct separator, then $\partial(T_v)$ consists of the parent and grandparent of v.

Now consider an indirect separator node v with parent p and grandparent g. Observation 2.7 implies that $\partial(T_v) = \{p, x\}$, where $x \in \partial(T_p)$. Since v is an indirect separator node, $x \neq g$. But $g \in \partial(T_p)$, so x must be the remaining node in $\partial(T_p) \setminus \{g\}$. This observation allows us to determine all subtree boundaries in a top-down fashion. □

Once we have determined subtree boundaries, we can determine the edges of the underlying tree using the following lemma.

Lemma 10.2. *Let T be a search tree on a tree G. Let $u, v \in V(T)$ such that u is an ancestor of v. Then $\{u, v\} \in E(G)$ if and only if $u \in \partial(T_v)$, but $u \notin \partial(T_c)$ for each child c of v.*

Proof. If $u \notin \partial(T_v)$, then there is no edge in G between u and $V(T_v)$, so, in particular, $\{u, v\} \notin E(G)$. Now suppose $u \in \partial(T_v)$. Since G is a tree and $G[T_v]$ is connected, there must be exactly one edge $\{u, x\}$ between u and $V(T_v)$. If $u \in \partial(T_c)$ for some child c of v, then $x \in V(T_c)$, so $\{u, v\} \notin E(G)$. Otherwise, we have $x \notin V(T_c)$ for each child c of v, and hence $x = v$. Thus, in each case, the claimed equivalence holds. □

By lemma 2.3, all edges of the underlying graph are between ancestor and descendant in the search tree, so lemma 10.2 indeed determines all edges. Thus, our representation indeed uniquely determines the underlying tree.

10.1.2. Rotations

We now show how to implement rotations in our data structure. See algorithm 10.11 for pseudocode, or `stt/src/twocut/basic.rs` in the source code.

Lemma 10.3. *Given a node v in an STT T, represented as described at the start of the section, we can rotate v with its parent in $\mathcal{O}(1)$ time.*

Proof. Let $p = \texttt{parent}(v)$, let $g = \texttt{parent}(p)$, and let $c = \texttt{dsep\_child}(v)$ (g and/or c may be $\perp$). We denote by T' the tree after the rotation, and by $\texttt{parent}'(\cdot)$, $\texttt{dsep\_child}'(\cdot)$, $\texttt{isep\_child}'(\cdot)$ the correct respective pointers in T'. In the following, we frequently make use of the fact that each node has at most one direct and at most one indirect separator child (lemma 9.5).

For the parent pointers, we have $\texttt{parent}'(v) = g$, $\texttt{parent}'(p) = v$, and, if $c \neq \perp$, additionally $\texttt{parent}'(c) = p$ (see lines 5 to 8 in algorithm 10.11).

If $g \neq \perp$, we may need to adjust its child pointers. Observe that $V(T'_v) = V(T_p)$, so $\partial(T'_v) = \partial(T_p)$. Thus, v is an (in)direct separator child in T' if and only if p was an (in)direct separator child in T. One of $\texttt{dsep\_child}'(g)$ and $\texttt{isep\_child}'(g)$ may accordingly change from p to v (lines 9 to 13).

We now consider the child pointers of p. Note that p gains a new parent (v) and keeps all other ancestors, loses a child (v) and possibly gains a child (c). First, we have $\texttt{dsep\_child}'(p) = c$, since c is the unique node with $\partial(T_c) = \{v, p\}$ if such a node exists; otherwise, $\texttt{dsep\_child}'(p) = \perp = c$ (line 19).

If p has a (direct or indirect) separator child $y \neq v$ in T, then y clearly still is a separator in T', and $v \notin \partial(T_y) = \partial(T'_y)$, so $\texttt{isep\_child}'(p) = y$ (lines 14 to 16). If y does not exist, then p has no separator child in T. Since p does not gain any children in T' besides c, this means that $\texttt{isep\_child}'(p) = \perp$ (lines 17 to 18).

Now consider the child pointers of v. Note that v loses its parent and keeps all other ancestors, gains a child (p) and possibly loses a child (c). If $g = \perp$, then v is the root of

Algorithm 10.11 Rotating a node with its parent.

```
procedure ROTATE(v)
    $p \leftarrow$ parent($v$)
    $g \leftarrow$ parent($p$)
    $c \leftarrow$ dsep_child($v$)
    parent($p$) $\leftarrow v$
    parent($v$) $\leftarrow g$
    if $c \neq \perp$ then
        parent($c$) $\leftarrow p$
    if $g \neq \perp$ then
        if dsep_child($g$) $= p$ then
            dsep_child($g$) $\leftarrow v$
        else if isep_child($g$) $= p$ then
            isep_child($g$) $\leftarrow v$
    $x \leftarrow$ dsep_child($p$)
    if $x \neq \perp$ and $x \neq v$ then
        isep_child($p$) $\leftarrow x$
    else if isep_child($p$) $= v$ then
        isep_child($p$) $\leftarrow \perp$
    dsep_child($p$) $= c$
    if $g \neq \perp$ then                                  ▷ p was not the root
        if $x \neq v$ then                                  ▷ p separates v and g
            dsep_child($v$) $= p$
        else                                                ▷ v separates p and g
            dsep_child($v$) $=$ isep_child($v$)
            if $p$ was a separator node before the rotation then
                isep_child($v$) $\leftarrow p$
            else                                ▷ v separates all ancestors from p
                isep_child($v$) $\leftarrow \perp$
    else                                                    ▷ p was the root
        dsep_child($v$) $= \perp$
    if $c \neq \perp$ then
        Swap dsep_child($c$) and isep_child($c$)
```

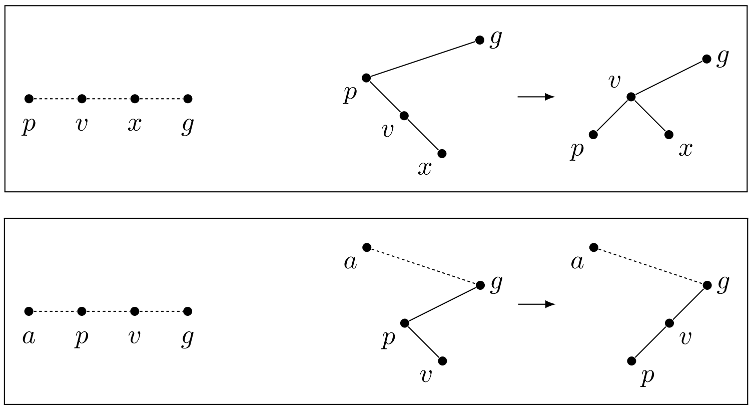


Figure 10.1.: Two special cases for a rotation at v: If v is a direct separator in T and has a direct separator child x in T' (top); if v is a direct separator and its parent p is a separator (bottom). The path in the underlying graph is shown to the left.

T', and thus $\texttt{dsep\_child}'(v) = \texttt{isep\_child}'(v) = \texttt{isep\_child}(v) = \perp$ (line 30). Suppose $g \neq \perp$. Since T is 2-cut, v, p, g lie on a common path (using lemma 9.6). The vertex g cannot lie between v and p on this path, otherwise v and p would be in different subtrees of g in T.

- If v is not a direct separator in T, then p separates v from g. Thus, we have $\texttt{dsep\_child}'(v) = p$. If $y = \texttt{isep\_child}(v) \neq \perp$, then y is still a separator child of v. Further $\partial(T'_y) = \partial(T_y) \subseteq \partial(T_v) \cup \{v\}$. Since p separates v from g, we have $g \notin \partial(T_v)$, and thus $g \notin \partial(T_y)$. This means that y is not a *direct* separator child, and thus $\texttt{isep\_child}'(v) = \texttt{isep\_child}(v) = y$.

  Since v only gains p as a child, no other nodes can become the indirect separator child of v. Thus $\texttt{isep\_child}'(v) = \texttt{isep\_child}(v)$, even if $\texttt{isep\_child}(v) = \perp$.

- If v is a direct separator in T, then $\partial(T_v) = \{p, g\}$ and v lies on the path between p and g. We claim that $\texttt{dsep\_child}'(v) = \texttt{isep\_child}(v)$ (line 24). Indeed, if $x = \texttt{dsep\_child}'(v) \neq \perp$, then $\partial(T'_x) = \{v, g\}$. Since v is on the path between p and g, we have $x \neq p$ (see figure 10.1, top), so x must already have been a child of v in T, and $\partial(T_x) = \partial(T'_x) = \{v, g\}$, so $x = \texttt{isep\_child}(v)$. Conversely, if $y = \texttt{isep\_child}(v) \neq \perp$, then y is still a child of v in T'. Further $\partial(T'_y) = \partial(T_y) \subseteq \partial(T_v) \cup \{v\} \setminus \{p\} = \{v, g\}$, implying that $\texttt{dsep\_child}'(v) = y$.

  To determine $\texttt{isep\_child}'(v)$, consider the following two cases.

  - If p is a separator node in T, then $\partial(T_p) = \{g, a\}$, where a is some ancestor of g. Lemma 9.1 implies that p separates g from a. Since v separates g from p, the underlying tree has a path containing a, p, v, g, in that order (see figure 10.1, bottom). Hence, p lies on the path between a and v, so $\partial(T'_p) = \{v, a\}$ and thus $\texttt{isep\_child}'(v) = p$ (line 26).
  - If p was a 1-cut node in T, then we claim that $\texttt{isep\_child}'(v) = \perp$ (line 28). Suppose otherwise that $x = \texttt{isep\_child}'(v) \neq \perp$. Then $\partial(T'_x) = \{v, a\}$, where a

is some ancestor of g. This implies $a \in \partial(T'_v) = \partial(T_p)$. But then $\partial(T_p) = \{g, a\}$, contradicting the assumption.

Finally, consider a separator child x of c. Since c swapped parent (v) and grandparent (p), if x was a direct separator in T, it is an indirect separator in T', and vice versa. Hence, we have $\texttt{isep\_child}'(c) = \texttt{dsep\_child}(c)$ and $\texttt{dsep\_child}'(c) = \texttt{isep\_child}(c)$ (line 32).

All nodes other than v, p, g, c do not gain or lose children and do not change parent, hence all remaining pointers are the same in T and T'. □

10.2. Linking and cutting

In this section, we show how to implement the operations Link and Cut to add and remove edges.[3] The underlying forest G is maintained as a collection of 2-cut STTs, which we call a *search forest*. Since we do not allow adding and removing nodes, we can maintain all nodes in a fixed-size array. The structure of each STT is represented by the node pointers described in section 10.1.

We assume that we have a black-box procedure Access(v) that, given a node v, brings it to the top of its tree with some sequence of rotations. For now, let us assume another procedure SubAccess(v) that (i) makes v the child of the root via a sequence of rotations and (ii) leaves the previous root in place.

See algorithms 10.12 and 10.13 for pseudocode of Link and Cut. Note that we ignore the supplied weight w in Link for now.

We now argue the correctness of the two procedures. Below, G and G' denote the underlying forest before and after the operation.

- Consider a call Link(u, v, w). Let F be the search forest after the two calls to Access, and let F' the search forest after Link. If we only consider parent pointers, then F' is clearly a valid search forest on G'. It remains to show that child pointers are still valid. For this, observe that for every node $x \in V(F) \setminus \{u\}$, we have $\partial(F_x) = \partial(F'_x)$, and we have $\partial(F_u) = \emptyset$, $\partial(F'_u) = \{v\}$. Thus, no node becomes a separator child or stops being one, and no direct separator node becomes an indirect one or vice versa. Thus, all child pointers stay valid.

  Observe that the call Access(v) (line 4) is not necessary for correctness. However, it is important for the complexity analysis in section 10.6.

[3]See `stt/src/twocut/mod.rs`

Algorithm 10.12 The Link operation.

```
1: procedure Link($u, v, w$)
2:     ▷ Assume $\{u, v\} \notin E(G)$
3:     Access($u$)
4:     Access($v$)
5:     parent($u$) ← $v$
```

Algorithm 10.13 The Cut operation.

```
1: procedure Cut($u, v$)
2:     ▷ Assume $\{u, v\} \in E(G)$
3:     Access($v$)
4:     SubAccess($u$)
5:     parent($u$) ← $\perp$
```

- Now consider a call Cut(u, v). Again, let F be the search forest after the calls to Access and SubAccess, and let F' be the search forest after Cut. It is clear that v is the parent of u in F. Thus, setting `parent`$(u) \leftarrow \perp$ correctly removes the edge $\{u, v\}$ from the underlying forest. Again, the boundaries of rooted subtrees other than T_v do not change between F and F', implying that no further pointer changes are necessary to make F' valid.

The running time of both operations is clearly dominated by the calls to Access and SubAccess.

10.2.1. Stable Access implementations.

It is possible to avoid implementing the SubAccess procedure if Access satisfies the following property. An Access implementation is called *stable* if, in a search tree produced by Access(v), the depth of the previous root r is bounded by some constant, and all nodes on the root path of r are 1-cut.

stable

Algorithm 10.14 gives an alternative implementation of Cut using stable Access. We only need to show that v is the parent of u in the forest F after the two calls to Access, then the analysis above works. Indeed, stability implies that u is 1-cut in F. Since v is an ancestor of u and $\{u, v\} \in E(G)$, we have $\partial(F_u) = \{v\}$, implying that v is the parent of u.

Since Link does not use SubAccess, it need not be modified.

10.3. Maintaining edge weights

In this section, we show how to maintain edge weights in 2-cut STTs under rotations and implement the PathWeight operation.[4] We start with a simpler variant that requires the edge weights to form a commutative *group* instead of a monoid.

10.3.1. Group weights

Let $(W, +)$ be a commutative group with identity element 0. We use the subtraction operator $-$ in the usual way. Let $W' = W \cup \{\infty\}$, and define $w + \infty = \infty + w = \infty$ for all $w \in W$. (We do not define an inverse for ∞, so we cannot subtract ∞).

Let F be a 2-cut search forest on a forest G with edge weights from $(W, +)$. The weight of a path in F is defined as the sum of the weights of its edges. For two vertices $u, v \in V(G)$ that belong to the same tree, let the *distance* $d(u, v) \in W'$ denote the weight of the unique path between u and v. Define $d(\perp, u) = d(u, \perp) = \infty$ for $u \in V(G) \cup \{\perp\}$.

distance

For each node v, we store a field `pdist`(v) indicating the distance between v and the parent of v in F. Consistently with the definition of $d(\cdot, \cdot)$, we let `pdist`$(v) = \infty$ if v is the root of its tree.

Since we start with a forest with no edges, we can initially set `pdist`$(v) \leftarrow \infty$ for each node v. We now show how to update `pdist` when the forest is modified.

[4] In the source code, the weight update procedures are found in `stt/src/twocut/node_data.rs`; implementations of PathWeight (for stable and non-stable Access) are found in `stt/src/twocut/mod.rs`

Algorithm 10.14 CUT with stable ACCESS.

```
procedure CUT($u, v$)
    ▷ Assume $\{u, v\} \in E(G)$
    ACCESS($u$)
    ACCESS($v$)
    parent($u$) ← $\perp$
```

Rotations. Consider a rotation of v with its parent p in a search tree T, resulting in a search tree $T' = \text{rot}(T, v, p)$. We only need to update `pdist`(v) for nodes that change parents. That includes v and p as well as the direct separator child c of v, if it exists. The parents of these nodes are all from the set $\{v, p, g\}$, where g is the parent of p in T (possibly $g = \perp$). We now show that from the field `pdist` in T, we can compute all (relevant) mutual distances between the nodes c, v, p, g.

Lemma 10.4. *Let v be a node in a 2-cut search tree T with edge weights from a commutative group $(W, +)$, equipped with the field* `pdist`. *Then, in constant time, we can compute the distance from v to the parent p and grandparent g of v, if either exists.*

Proof. If p exists, then clearly $d(v, p) =$ `pdist`(v). Now suppose g exists. If v is a separator child, then v is on the path between v and g. Hence, we have $d(v, g) = d(p, g) - d(v, p) =$ `pdist`$(p) -$ `pdist`(v). If v is *not* a separator child, then p is on the path between v and g, and thus $d(v, g) = d(v, p) + d(p, g) =$ `pdist`$(v) +$ `pdist`(g). ☐

We now first compute all mutual distances between c, v, p, g using lemma 10.4, except for $d(c, g)$. Then, we execute the rotation, and then update the `pdist` fields of c, v, p using the new `parent` field and the previously computed distances (observe that the new parent of c is p).

Algorithm 10.15 shows an improved implementation that avoids unnecessary calculations. We omit the formal line-by-line correctness proof.

Linking and cutting. At the end of LINK(u, v, w), we make v the parent of u. We can simply set `pdist`$(u) \leftarrow w$ here. Similarly, at the end of CUT(u, v), the node u is removed from its parent, and we set `pdist`$(u) \leftarrow \infty$.

Computing path weight. We now turn to the implementations of PATHWEIGHT(u, v). If SUBACCESS is available, the implementation is straight-forward: Simply make u a child of v and return `pdist`(u), if u and v are indeed in the same tree.

Now assume ACCESS is stable. We then can implement PATHWEIGHT(u, v) without using SUBACCESS (see algorithm 10.16). First, we call ACCESS(u), and then ACCESS(v). Afterwards, we follow parent pointers to check whether v is the root of the search tree containing u. If no, we return $\perp$. If yes, we return the sum $\sum_{x \in \text{Path}(u) \setminus \{v\}}$ `pdist`(x).

We now argue that this procedure is correct. Let F be the search forest after the two calls to ACCESS. Clearly, v is the root of its search tree in F. If u is in a different search tree, the algorithm correctly returns $\perp$.

Algorithm 10.15 Updating `pdist` fields directly before a rotation (group weights).

```
procedure UpdateWeights($v$)
    $p \leftarrow$ parent($v$)
    $g \leftarrow$ parent($p$)
    $c \leftarrow$ dsep_child($v$)
    pdist'($p$) $\leftarrow$ pdist($v$)
    if $v$ is a direct separator then
        pdist'($v$) $\leftarrow$ pdist($p$) $-$ pdist($v$)
    else
        pdist'($v$) $\leftarrow$ pdist($v$) $+$ pdist($p$)
    if $c \neq \bot$ then
        pdist'($c$) $\leftarrow$ pdist($v$) $-$ pdist($c$)
    Update pdist with pdist'
```

Algorithm 10.16 Implementing PathWeight with stable Access

```
procedure PathWeight($u, v$)
    Access($u$)
    Access($v$)
    $x \leftarrow u$
    $w \leftarrow 0$
    while parent($x$) $\neq \bot$ do
        $w \leftarrow w +$ pdist($x$)
        $x \leftarrow$ parent($x$)
    if $x = v$ then
        return $w$
    else
        return $\bot$
```

Otherwise, u is a descendant of v. Let $u = u_1, u_2, \dots, u_k = v$ be the path from u to v in the search forest F. Stability of Access implies that $u_1, u_2, \dots, u_{k-1}$ are all 1-cut. This means that for each $i \in [k-2]$, the path from any node in $V(T_{u_i})$ to any node outside of $V(T_{u_i})$ must contain u_{i+1}. In particular, the path from u to u_{i+2} contains u_{i+1}. Thus, by induction, the path in the *underlying forest* from $u = u_1$ to $v = u_k$ contains $u_1, u_2, \dots, u_k$, in that order, and the total weight of this path is $\sum_{i=1}^{k-1} d(u_i, u_{i+1}) = \sum_{i=1}^{k-1} \texttt{pdist}(u_i)$.

Observe that both implementations need constant time outside of calls to Access and SubAccess. In particular, in algorithm 10.16, the depth of u is bounded after the Access calls, by stability.

10.3.2. Monoid weights

Now suppose the edge weights form a commutative monoid $(W, +)$. Recall that there is no inverse in a monoid, so subtraction is not possible, which breaks the proof of lemma 10.4.

As before, let F be a 2-cut search forest on a forest G with edge weights from $(W, +)$, and let $d(u, v) \in W \cup \{\infty\}$ denote the distance between two vertices $u, v \in V(G) \cup \{\bot\}$. We now store two fields for every node.

- `pdist`(v) is the distance between v and its parent, as before.
- `adist`(v) is ∞ if v is not a separator. If v is a separator, then `adist`(v) is the distance between v and the node $x \in \partial(T_v)$ that is *not* the parent of v.

Together, `pdist`(v) and `adist`(v) store the distance from v to each node $x \in \partial(T_v)$.

Rotations. We use the following analogue of lemma 10.4.

Lemma 10.5. *Let v be a node in a 2-cut search tree T with edge weights from a commutative monoid $(W, +)$, equipped with the fields* `pdist` *and* `adist`*. Suppose a rotation at some node v is allowed. Let p be the parent of v, let g be the grandparent of v, and let a be the ancestor of p such that $a \in \partial(T_p)$ and $a \neq g$. Non-existing nodes are $\bot$.*

Then we can compute all mutual distances between v, p, g, a in constant time.

Proof. First observe that for al $x \in V(T)$ and $y \in \partial(T_x)$, we can compute $d(x, y)$ using either `pdist`(x) or `adist`(x), depending whether $y =$ `parent`(x) or not. This only leaves us with two distances to compute.

First, we need to compute $d(v, g)$ if $g \notin \partial(T_v)$, i.e., if v is not a separator. In that case, we know that p is on the path between v and g, so $d(v, g) = d(v, p) + d(p, g) =$ `pdist`$(v) +$ `pdist`(p).

Second, we need to compute $d(v, a)$, assuming that $a \neq \bot$. Observe that v must be a separator, since p is a separator and the rotation is allowed (lemma 9.3). If v is an indirect separator, then $a \in \partial(T_v)$, so $d(v, a) =$ `adist`(v). If v is a direct separator, then g, v, p, a must lie on a common path in G, in that order (recall that p separates g from a). Thus $d(v, a) = d(v, p) + d(p, a) =$ `pdist`$(v) +$ `adist`(p). □

Consider now a rotation between v and p in a search tree T, resulting in the search tree $T' = \mathrm{rot}(T, v, p)$. Let c be the direct separator child of v, and let p, g, a be defined as in lemma 10.5.

We start by computing all mutual distances between v, p, g, a with lemma 10.5, and then execute the rotation. Observe that only c, v, p change boundaries and/or parents and thus need to be updated.

First observe that $\partial(T'_c) = \partial(T_c) = \{v, p\}$, and c switches parents from v to p. Thus, we can simply swap `pdist`(c) and `adist`(c). Further observe that we can compute the boundaries of v and p as follows:

- Since $V(T'_v) = V(T_p)$, we have $\partial(T'_v) = \partial(T_p) \subseteq \{g, a\}$. We can distinguish between the cases $\{g, a\}$ and $\{g\}$ by checking whether v is a separator or not.
- We have $\partial(T'_p) \subseteq \{v, g, a\}$ and $v \in \partial(T'_p)$. We can distinguish between the cases $\{v, g\}$, $\{v, a\}$, and $\{v\}$ by checking if p is a direct separator, an indirect separator, or not a separator.

Now that we have the boundaries of v and p, we can use the `parent` pointer to distinguish between parent and other boundary node. Thus, we can update the fields `pdist` and `adist` based on the distances computed before the rotation. See `stt/src/twocut/node_data.rs` in the source code for an optimized version of the procedure.

Linking and cutting. The `pdist` field is updated as described in section 10.3.1. We now argue that `adist` does not need to be updated for either operation. Indeed, an update is only necessary when some node gains or loses a boundary vertex besides the parent. LINK(u, v, w) makes v the parent of u. Let T be the resulting search tree. Since the only edge between $V(T_u)$ and $V(T - T_u)$ is $\{u, v\}$, we have $\partial(T_u) = \{v\}$ and no node other than u changes boundary. For CUT, a symmetric argument holds.

Computing path weight. This can be implemented exactly as in the group case, since weights are only added, never subtracted (see algorithm 10.16).

Remark. We require the weights to form a *commutative* monoid, since we use commutativity of $d(\cdot,\cdot)$. It should be easy to support non-commutative monoids, by using two fields $\texttt{pdist}_1$ and $\texttt{pdist}_2$, one for $d(v,p)$ and one for $d(p,v)$, and similarly duplicating `adist`. In the above procedures, we then have to carefully choose the right direction. To keep the implementation simple, and since the author is not aware of any applications that require non-commutative groups or monoids, we restrict ourselves to commutative monoids.

10.4. Rooted forests

In contrast to link-cut trees, our implementation of STTs cannot represent rooted trees without modification. Consider a call to FindRoot(v) when v is the root of its STT. Since v has no separator children, both its child pointers are $\perp$, thus we cannot navigate to the root of the underlying tree (or *any* node other than v, for that matter). In this section, we show how we can implement FindRoot(v) and other rooted-tree operations using extra data.[5]

Let G be a tree with a designated root r. Let T be a search tree on G. We store (a pointer to) the root r in each node on the root path of r in T. Formally, we maintain the following property for each node $v \in V(T)$:

$$\texttt{droot}(v) = \begin{cases} r, & \text{if } r \in V(T_v) \\ \perp, & \text{otherwise} \end{cases}$$

Implementing FindRoot(v) is now trivial: Call Access(v), and then return $\texttt{droot}(v)$.

We now describe how to update $\texttt{droot}(v)$ under rotations. Consider a rotation of v with its parent p. Let T, T' denote the search tree before and after the rotation and let `droot`, $\texttt{droot}'$ denote the respective values before and after the rotation.

Observe that only $\texttt{droot}(v)$ and $\texttt{droot}(p)$ may change. Since $V(T'_v) = V(T_p)$, we have $\texttt{droot}'(v) = \texttt{droot}(p)$. For $\texttt{droot}'(p)$, consider the following cases.

- If $\texttt{droot}(p) = \perp$, then $\texttt{droot}'(p) = \perp$, since p gains no new descendants with the rotation.
- If $\texttt{droot}(p) \neq \perp$ and $\texttt{droot}(v) = \perp$, then r is in $V(T_c)$ for some child $c \neq v$ of p. Observe that c is still a child of p in T', hence $\texttt{droot}'(p) = \texttt{droot}(p)$.
- Finally, suppose $\texttt{droot}(p) \neq \perp$ and $\texttt{droot}(v) \neq \perp$. Let c be the direct separator child of v, and recall that the rotation makes c a child of p. If c does exist and $\texttt{droot}(c) \neq \perp$, then r is in $V(T_c) = V(T'_c)$ and hence in $V(T'_p)$, so $\texttt{droot}'(p) = \texttt{droot}(p)$. Otherwise, $r = v$ or $r \in V(T_{c'})$ for some child $c' \neq c$ of v in T, hence $r \notin V(T'_p)$ and thus $\texttt{droot}'(p) = \perp$.

We now sketch the implementations of the remaining operations. We refer to the code in `stt/src/twocut/rooted.rs` for more details.

Link(u, v) works as described in section 10.2, except that afterwards we set $\texttt{droot}(u) \leftarrow \perp$. Note that, by assumption, u was the root of its underlying tree before the operation.

[5] See `stt/src/twocut/rooted.rs`

After bringing u to the root, each proper descendant x of u has $\texttt{droot}(x) = \bot$. Then, u becomes the child of v, so it is no longer the underlying tree root.

CUT(v) now only takes one parameter. If the parent u of v (in the underlying tree) is known, we can simply call ACCESS(v) and SUBACCESS(u), then detach u from v and set $\texttt{droot}(v) \leftarrow v$. However, we do not assume that u is known. To find u, after calling ACCESS(v), we first call SUBACCESS(r) (note that $r = \texttt{droot}(v)$). Then, we can find u by first moving to the direct separator child of r, and then following indirect separator child pointers as long as possible. This moves along the underlying path from r to v (possibly skipping nodes), stopping at u. If r has no direct separator child, then $u = r$.

For LCA(u, v), we first call ACCESS(v) and SUBACCESS(u). Some simple checks determine whether u is an ancestor of v or vice versa. Otherwise, the LCA has to be in the direct separator child x of u. Now, we check if $\texttt{droot}(d) \neq \bot$ or $\texttt{droot}(i) \neq \bot$, where d and i are the direct and indirect separator children of x. If either is true, we repeat with $x \leftarrow i$, resp., $x \leftarrow d$. Otherwise, it can be seen that x must be the LCA of u and v. Calling ACCESS(x) at the end pays for following the root path of x via amortization.

Finally, EVERT(v) is implemented as follows. First, call ACCESS(v). We then need to set $\texttt{droot}(v) \leftarrow v$ and $\texttt{droot}(x) \leftarrow \bot$ for each node $x \neq v$. Observe that every node x with $\texttt{droot}(x) \neq \bot$ must be on the root path of $r = \texttt{droot}(v)$, hence we can make the necessary changes by following parent pointers from r. To pay for this, we call ACCESS(r) afterwards.

Remark. We used SUBACCESS above for simplicity. It is possible to implement all operations with only stable ACCESS; see the Rust implementation for more details.[6] As usual, the running time of each operation is dominated by calls to ACCESS.

10.5. Implementing Access

Our dynamic STT algorithms (MOVETOROOTTT and the three SPLAYTT variants) all readily work as implementations of ACCESS. This is obviously not the case for arbitrary dynamic STT algorithms, since the dynamic STG model does not require bringing an accessed node to the root. A more subtle problem is that the dynamic STG model allows pointer moves to arbitrary children, but our STT implementation does not store 1-cut children of nodes.

For our algorithms, we make the following observations about an access to node v.

- The algorithm first finds v by moving the pointer along the root path of v. In our data structure, we directly receive a pointer to the node v, so this part is not necessary.

- Afterwards, only nodes on the root path of v are touched. While, technically, pointer-moves to non-separator children are sometimes performed (e.g., going back to v after the first pass in TWOPASSSPLAYTT), we can instead jump directly to a node already stored in memory (usually v itself), and continue from there.

[6] `StableRootedDynamicForest` in `stt/src/twocut/rooted.rs`

- Rotations have running time $\mathcal{O}(1)$ in our STT implementation, and the cost of each algorithm is dominated by the number of rotations (plus one).

These observations together imply

Lemma 10.6. *Each of* MoveToRootTT*,* GreedySplayTT*,* TwoPassSplayTT*, and* LocalTwoPassSplayTT *can be used to implement* Access*. The running time of the respective implementation is proportional to the cost of the algorithm.*

As mentioned before, just using a black-box Access implementation is not quite sufficient for our purposes. SubAccess is not hard to implement using any one of our three algorithms: MoveToRootTT simply stops when the accessed node v is directly below the root, and the SplayTT algorithms likewise stop or perform a single final rotation at the end.[7]

However, SubAccess is not strictly necessary, since all our algorithms are stable, which we show now.

10.5.1. Stability

Given an STT T and nodes $v, x \in V(T)$, we define the property $P(T, v, x)$ as satisfied if

(a) all ancestors of x are 1-cut; and

(b) the root paths of v and x only intersect at the root of T.

We will show that throughout a call Access(v) with any one of our four algorithms, the invariant $P(T, v, r)$ holds, where r is the root of the tree at the start of the call. We need the following technical lemma.

Lemma 10.7. *Let T be a 2-cut STT and let $u, v, x \in V(T)$, such that*

(i) $u \prec_T v$; and

(ii) if u is a child of the root and $u \neq v$, then the child of u on the root path of v is 1-cut.

Then $P(T, v, x)$ implies $P(T', v, x)$, where $T' = \mathrm{rot}(T, u)$.

Proof. Let p be the parent of u in T. First, suppose that p is not the root. Then $P(T, v, x)$ implies that p is not on the root path of x. Hence, rotating at u may remove p from the root path of v, but does not change the root path of x. Thus, $P(T', v, x)$ holds.

Now suppose that p is the root of T. Then rotating at u adds u to the root path of x. We have $|\partial(T'_u)| = 0$, $|\partial(T'_p)| = 1$, and $|\partial(T'_y)| = |\partial(T_y)| = 1$ for every other node y on the root path of x. This proves part (a) of $P(T', v, x)$. In order to prove part (b), let v' be the child of u on the root path of v in T. Assumption (ii) implies that v' is 1-cut, so rotating at u does not change the parent of v' (lemma 2.10), and thus v' and p are both children of u in T'. Since v is a descendant of v' and x is a descendant of p, part (b) of $P(T', v, x)$ holds. $\square$

Lemma 10.8. MoveToRootTT *and all three* SplayTT *variants are stable.*

[7]See `stt/src/twocut/splaytt.rs` for all implementations of Access and SubAccess.

Proof. Let T be an STT with root r, and let T' be the result of calling ACCESS(v) (using any of the four algorithms). Observe that $P(T, v, r)$ trivially holds.

We now argue that our implementations of ACCESS(v) only perform rotations satisfying lemma 10.7 with $x = r$. Clearly, all rotations are performed on the root path of v. To see that (ii) holds, we need to consider the algorithms in more detail. We are only interested in rotations at a node $u \neq v$ that is the child of the root. Such a rotation only happens in the following circumstances.

- A ZIG-ZIG step at a grandchild u' of the root r, in any of the SPLAYTT variants. A ZIG-ZIG step only happens if u', u, r are on a path in G in that order, implying that u' is 1-cut before the rotation.
- A final ZIG step in the first pass of TWOPASSSPLAYTT, when bringing the final branching node to the root. In that case, u was a branching node before rotating it to the root, so by definition, the child of u on the root path of v is 1-cut.
- The single rotation in MOVETOROOTTT (algorithm 9.4, line 6) or LOCALTWO-PASSSPLAYTT (algorithm 9.10, line 13). This rotation is only applied to branching nodes; but u is the child of the root, so it is 1-cut and cannot be a branching node. Thus, this cannot actually happen.

By induction, lemma 10.7 implies that $P(T', v, r)$ also holds. It remains to show that the depth of r in T' is bounded. Since $P(T'', v, r)$ holds for every intermediate tree T'', the depth of r can only increase when a rotation involving the (current) root is performed. It is easy to see that each variant performs at most three rotations or `splay_step` calls that involve the root, so the final depth of r is at most six. We thus conclude that MOVETOROOTTT and all three SPLAYTT variants are stable. □

10.6. Running time analysis

In section 9.3, we showed static optimality of SPLAYTT. The proof does not work in the dynamic forest setting, and indeed the concept of static optimality makes no sense here, since there is no "static" reference STT. However, we can show that the working-set bound, and thus an $\mathcal{O}(\log n)$ amortized worst-case bound, holds.

For this, we extend the potential function Φ_w^{ST} (see section 9.3.1) to search forests: Given a search forest F and a weight function w on F, let $\Phi_w^{\mathrm{ST}}(F)$ be the sum of $\Phi_w^{\mathrm{ST}}(T)$ over all search trees T that constitute F.

Consider a SPLAYTT implementation of ACCESS. By lemma 10.6, the asymptotic running time of ACCESS is its *cost* in the dynamic search tree model. Thus, when we maintain a single STT under only ACCESS, the working-set bound holds by theorem 9.15.

Recall that all dynamic forest operations can be implemented using only ACCESS, and their running time is dominated by calls to ACCESS. However, we need to also consider the potential change of the dynamic forest operations outside of ACCESS.

Observe that PATHWEIGHT, FINDROOT, LCA, EVERT do not affect the potential outside of calls to ACCESS, and CUT can only *decrease* it. Thus, we only need to consider LINK.

Before proving the working-set bound, let us briefly show the easier $\mathcal{O}(\log n)$ worst-case bound.

Theorem 10.9. *Starting with a (rooted or unrooted) forest with n vertices and without edges, every* SplayTT *variant of our dynamic forest data structure performs m dynamic forest operations in time $\mathcal{O}((m+n)\log n)$.*

Proof. Use the potential function Φ_w^{ST} with $w = \mathbb{1}$. The amortized running time of each operation is at most $\mathcal{O}(\phi_w^{\mathrm{ST}}(V(G))) \subseteq \mathcal{O}(\log n)$ by lemma 9.20. Each Link can only increase the potential of a single node, and thus increases the total potential by no more than $\log n$. This gives us an overall $\mathcal{O}(m \log n)$ amortized running time.

It remains to consider the potential at the start and end. Clearly, the potential of any search forest is between zero and $n \log n$, thus the overall potential change is at most $n \log n$. Adding the potential change to the amortized running time yields the desired bound on the actual running time. □

We now proceed with the proof of the working-set bound. We need a slightly more detailed analysis of the Link operation.

Link(u, v) increases the node potential of v when attaching the search tree containing u to it (see algorithm 10.12). If F is the search forest directly before this (after calling Access(u) and Access(v)), and F' is the search forest after, then the potential increase is trivially $\phi^{\mathrm{ST}}(F'_v) - \phi^{\mathrm{ST}}(F_v)$. Hence, this final part of Link(u, v) satisfies lemma 9.20 in the same way Access(v) does. Adapting the proof of theorem 9.15 is now straight-forward.

Theorem 10.10. *Consider a sequence $X = (x_1, x_2, \dots, x_m)$ of dynamic forest operations, starting with a (rooted or unrooted) forest with n vertices and without edges. Let U_i be the up to two vertices that are parameters of x_i. For each $i \in [m]$, let $W_i = U_j \cup U_{j+1} \cup \dots \cup U_{i-1}$, where $j \in [m]$ is maximal such that $U_i \subseteq W_i$, or $j = 0$ if there is no such j.*

Every SplayTT *variant of our dynamic forest data structure performs X in time $\mathcal{O}(n \log n + \sum_{i=1}^{m} \log |W_i|)$.*

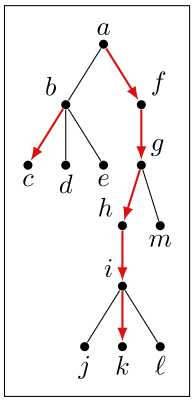
Proof. By the discussion above, we need to only consider the calls to Access made by the dynamic forest operations, and the final modification in Link, which we can treat as another Access for the purpose of this analysis.

It is easy to see that theorem 9.15 still holds in a search forest (the weight change in the proof does not increase the overall potential). Thus, if $Y = (y_1, y_2, \dots)$ is the sequence of accessed nodes, the amortized cost of y_i is $\mathcal{O}(\log |W'_i|)$, where W'_i is the set of elements accessed since the last time y_i was accessed (including y_i).

Now consider the i-th access, occurring in the j-th dynamic forest operation. Since every dynamic forest operation accesses all its parameter vertices, we have $W'_i \subseteq W_j$. Moreover, each dynamic forest operation performs a constant number of accesses, so the j-th operation's amortized running time is $\mathcal{O}(\log |W_j|)$. The desired bound follows. □

10.7. SplayTT vs. link-cut trees

Sleator and Tarjan's link-cut trees [ST83, ST85b] maintain a decomposition of the underlying rooted tree into paths, directed away from the root. There is a BST for each such



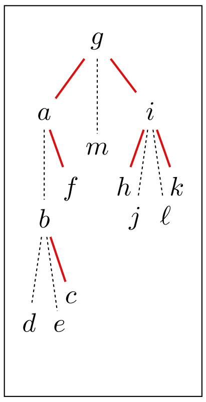


Figure 10.2.: A rooted tree G, decomposed into paths (left) and a link-cut tree T on G (right). The decomposition of G into directed paths is indicated by red edges. In the link-cut tree, BST edges are red, and non-BST edges are dashed.

path, and these BSTs are hierarchically arranged and connected via edges that are not covered by paths. Figure 10.2 shows an example. It is not hard to see that link-cut trees are essentially 2-cut STTs [BCI+20], if we disregard the different edge types and order of children.

Moving a node v to the root in a link-cut tree is performed roughly as follows, utilizing the classical BST Splay algorithm. First, for each BST B between v and the root, splay the node v_B to the root of B, where v_B is the lowest node in B that is an ancestor of v in the overall link-cut tree. This shortens the path from v to the root, such that every node on that path comes from a different BST. Then, an operation called *splice* is performed, which splits and merges BSTs until the path from v to the root is contained in a single BST. Finally, v is splayed to the root.

TwoPassSplayTT, in a way, works very similar. Disregarding the left-right order of BST nodes in the link-cut tree, TwoPassSplayTT performs almost the same rotations as link-cut trees. The main difference is that no path-decomposition is maintained; instead, SplayTT "automatically" detects a decomposition of the search path.

As discussed at the beginning of the chapter, link-cut trees "natively" maintain *rooted* forests. Each path in the decomposition can be seen as oriented towards the root, and this orientation is preserved by the left-right order in the corresponding BST. To support Evert (and hence arbitrary Links in unrooted forests), it must be possible to *reverse* paths and the corresponding BSTs. To preserve the $\mathcal{O}(\log n)$ amortized cost this has to be done lazily using a *reverse bit*, which complicates the implementation somewhat.

All in all, for maintaining unrooted forests, our STT-based framework is arguably conceptually simpler than link-cut trees, since no reverse bit is required an there is no need to explicitly maintain a (directed) path decomposition of the underlying forest. The main complexity lies in the implementation of the STT rotation primitive, which is easily separated and reused, simplifying the engineering of new variants. In contrast, variants of link-cut trees are somewhat restricted by the explicit decomposition into BSTs; for example, no equivalent of our GreedySplayTT algorithm for link-cut trees exists.

10.8. Practical implementation

The author implemented dynamic forest data structures in the *Rust* and *C++* programming languages. We first consider the fully-fledged Rust implementation, which is available at `https://github.com/berendsohn/stt-rs`.

Edge-weighted unrooted forests in Rust. We combine our basic 2-cut STT data structure (section 10.1) with one variant (stable/non-stable) of the LINK, CUT, and PATHWEIGHT procedures described in sections 10.2 and 10.3 and one of the ACCESS algorithms (GREEDYSPLAYTT, TWOPASSSPLAYTT, LOCALTWOPASSSPLAYTT, and MOVETOROOTTT). Overall, we obtain eight different implementations, denoted by (`Stable`) `Greedy Splay`, (`Stable`) `2P Splay`, (`Stable`) `L2P Splay`, and (`Stable`) `MTR`.

Further, we have `Link-cut`,[8] an implementation of the amortized variant of Sleator and Tarjan's link-cut trees [ST85b], where the handling of edge weights is similar to the way described in section 10.3.[9]

Finally, we have two linear-time data structures. `1-cut`[10] is a naive dynamic forest implementation that maintains a rooting of each tree (i.e., a 1-cut search tree on each tree). The dynamic forest operations are implemented as described above, where ACCESS(v) repeatedly rotates the root with one of its children until v is the root (see also proposition 8.5 in section 8.2). `Petgraph`[11] is a naive dynamic forest implementation using the Petgraph[12] library, which appears to be the most popular graph library for Rust at the time of writing. The other implementations were tested for correctness using `Petgraph` as a reference.

Rooted forests in Rust. The author also implemented data structures maintaining rooted forests without edge weights. We support LINK, CUT, FINDROOT and (depending on the experiment) EVERT. An extension of our STT-based data structures is sketched in section 10.4 and again yields eight STT-based variants.

`Link-cut` is the same link-cut tree implementation as above. If EVERT is not needed, we disable any checks and modifications of the reverse bit (though the slight space overhead remains). Finally, `Simple`[13] is a naive implementation that maintains the rooted forest explicitly via parent pointers.

Rust implementation notes. All unrooted (resp. rooted) implementations share a common interface (the Rust *traits* `DynamicForest`, resp. `RootedDynamicForest`) that is used by the experiments. Code is reused whenever possible through heavy use of generics. In particular, new weight types can be easily added, and weights can also be omitted entirely.

There are some differences between the pseudocode presented here and the actual Rust implementation. This is due to the fact that procedures like `can_rotate`(v) and

[8]Found in `stt/src/link_cut.rs` in the source code.

[9]Sleator and Tarjan only describe how to maintain vertex weights. Tarjan and Werneck [Wer06a] simulate edge weights by adding a vertex on each edge and maintaining vertex weights. We did not test this approach.

[10]Found in `stt/src/onecut.rs`

[11]Found in `stt/src/pg.rs`

[12]`https://crates.io/crates/petgraph`

[13]Found in `stt/src/rooted.rs`

`can_splay_step`(v) contain multiple `is_separator`($\cdot$) checks, which can cause an unnecessarily large number of calls to the `parent`($\cdot$) function, even though the parent and possibly further ancestors of v may be already known (consider, e.g., algorithms 9.4 and 9.6). Hence, we eliminated some of the additional calls by, e.g., introducing a function `is_separator_hint`(v, p), which is more efficient, but requires $p =$ `parent`(v) to be given.

We applied this principle liberally in all STT-based variants and our `Link-cut` implementation. The performance gains were relatively small, and we did not attempt any fine-tuning beyond this.

Unweighted unrooted forests in C++. For comparison, a C++ implementation is available at `https://github.com/berendsohn/stt-cpp`. This implementation does not support weights and rooted forests, only connectivity queries. We also omit the more complicated TwoPassSplayTT algorithm; only the stable variants of the remaining algorithms (GreedySplayTT, LocalTwoPassSplayTT, and MoveToRootTT) are available. We denote these implementations by `Greedy Splay C++`, `L2P Splay C++`, and `MTR C++`.

The author made several attempts at optimization, yielding multiple variants of each of the three algorithms. These variants can be chosen via compile-time flags; see the source code for more details. It turned out that the most impactful improvement was to replace calls to `is_separator_hint` and similar functions by a function that returns the *separator type* of a node, which can be "direct separator", "indirect separator", or "no separator". The `splay_step` procedure needs this information to choose between ZIG-ZIG and ZIG-ZAG steps, so retaining it once computed is advantageous.

We compare the STT-based implementation with two external dynamic forest libraries. First, the *dtree* C++ library[14] is a fully-fledged dynamic forest library, supporting vertex weights under a variety of different queries and updates. Due to its generic architecture, it is also possible to omit vertex values (like with our Rust library). It is further possible to choose between various implementations, including link-cut trees and top trees. We use dtree's link-cut trees without weights and call it `dtree`.

Second, Tarjan and Werneck implemented link-cut trees and several top tree variants [TW10]. All implementations include vertex or edge weights and it is not possible to remove them without significant edits to the source code. We use their link-cut tree implementations with vertex weights (`tarjan-werneck-v`) and edge weights (`tarjan-werneck-e`), ignoring the weights in both cases. Note that maintaining weights give these implementations an unfair disadvantage.

10.9. Experimental evaluation

We now describe our experiments and discuss their results. To reduce variance, every (sub)experiment was repeated ten times (for randomized experiments, the input is newly generated each time). All experiments can be reproduced by scripts in the source code; see the included `README.md` files for more details.

[14]Found at `https://www.davideisenstat.com/dtree/`

	Running time (µs/query)					
Algorithm	$n = 1000$	5000	10 000	50 000	100 000	500 000
`Link-cut`	0.45	0.50	0.55	0.69	0.74	1.36
`Greedy Splay`	0.40	0.47	0.53	0.66	0.71	1.24
`Stable Greedy Splay`	0.38	0.45	0.50	0.63	0.68	1.21
`2P Splay`	0.45	0.52	0.57	0.70	0.75	1.27
`Stable 2P Splay`	0.40	0.48	0.54	0.66	0.71	1.23
`L2P Splay`	0.38	0.48	0.54	0.67	0.72	1.24
`Stable L2P Splay`	0.36	0.47	0.52	0.65	0.70	1.23
`MTR`	0.34	0.42	0.47	0.57	0.61	1.08
`Stable MTR`	0.36	0.44	0.48	0.59	0.63	1.10
`1-cut`	0.15	0.49	0.86	3.38	–	–
`Petgraph`	7.35	–	–	–	–	–

Table 10.1.: Results for the uniformly random connectivity queries experiment.

Uniformly random connectivity queries. In our first experiment, weights are empty, so the updating logic from section 10.3 is not required and PATHWEIGHT(u, v) simply indicates whether u and v are connected or not. This allows us to directly compare the dynamic forest implementations without edge weight handling.

A list of queries is pre-generated, starting with an empty forest. For each query, we draw two vertices u, v uniformly at random; if u and v are not connected, we call LINK(u, v); otherwise, we either call PATHWEIGHT(u, v) or call CUT on some edge on the path between u and v, with probability $\frac{1}{2}$ each. This gives us roughly 40% LINK, 30% CUT, and 30% PATHWEIGHT queries. We then execute the list of queries once for each implementation.

We start with the Rust implementations. We always set $m = 10n$, and let n range from 1000 to 500 000.[15] See table 10.1 for the result. The `Petgraph` and `1-cut` implementations became very slow after a certain threshold and were therefore excluded for larger n. `Petgraph` in particular performed very badly (worse than the second-worst implementation by a factor of over 15 at $n = 1000$), so we exclude it from all further experiments.

Among our SPLAYTT variants, the stable ones are always slightly faster than the non-stable ones. The fastest overall is `Stable Greedy Splay`, which is somewhat surprising, since the discussion in section 9.4 indicates that `Stable L2P Splay` should be faster. The difference is small, however. Both algorithms outperform `Link-cut` by roughly 10% for large n.

The much simpler `MTR` and `Stable MTR` are faster than all Splay-based data structures, perhaps because of the uniformity of the input (as discussed in section 9.1). The simple linear-time 1-cut data structure is faster for smaller values of n, but is the worse by some margin already at $n = 10\,000$.

The same experiment was conducted for the various C++ implementations, with values $1000 \leq n \leq 1\,000\,000$, and again $m = 10n$. The results are shown in table 10.2. This time, `Greedy Splay C++` and `L2P Splay C++` are roughly on par. They outperform the well-optimized `dtree` library by roughly 20%. The Tarjan-Werneck implementations are

[15]The maximum value for n is chosen such that the overall experiment still takes a reasonable amount of time. This also applies to the other experiments.

	Running time (µs/query)					
Algorithm	$n = 5000$	10 000	50 000	100 000	500 000	1 000 000
`Greedy Splay C++`	0.26	0.22	0.31	0.34	0.57	0.85
`L2P Splay C++`	0.25	0.22	0.31	0.34	0.56	0.84
`MTR C++`	0.29	0.20	0.26	0.29	0.48	0.75
`dtree`	0.29	0.28	0.37	0.39	0.74	1.04
`tarjan-werneck-v`	0.49	0.53	0.65	–	–	–
`tarjan-werneck-e`	0.58	0.56	0.66	–	–	–

Table 10.2.: Results for the C++ connectivity queries experiment.

slower, although, as mentioned above, they are at an unfair disadvantage. Interestingly, even when implementing the same algorithm, the C++ implementations are much faster than the respective Rust implementations (by a factor of more than two).

The following advanced experiments only concern the Rust implementations, since the C++ implementations do not support edge weights and rooted forests. We omit the non-stable variants for the remaining experiments, since they so far performed very similarly to the stable variants (just slightly worse, usually).

Incremental MSF. Our second, more practical experiment consists of solving the incremental minimum spanning forest (MSF) problem. We are given the edges of a weighted graph one-by-one and have to maintain an MSF, i.e., a minimum spanning tree on every component. Edges are never removed.

A simple solution using dynamic forests works as follows. Whenever an edge $\{u, v\}$ with weight w arrives, if u and v are in different components, add the edge to the forest. Otherwise, find the heaviest edge on the path from u to v, and if its weight is larger than w, replace it with the new edge.

To find the actual heaviest edge instead of just its weight, we extend our edge weight monoid $(\mathbb{N}, \max)$ to also contain a heaviest edge. When two weights are added, the heavier of the two edges is used, with ties being broken arbitrarily. The result is still essentially a commutative monoid, hence our algorithms can be used without change.[16]

As a first experiment, we randomly generate inputs on $n \leq 500\,000$ vertices with $m = 8n$ edges (following Tarjan and Werneck [TW10]).

Second, we use the `ogbl-collab` data set[17] [HFZ+20] to generate input that might be closer to real-world applications. The data set consists of a set of authors and collaborations between authors, annotated with a year. We interpret this as a dynamically changing graph where the first collaboration creates an edge with weight 1, and each subsequent collaboration increases the weight of the edge. Inverting the edge weights yields a natural dynamic MSF problem, with the additional allowed operation of *decreasing* an edge weight, which can be easily implemented by first removing the edge (if it exists in the current

[16]The monoid is not technically commutative, since there might be ties between edge weights and these ties might be broken differently depending on the order of summands. However, we can treat it as a commutative monoid, since we do not care how ties are broken.

[17]Available under the ODC Attribution License at
`https://ogb.stanford.edu/docs/linkprop/#ogbl-collab`

MSF), and then adding it again with the new weight. The resulting input consists of 235 868 vertices and 1 179 052 queries.

We also compare the online algorithms with the Petgraph library's implementation of Kruskal's offline algorithm. Results are shown in table 10.3.

Kruskal's algorithm outperforms the online algorithms by a large factor (this is expected, since it is offline and less general). Otherwise, the results of this experiment are similar to the uniformly random query experiments; the only difference is that the advantage of `Stable Greedy Splay` against `Stable L2P Splay` has vanished.

Random queries with variable probability of PathWeight. Informal experiments lead the author to believe that `Link-Cut` performs better compared to our approaches when PathWeight queries are common (and thus the reverse bit is rarely changed). Hence, the first experiment was repeated with $n = 50\,000$, except that the probability p of generating a PathWeight query (instead of a Cut) is variable. Figure 10.3 shows that `link-cut` outperforms our SplayTT algorithms when p is close to 1, thus confirming the suspicion.

Degenerate queries. `MTR` and `Stable MTR` are faster than the other algorithms on uniform queries, despite having asymptotic worst-case performance of $\Theta(n)$ per operation. To experimentally confirm the worst-case behavior, we create a path G of $n \leq 10\,000$ nodes $v_1, v_2, \ldots, v_n$, and then call PathWeight(v_i, v_n) for all $i \in [n]$ in order. While the queries have strong locality, the two vertices v_i, v_n are very far from each other on average. All Splay-based approaches are able to exploit the locality and outperform the linear-time data structures (`MTR`, `Stable MTR`, and `1-cut`) by a factor of over 100 when $n = 10\,000$.

To check how "robust" our degenerate example is, we performed the following "noisy" experiment. Fixing $n = 5000$, for each $i \in [n]$, we call PathWeight(v_j, v_n), where $j = i + \lfloor x \rfloor$ and x is drawn from a normal distribution with mean 0 and standard deviation σ, for some values $\sigma \leq 300$. (See figure 10.4.) As expected, `1-cut` still performs very badly, since the added noise does not change the expected distance between v_i and v_n. `MTR` and `Stable MTR`, on the other hand, do adapt, though even with $\sigma = 300$ both are still slower than the Splay-based variants by at least 50%.

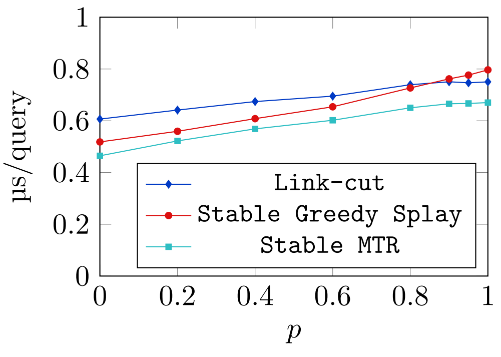


Figure 10.3.: Random queries with increasing probability p of PathWeight.

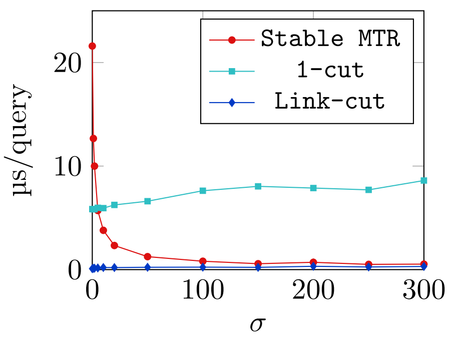


Figure 10.4.: Noisy degenerate input.

	Running time (µs/edge)					
Algorithm	$n = 5000$	10 000	50 000	100 000	500 000	`ogbl`
Kruskal (petgraph)	0.09	0.09	0.14	0.20	0.48	0.18
`Link-cut`	1.01	1.12	1.38	1.71	2.82	0.51
`Stable Greedy Splay`	0.90	1.00	1.24	1.51	2.48	0.38
`Stable 2P Splay`	0.95	1.05	1.29	1.55	2.55	0.41
`Stable L2P Splay`	0.89	0.99	1.23	1.50	2.48	0.39
`Stable MTR`	0.77	0.85	1.05	1.31	2.20	0.39
`1-cut`	0.40	0.60	1.46	1.96	–	0.17

Table 10.3.: Results for the uniformly random incremental MSF experiment.

Lowest common ancestors. In our final two experiments, we maintain a *rooted* forest on n vertices and execute $10n$ queries among $\textsc{Link}(u, v)$, $\textsc{Cut}(v)$, and $\textsc{LCA}(u, v)$. The query distribution is as follows. A random non-root node is $\textsc{Cut}$ with probability $\frac{1}{2} \cdot \frac{m}{n-1}$, where m is the current number of non-root nodes. Otherwise, a pair of nodes $\{u, v\}$ is generated uniformly at random, and $\textsc{Link}(u, v)$ or $\textsc{LCA}(u, v)$ is chosen depending on whether u and v are in the same tree (the $\textsc{Link}$ is performed at the root of the tree containing u). Overall, we have roughly 46% $\textsc{Link}$s, 38% $\textsc{Cut}$s, and 16% LCAs.

In the second experiment, we additionally allow $\textsc{Evert}(v)$, i.e., changing the root of a tree. Each $\textsc{Cut}$ is replaced with $\textsc{Evert}$ with probability $\frac{1}{2}$, resulting in roughly 30% $\textsc{Link}$s, 20% $\textsc{Cut}$s, 30% LCAs and 20% $\textsc{Evert}$s.

As expected, `Link-cut` outperforms our data structures considerably in the first experiment, where only our data structures have to maintain extra data (to represent rooted trees). When $\textsc{Evert}$ is allowed, the difference is much less pronounced, and `Stable Greedy Splay` is actually slightly faster for $n = 500\,000$. The `Simple` data structure performed much worse than all others and was excluded from experiments with large n. See tables 10.4 and 10.5 for more details.

Discussion. In our experiments, the $\textsc{Splay}$TT-based data structures outperform link-cut trees by 10-20% if the dynamic forest is unrooted. This holds for both our own Rust implementations, and when comparing our C++ implementation with third-party libraries. Link-cut trees in turn are faster by similar margins for rooted dynamic forests (in particular without the root-changing $\textsc{Evert}$ operation).

Among the tested $\textsc{Splay}$TT-based variants, `Greedy Splay` and `L2P Splay` performed best overall, which matches the theoretical analysis (see section 9.4). However, the even simpler `MTR` algorithm outperformed our more sophisticated algorithms by 10-15%, except for specifically constructed inputs. It would be interesting to investigate whether there exist practical applications where the adaptivity of Splay-based data structures makes up for their increased complexity.

Finally, we note the large discrepancy between the Rust and C++ implementation of the same data structure. There are multiple possible reasons. First, the Rust library consists entirely of *safe* Rust code.[18] This makes it impossible to work with raw node

[18]There is an unsafe optimization that can be enabled via a compiler flag, though it makes no noticeable difference in performance.

	Running time (µs/query)				
Algorithm	$n = 10\,000$	20 000	50 000	100 000	500 000
`Link-cut`	0.28	0.30	0.34	0.36	0.67
`Stable Greedy Splay`	0.37	0.40	0.43	0.46	0.78
`Stable 2P Splay`	0.40	0.44	0.47	0.49	0.91
`Stable L2P Splay`	0.37	0.41	0.44	0.47	0.90
`Stable MTR`	0.31	0.34	0.37	0.39	0.81
`Simple`	1.40	2.26	4.17	–	–

Table 10.4.: Results for the LCA experiment.

	Running time (µs/query)				
Algorithm	$n = 10\,000$	20 000	50 000	100 000	500 000
`Link-cut`	0.42	0.45	0.50	0.53	0.97
`Stable Greedy Splay`	0.45	0.48	0.52	0.55	0.93
`Stable 2P Splay`	0.49	0.53	0.57	0.60	1.11
`Stable L2P Splay`	0.45	0.49	0.53	0.57	1.09
`Stable MTR`	0.36	0.39	0.43	0.46	0.94
`Simple`	1.30	1.91	3.08	–	–

Table 10.5.: Results for the LCA experiment with EVERT.

pointers, as in the C++ implementation; the author opted to instead store node *indices* instead of pointers. To access a node, the index must be passed to a vector storing all nodes. This slightly increases the number of necessary memory accesses.

Second, the Rust library is much more powerful than the C++ implementation and makes heavy use of generics. There is no direct reason known to the author why this would hurt performance, but perhaps this inhibits compile-time optimization in some way. Finally, some parts of the library might simply be poorly written, since the author is not an experienced Rust programmer and the language is quite complex.

10.10. STGs for graphs of bounded tree-width

In this section, we discuss a data structure that maintains a k-cut search tree on a graph (with tree-width at most k) and can execute rotations in time $\mathcal{O}(2^k)$. At the end of the section, we name a possible application.

Theorem 10.11. *Let G be a connected graph. We can maintain a search tree on G under rotations, such that the running time a rotation between u and v is $\mathcal{O}(2^k)$, where $k = \max(|\partial(T_u)|, |\partial(T_v)|)$ and T is the search tree before the rotation.*

Proof sketch. Recall the naive data structure sketched at the beginning of section 10.1. Each node maintains v the `parent`(v) field as usual. Additionally, it maintains $\partial(T_v)$ as a list `bd`(v), a list `anc_adj`(v) of ancestors of v that are adjacent to v, and a list of its children `children`(v).

To rotate a node v with its parent p, we need to identify the children of v that switch parents to p. By definition, these are exactly the children c with $p \in \partial(T_c) = \texttt{bd}(c)$, so they can be identified by going through the list $\texttt{children}(v)$.

After executing a rotation, the fields $\texttt{anc\_adj}(\cdot)$ and $\texttt{bd}(\cdot)$ can only change for v and p. The former only changes if v and p are adjacent; then the edge $\{v, p\}$ is moved from $\texttt{anc\_adj}(v)$ to $\texttt{anc\_adj}(p)$. The latter field $\texttt{bd}(x)$ is always easily computed from $\texttt{anc\_adj}(x)$ and the fields $\texttt{bd}$ of the children of x.

Finding the children to switch parents and recomputing the boundaries both takes $\Theta(n)$ time in the worst case. We now show how to improve that to $\mathcal{O}(2^k)$.

The idea is to *group* the children by boundary. Let v be a node in a search tree T. For each child c of v, we have $v \in \partial(T_c)$ and $\partial(T_c) \subseteq \partial(T_v) \cup \{v\}$ (by observation 2.7). Hence, if $|\partial(T_v)| \leq k$, then there are at most 2^k possible boundaries for children of v. Instead of an explicit list of children, we store a list of *child groups*, each of which specifies the boundary and the parent of the contained nodes. It is not hard to adapt the rotation algorithm given above so that it uses child groups instead of children. We omit the details here. Theorem 10.11 follows. □

Remark. The field $\texttt{anc\_adj}$ is necessary in the non-tree case, since the underlying graph can *not* be computed just from boundary sizes (i.e., lemma 10.2 does not generalize to arbitrary graphs). Indeed, take any search tree T on G with some nodes u, v with $u \in \partial(T_v)$, but $\{u, v\} \notin E(G)$. Then adding the edge $\{u, v\}$ does not change the subtree boundaries, but obviously changes the graph.

Dynamic tree decompositions. Our main motivation for theorem 10.11 is an application to the *dynamic tree-width* problem. Recall that k-cut a search tree on a graph is essentially a *tree decomposition* of width k (see section 2.4). Thus, theorem 10.11 can be seen as maintaining a *dynamic tree decomposition* of G under rotations. If we could generalize, e.g., SplayTT to graphs of bounded tree-width, theorem 10.11 would perhaps enable us to extend our dynamic forest data structure to a data structure that maintains a tree decomposition of bounded width on a bounded-tree-width graph, under edge insertions and deletions.

This problem has been first considered by Bodlaender [Bod94], who showed how to maintain a tree decomposition of width *eleven* on graphs of tree-width *two* with $\mathcal{O}(\log n)$ time per operation. Recently, Korhonen, Majewski, Nadara, Pilipczuk, and Sokołowski [KMN+23] gave a data structure that maintains a $(6t + 5)$-width tree decomposition on graphs with tree-width t, with time $\mathcal{O}_k(2^{\sqrt{\log n} \log \log n})$ per operation.[19]

Modifying tree decompositions directly is rather complicated. Using the data structure of theorem 10.11, this problem is abstracted away to some extent – as long as we keep the search tree k-cut, we can freely perform rotations, and need not worry about rebuilding bags correctly, for example. Hence, approaches other than SplayTT may also benefit from this idea. Still, it seems that the most promising avenue is trying to adapt SplayTT.

Open question 10.1. Is there a generalization of SplayTT that maintains $\mathcal{O}(k)$-cut search trees on (static) graphs of tree-width k?

[19] $\mathcal{O}_k(\cdot)$ hides factors depending only on k.

Part III.

The diameter of graph associahedra

11. Rotation distance and static search trees

Let G be a connected graph. The *search tree rotation graph* (or simply *rotation graph*) $\mathcal{R}(G)$ on G is the graph where

rotation graph

- the set of vertices is the set of search trees on G; and
- two vertices T, T' are adjacent if T can be obtained from T' by a single rotation.

Observe that the shortest path between two search trees T and T' in $\mathcal{R}(G)$ is precisely the minimum number of rotations required to transform T into T'. We call this the *rotation distance* between T and T' and denote it by $d_G(T, T')$ (or $d(T, T')$, if G is clear from context). The diameter $\operatorname{diam}(\mathcal{R}(G))$ is thus the maximum rotation distance between two search trees on G. In this final part of the thesis, we study the quantity $\operatorname{diam}(\mathcal{R}(G))$, mainly in the case where G is a tree. It turns out that there is a connection to polyhedral combinatorics, which we discuss now.

rotation dist. $d(T, T')$

Graph associahedra. An *associahedron* is a convex polytope whose graph[1] is the rotation graph of binary search trees of a given size, as well as the *flip graph* of various other combinatorial structures [Tam54, Pou14b]. A generalization of associahedra called *graph associahedra* was introduced by Carr and Devadoss [CD04]. It turns out that the graphs of these polytopes are the rotation graphs of STGs, though Carr and Devadoss defined them with different terminology. We briefly define graph associahedra and discuss their connection to STGs.

associahedron

graph associahedron

Let G be a connected graph. A *tube* of G is a connected induced subgraph of G. A *tubing* is a set of tubes such that every pair of tubes is either nonadjacent or nested. Now consider the inverse of the partial order induced by set inclusion on all tubings of G. This is the *face lattice* of the G-associahedron, denoted by $\mathcal{A}(G)$. In other words, each k-dimensional face of $\mathcal{A}(G)$ corresponds to a tubing of size $n + 1 - k$, and a face F is contained in a face F' if for the corresponding tubings U, U' we have $U \supseteq U'$. Carr and Devadoss [CD04, Theorem 2.6] showed that $\mathcal{A}(G)$ is realizable, i.e., there is a geometric polytope with this face lattice, for each graph G. Devadoss [Dev09] gave a simple realization with *integer* coordinates.

tubing

It is not hard to see that maximal tubings are in bijection with STGs, and that edges of $\mathcal{A}(G)$ correspond to rotations in STGs [MP15, CLPL18]. Hence, the graph of $\mathcal{A}(G)$ is exactly the rotation graph $\mathcal{R}(G)$, and the diameter of $\mathcal{A}(G)$ is exactly the diameter of $\mathcal{R}(G)$.

If G is a path, then $\mathcal{A}(G)$ is the classical *associahedron*, mentioned above. The study of BST rotation distances, and thus the diameter of the associahedron goes back to the 1980's [CW82, Pal87, STT88].

[1]The *graph* or *1-skeleton* of a polytope is the graph formed by the vertices and edges of the polytope.

permutahedron

stellohedron

cyclohedron

If G is a clique, then $\mathcal{A}(G)$ is called the *permutahedron.* Following our observations in section 2.2.5, the rotation graph $\mathcal{R}(G)$ is the *flip graph* of permutations, where a flip is an adjacent swap. Similarly, a star-associahedron is called *stellohedron* [PRW08] and its graph is the flip graph of *partial permutations* (see section 2.2.5). Another important special case is the cycle-associahedron, which is also called *cyclohedron* or *Bott-Taubes polytope* [PRW08].

Graph associahedra are easily generalized to *disconnected* graphs, as the definition of tubings and the face lattice do not require the graph to be connected. Tubings on disconnected graphs correspond to *sets* of STGs, one for each connected component. A rotation in this setting is a rotation in one of the STGs. Clearly, if G is the disjoint union of two graphs H_1, H_2, then $\mathcal{R}(G)$ is the Cartesian product of $\mathcal{R}(H_1)$ and $\mathcal{R}(H_2)$. Hence, we have $\operatorname{diam}(\mathcal{R}(G)) = \operatorname{diam}(\mathcal{R}(H_1)) + \operatorname{diam}(\mathcal{R}(H_2))$, and we can focus on the case of connected graphs.

We now discuss old and new results on the diameter of graph associahedra. The following basic properties will be useful for the discussion.

Lemma 11.1 (Manneville and Pilaud [MP15, Theorem 1])**.** $\operatorname{diam}(\mathcal{R}(G))$ *is non-decreasing. That is, if* G *is a subgraph of* G'*, then* $\operatorname{diam}(\mathcal{R}(G')) \leq \operatorname{diam}(\mathcal{R}(G))$.

Lemma 11.2 (Manneville and Pilaud [MP15, Theorem 3])**.** *Let G be a connected graph on n vertices and m edges. Then*

$$\max(m, 2n-20) \leq \operatorname{diam}(\mathcal{R}(G)) \leq \tbinom{n}{2}.$$

Diameter of individual graph associahedra. We start with the special case of the classical associahedron, i.e., the rotation graph of *BSTs.* Let P_n denote the path on n vertices. Sleator, Tarjan, and Thurston [STT88] proved that the diameter of $\mathcal{R}(P_n)$, is at most $2n-6$ for $n \geq 11$, and that this bound is tight for large enough n. They used a characterization of $\mathcal{R}(P_n)$ as the flip graph of *triangulations.* Pournin [Pou14b] showed that the bound is tight for all $n \geq 11$.

Theorem 11.3 ([STT88, Pou14b])**.** *Let P_n be the path with n vertices. We have* $\operatorname{diam}(\mathcal{R}(G)) = 2n-6$ *if* $n \geq 11$*, and* $2n-6 \leq \operatorname{diam}(\mathcal{R}(G)) \leq 2n-2$ *if* $1 \leq n \leq 10$.

Exact bounds are also known for other simple graphs. In the following, let n denote the number of vertices in the underlying graph. The diameter of the permutahedron is easily seen to be $\binom{n}{2}$ (thus, the upper bound in lemma 11.2 follows from lemma 11.1). The stellohedron can be shown to have diameter precisely $2(n-1)$ for n large enough [MP15]. Cardinal, Pournin, and Valencia-Pabon [CPVP22] studied the diameter of graph associahedra of *complete split graphs* (which interpolate between Stars and Cliques), *unbalanced bicliques* and *complete multipartite graphs.* See the paper for definitions and precise results. Gargantini, Pastine, and Torres [GPT24] provided additional exact results for small balanced bicliques.

Pournin [Pou14a] showed that the diameter of the cyclohedron is $\lceil \frac{5}{2}n \rceil - \mathcal{O}(\sqrt{n})$, settling that question up to lower-order terms. The exact diameters of cyclohedra are currently not known except for small values.

Another approximate result by Cardinal et al. [CPVP22] is for associahedra of *trivially perfect graphs* (defined in section 2.3).

Theorem 11.4 ([CPVP22])**.** *Let G be a connected trivially perfect graph with m edges. Then $m \leq \operatorname{diam}(\mathcal{R}(G)) \leq 2m$.*

Observe that the lower bound follows directly from lemma 11.2.

We now turn to our new results. First, we present an upper bound in terms of the *unweighted static optimum* (see chapter 3). Recall that $\mathbb{1}$ is the *unit weight function*, which assigns weight one to each vertex.

Theorem 11.5. *Let G be a connected graph on n vertices. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \leq 2(\operatorname{StOPT}(G, \mathbb{1}) - n).$$

We prove theorem 11.5 in section 11.1. The proof is based on a very simple algorithm that transforms a search tree T into another search tree T': If r is the root of T', bring r to the root in T. Then, recurse on each subtree. The number of rotations can be shown to be $\operatorname{cost}(G, \mathbb{1}) - n$. Hence, the distance to the optimal static search tree is $\operatorname{StOPT}(G, \mathbb{1}) - n$, which yields theorem 11.5.

Theorem 11.5 is typically not tight, but it allows us to re-derive several known results (and, in some instances, slightly improve them). First, from the existence of centroid trees, we know that $\operatorname{StOPT}(G, \mathbb{1}) \in \mathcal{O}(n \log n)$ for each tree G on n vertices, and hence $\operatorname{diam}(\mathcal{R}(G)) \in \mathcal{O}(n \log n)$. This has been known before via a special case of the same algorithm described above [CLPL18].

The explicit connection to unweighted optimal search trees is new. Another immediate consequence is the upper bound in theorem 11.4, which follows by combining theorem 11.5 with lemma 7.6. The original proof by Cardinal et al. [CPVP22] is more involved and uses the concept of *universal clique decompositions.*

The algorithm can be modified to only move a subset of vertices; this gives us the following result (proof in section 11.2). See section 2.2.6 for the definition of $\operatorname{torso}_G(\,\cdot\,)$.

Theorem 11.6. *Let G be a graph, let $A \subseteq V(G)$, and let $H = \operatorname{torso}_G(A)$. For each $v \in A$, let $w(v)$ be the number of vertices connected to v in $G - (A \setminus \{v\})$, including v itself. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \leq 2 \cdot (\operatorname{StOPT}(H, w) - |A|) + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C)).$$

An interesting immediate consequence of theorem 11.6 is an upper bound for associahedra of *graph subdivisions.*

Theorem 11.7. *Let H be a connected graph, and let G be a subdivision of H. Let $s_e \in \mathbb{N}_0$ be the number of times the edge $e \in E(H)$ was subdivided to obtain G, and let $w(v) = \sum_{e \in E_v} s_e$ for each $v \in V(H)$, where E_v is the set of edges incident to v. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \leq 2 \cdot \operatorname{StOPT}(H, w + \mathbb{1}) + 2|V(G)| - 4|V(H)|.$$

All these new upper bounds are far from tight for certain graphs. In chapter 12, we provide tight bounds (up to a constant factor) for many trees. The main technique is a reduction in the style of theorem 11.6, for both upper and lower bounds.

The proofs are quite involved and highlight a strong connection between rotation distance and the *dynamic* search tree model. We refer to chapter 12 for more details. In the following, we discuss some core results.

First, we show that theorem 11.5 is tight not just on trivially perfect graphs, but also on certain trees.

Theorem 11.8. *Let G be a tree with no vertices of degree two. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \in \Theta(\mathrm{StOPT}(G, \mathbb{1})).$$

Our main result is an algorithmic characterization of the diameters of all trees of bounded *path-width* (see section 2.4).

Theorem 11.9. *Given a tree G on n vertices with path-width k, we can approximate* $\operatorname{diam}(\mathcal{R}(G))$ *up to a factor of four and an additive term of $\mathcal{O}(kn)$, in time $\mathcal{O}(kn^2)$.*

The algorithm is very simple and essentially consists of recursive application of a structural result. The path-width appears due to a connection with the so-called *Strahler number* (defined in section 12.5.2). As an example, applying a single step of the algorithm yields the following result on *caterpillars*, i.e., graphs with path-width one.

Theorem 11.10. *Let G be a caterpillar graph consisting of a path P of n vertices and a set of m leaves attached directly to P. Let $w(v)$ denote the number of neighbor leaves of $v \in V(P)$. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \in \Theta(\mathrm{StOPT}(P, w) + n).$$

Recall that the cost of the optimal static search tree on a path is asymptotically equal to the entropy of the weight distribution, times the total weight [Meh75]. Hence, we characterize the diameter of caterpillar associahedra in terms of the entropy of the *leaf distribution.* We can obtain similar tight bounds for other graph classes like spiders, and lobsters (see chapter 2 for definitions).

Graph classes. Another, less specific problem is to determine the *maximum* diameter of graph associahedra within a graph class. Formally, let $\mathcal{G}$ be a set of graphs containing at least one graph for each number of vertices, and let

$$\operatorname{Diam}(\mathcal{G}, n) = \max_{\substack{G \in \mathcal{G} \\ |V(G)| = n}} \operatorname{diam}(\mathcal{R}(G)).$$

We observed above that $\operatorname{Diam}(\mathcal{G}, n) \in \mathcal{O}(n \log n)$ if $\mathcal{G}$ is the set of all trees. This bound is known to be asymptotically tight.

Theorem 11.11 (Cardinal, Langerman and Pérez-Lantero [CLPL18])**.** *If $\mathcal{G}$ is the set of all trees, then* $\operatorname{Diam}(\mathcal{G}, n) \in \Theta(n \log n)$.

Observe that this also follows from theorem 11.10, by taking the caterpillar with a path of $\frac{n}{2}$ vertices, each of which has a leaf attached to it.

Cardinal, Pournin, and Valencia-Pabon [CPVP22] further provided the following results concerning the graph parameters tree-depth, path-width, and tree-width. We start with tree-depth.

Theorem 11.12 ([CPVP22, Theorems 12 and 13]). *Fix $t \in \mathbb{N}$, and let $\mathcal{G}_t^{\mathrm{td}}$ be the class of graphs with tree-depth at most t. Then, for each n,*

$$\mathrm{Diam}(\mathcal{G}_t^{\mathrm{td}}, n) \leq 2tn,$$

and, for each n such that $t-1$ divides $n-1$,

$$\mathrm{Diam}(\mathcal{G}_t^{\mathrm{td}}, n) \geq \tfrac{t}{2} \cdot (n-1).$$

We can easily derive the upper bound from theorem 11.5, even with a small additive improvement. We also give a better lower bound that is implicit in the same paper [CPVP22].

Theorem 11.13. *Let $t \in \mathbb{N}$, and let $\mathcal{G}_t^{\mathrm{td}}$ be the class of graphs with tree-depth at most t. Then, for each $n \geq 5t-4$, we have*

$$2a_{t,n} - \tbinom{t-1}{2} \leq \mathrm{Diam}(\mathcal{G}_t^{\mathrm{td}}, n) \leq 2a_{t,n},$$

where $a_{t,n} = (t-1)n - \binom{t}{2}$.

Cardinal et al. [CPVP22] further give a lower bound for the class of graphs with fixed path-width. Using the well-known inequalities $\mathrm{tw}(G) \leq \mathrm{pw}(G) \leq \mathrm{td}(G) \cdot \log n$ [BGHK95], they obtain:

Theorem 11.14 ([CPVP22]). *Fix $t \in \mathbb{N}$, and let $\mathcal{G}_t^{\mathrm{tw}}$, $\mathcal{G}_t^{\mathrm{pw}}$, $\mathcal{G}_t^{\mathrm{td}}$ be the classes of graphs with tree-width, resp. path-width, resp. tree-depth at most t. Then, for each n,*

$$\Omega(t \cdot n) \ni \mathrm{Diam}(\mathcal{G}_t^{\mathrm{tw}}, n) \leq \mathrm{Diam}(\mathcal{G}_t^{\mathrm{pw}}, n) \in \mathcal{O}(t \cdot n \log n).$$

They show that the lower bound is not tight for small t; in particular, associahedra of graphs with path-width two can have diameter $\Omega(n \log n)$, hence $\mathrm{Diam}(\mathcal{G}_2^{\mathrm{pw}}, n) \in \Theta(n \log n)$. Our theorem 11.10 implies that this is true even for path-width *one*, since caterpillars are the graphs with path-width one. Still, the question stays open for non-tree graphs of unbounded tree- or path-width.

Open question 11.1. What are asymptotically tight bounds for $\mathrm{Diam}(\mathcal{G}_t^{\mathrm{tw}}, n)$ and $\mathrm{Diam}(\mathcal{G}_t^{\mathrm{pw}}, n)$?

k-cut rotation graphs. Several of the algorithms presented in parts I and II exclusively use *k-cut search trees*, due to them being easier to work with in several ways. An obvious question is to determine the maximum rotation distance between k-cut search trees. In chapter 13, we prove:

Theorem 11.15. *Let T, T' be two k-cut search trees on a tree G. Then there is a sequence of at most $(2k-1)n$ rotations that transforms T into T', such that each intermediate search tree is k-cut.*

Recall that the diameter of the rotation graph is important for the precise definition of the dynamic search tree model (see chapter 8).

Related work. Another related problem is *computing* the rotation distance between two given search trees. In general, this is known to be NP-hard [IKK$^+$23], though in the special case of *complete split graphs*, a polynomial-time algorithm is known [CPVP24]. It is unknown if the rotation distance of binary search trees can be computed in polynomial time, but a linear-time 2-approximation algorithm exists [CS10].

Manneville and Pilaud [MP15] also proved that $\mathcal{R}(G)$ is Hamiltonian (if $|V(G)| \geq 3$). Cardinal, Merino and Mütze [CMM21] showed that a Hamiltonian path can be generated efficiently if G is chordal.

Finally, the *chromatic number* of rotation graphs has so far only been considered for the classical (path-)associahedron [FMFPH$^+$09, BRSW18]. However, a related problem of coloring *facets* has been studied for certain graph associahedra [BIP23].

Organization of part III. In the remainder of this chapter, we prove some simpler results related to optimal static search trees (including theorems 11.6 and 11.13). Along the way, we will obtain some tools that are useful later. In chapter 12, we connect the dynamic search tree model to graph associahedra, and prove theorem 11.9 and related results. Finally, in chapter 13, we prove and discuss theorem 11.15.

11.1. Unweighted static search tree bound

In this section, we prove

Theorem 11.5. *Let G be a connected graph on n vertices. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \leq 2(\operatorname{StOPT}(G, \mathbb{1}) - n).$$

We start with bounding the distance between two search trees based on their cost.

Lemma 11.16. *Let G be a connected graph on n vertices and let T, T' be search trees on G. Then $d(T, T') \leq \operatorname{cost}(T', \mathbb{1}) - n$.*

Proof. We proceed by induction on n. If $n = 1$, then $T = T'$ and we are done. Suppose now that $n \geq 2$. We describe a sequence of rotations transforming T into T'.

Let r be the root of T'. We start by rotating r to top of T. This requires at most $n - 1 = |V(T'_r)| - 1$ rotations. Let T^1 be the resulting search tree.

Now consider a child subtree S of r in T^1. Since, by definition, $G[V(S)]$ is a component of $G - r$, and r is also the root of T', there is a child subtree S' of r in T' such that $V(S) = V(S')$. We recursively transform S into S', and repeat this for every other child subtree of r in T^1, after which we obtain T'. If K is the set of children of r in T', then, by induction, the total number of rotations is

$$|V(T')| - 1 + \sum_{c \in K} \operatorname{cost}(T'_c, \mathbb{1}) - |V(T'_c)| = \operatorname{cost}(T', \mathbb{1}) - n. \qquad \square$$

Clearly, by symmetry, we also have $d(T, T') \leq \operatorname{cost}(T, \mathbb{1}) - n$. This can be proved directly by observing that each node v is rotated upwards at most $\operatorname{depth}_T(v)$ times; so our two characterizations of $\operatorname{cost}(T, w)$ (observation 3.11) each yield a proof of lemma 11.16. Proving theorem 11.5 is now easy.

Proof of theorem 11.5. Let us write $D = \mathrm{StOPT}(G, \mathbb{1})$. Let T^* be an optimal static search tree on G, i.e., $\mathrm{cost}(T^*) = D$.

Let T, T' be arbitrary search trees on G. We claim that $d(T, T') \leq 2(D - n)$. By lemma 11.16, we have both $d(T, T^*) \leq D - n$ and $d(T', T^*) \leq D - n$. Our claim follows by the triangle inequality. □

11.1.1. Tree-depth

For the class of fixed-tree-depth graphs, theorem 11.5 is asymptotically tight, as shown by Cardinal, Pournin, and Valencia-Pabon [CPVP22]. We re-prove their result with better constant terms.

Theorem 11.13. *Let $t \in \mathbb{N}$, and let $\mathcal{G}_t^{\mathrm{td}}$ be the class of graphs with tree-depth at most t. Then, for each $n \geq 5t - 4$, we have*

$$2a_{t,n} - \binom{t-1}{2} \leq \mathrm{Diam}(\mathcal{G}_t^{\mathrm{td}}, n) \leq 2a_{t,n},$$

where $a_{t,n} = (t-1)n - \binom{t}{2}$.

Proof. We start with the upper bound. Let $G \in \mathcal{G}_t^{\mathrm{td}}$. By assumption, G has a search tree T^* of height t. For $i \in [t]$, let m_i be the number of vertices of depth *at least* i. Since at least $i - 1$ vertices have depth less than i, we have $m_i \leq n - (i - 1)$. Furthermore, observe that $\mathrm{cost}(T^*) = \sum_{i=1}^{t} m_i$. By theorem 11.5, we have

$$\tfrac{1}{2}\,\mathrm{diam}(\mathcal{R}(G)) \leq \mathrm{cost}(T^*) - n \leq \sum_{i=1}^{t}(n - (i-1)) - n = (t-1)n - \binom{t}{2} = a_{t,n}.$$

For the lower bound, consider the rooted tree T on n nodes such that there are $n-(t-1)$ nodes at depth t and one node each at depths 1 to $t-1$. Let $G = \mathrm{cl}(T)$.[2] Recall that T is a search tree on G (lemma 2.24), which implies that the tree-depth of G is at most t.

It is easy to see that $\mathrm{cost}(T, \mathbb{1}) = tn - \binom{t}{2}$, and thus G has $(t-1)n - \binom{t}{2} = a_{t,n}$ edges by lemma 7.4. This already gives a lower bound for $\mathrm{diam}(\mathcal{R}(G))$ with lemma 11.2.

However, we can obtain a better lower bound by observing that G is the *complete split graph* $\mathrm{SPK}_{t-1,n-t+1}$, i.e., it consists of a clique of size $t-1$ (the vertices of depth $< t$) and an independent set of size $n - t + 1$ (the vertices of depth t), such that each vertex of the independent set is adjacent to all clique vertices. Cardinal et al. [CPVP22] showed that $\mathrm{diam}(\mathcal{R}(\mathrm{SPK}_{p,q})) = 2pq + \binom{p}{2}$ if $q \geq 4p + 1$. Our claim follows by a simple calculation. □

We do not expect our upper bound to be tight up to constant additive terms; it seems more likely that the lower bound is correct.

Conjecture 11.2. For all t and large enough n, we have

$$\mathrm{Diam}(\mathcal{G}_t^{\mathrm{td}}, n) = \mathrm{diam}(\mathcal{R}(\mathrm{SPK}_{t-1,n-t+1})).$$

[2]See section 2.3 for a definition of $\mathrm{cl}(T)$.

Sparse graphs. The lower bound in the proof of theorem 11.13 uses very dense graphs with $\operatorname{diam}(\mathcal{R}(G)) \in \Theta(|E(G)|)$. It is natural to ask whether the upper bound of theorem 11.13 is still tight for *sparse* graphs. It is certainly not tight for paths: for the n-vertex path P_n, we have $\operatorname{td}(P_n) = \lceil \log(n+1) \rceil$, but $\operatorname{diam}(\mathcal{R}(P_n)) \leq 2n$. However, it *is* tight already for the very restricted class of *caterpillars*. This can be shown using theorem 11.10, proved in chapter 12.

Proposition 11.17. *For each n, t with $n \geq 2^t$, there exists a caterpillar G on n vertices with tree-depth t such that* $\operatorname{diam}(\mathcal{R}(G)) \in \Omega(tn)$.

Proof. W.l.o.g., let n be a multiple of $m = 2^{t-1} - 1$. Take a path P_m with m vertices and attach $\frac{n}{m} - 1 \geq 1$ leaves to each vertex on the path. By theorem 11.10, we have

$$\operatorname{diam}(\mathcal{R}(G)) \in \Omega((n-m)\log m + m) \supseteq \Omega(n \cdot t).$$

On the other hand, we have $\operatorname{td}(G) = \operatorname{td}(P_m) + 1 = t$. □

11.2. Torso upper bound

In this section, we prove:

Theorem 11.6. *Let G be a graph, let $A \subseteq V(G)$, and let $H = \operatorname{torso}_G(A)$. For each $v \in A$, let $w(v)$ be the number of vertices connected to v in $G - (A \setminus \{v\})$, including v itself. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \leq 2 \cdot (\operatorname{StOPT}(H, w) - |A|) + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C)).$$

See section 2.2.6 for the definition of $\operatorname{torso}_G(\,\cdot\,)$. Observe that with $A = V(G)$, we obtain theorem 11.5. The proof idea is similar to that of theorem 11.5, and relies on the following lemma. For convenience, we denote the weight function from theorem 11.6 by $w_{G,A}$ in the following.

Lemma 11.18. *Let G be a graph, let $A \subseteq V(G)$, and let $H = \operatorname{torso}_G(A)$. Let T be a search tree on G and let T^H be a search tree on H. Then we can transform T into a search tree T' with prefix T^H, using at most* $\operatorname{cost}(T^H, w_{G,A}) - |A|$ *rotations.*

Central to the proof of lemma 11.18 is the procedure Transform (algorithm 11.17), which we discuss first.

Lemma 11.19. *Let T be a search tree on a graph G, and let π be a permutation of a vertex set $A \subseteq V(G)$. Let T' be the search tree produced by* Transform(T, π). *Then T' has a topological ordering π' with prefix π, and* Transform(T, π) *uses at most $\sum_{v \in A} |V(T'_v)| - 1$ rotations.*

Proof. The bound on the number of rotations can be derived similarly as in the proof of theorem 11.5: The rotations that move $v \in A$ upwards entirely take place within $V(T'_v)$, since v is not touched again afterwards. This means there are no more than $|V(T'_v)|$ such rotations.

To show that π is a prefix of some topological ordering of T, take two nodes $u, v \in A$. Clearly, if u precedes v in π, then $v \not\prec_{T'} u$. Moreover, for each $u \in A$ and $v \in V(G) \setminus A$, we have $v \not\prec_{T'} u$. Thus, there is a topological ordering of T' that starts with π. □

Algorithm 11.17 Transform a search tree T according to a permutation π.

```
Input: Search tree $T$, permutation $\pi$ on a set $A \subseteq V(T)$.
procedure Transform($T, \pi$)
    if $|V(T) \cap A| > 1$ then
        $r \leftarrow$ first element of $\pi$ that is contained in $T$.
        Rotate $r$ to the root of $T$
        for each child $c$ of $r$ do
            Transform($T_c, \pi$)
```

We need to show one more technical statement to prove lemma 11.18.

Lemma 11.20. *Let T be a search tree on a graph G, let P be a prefix of T, and let $A = V(P)$. Then $|V(T_a)| \leq w_{G,A}(P_a)$ for each $a \in A$.*

Proof. For each $a \in A$, let $\mathcal{C}_a$ be the set of components of $G - A$ that are adjacent to a, and let $B_a = \{a\} \cup \bigcup_{C \in \mathcal{C}_a} V(C)$. Observe that $w_{G,A}(a) = |B_a|$ by definition.

We claim that $V(T_a) \subseteq \bigcup_{a' \in V(P_a)} B_{a'}$ for each $a \in A$. This implies that

$$|V(T_a)| \leq \sum_{a' \in V(P_a)} |B_{a'}| = \sum_{a' \in V(P_a)} w_{G,A}(a') = w_{G,A}(P_a),$$

as desired.

First, observe that trivially $a' \in B_{a'}$ for each $a' \in A$. Second, consider some child $c \notin A$ of a node $a' \in A$. Since $A = V(P)$ induces a prefix of T, we know that $V(T_c)$ is disjoint from A. Since further $V(T_c)$ is connected and adjacent to a' (lemma 2.3 and observation 2.7), we have $V(T_c) \subseteq B_{a'}$.

The two observations $a' \in B_{a'}$ and $V(T_c) \subseteq B_{a'}$ for each child $c \notin A$ of a' imply the claim and thus the lemma. □

Proof of lemma 11.18. Let π be an arbitrary topological ordering of T^H. Applying lemma 11.19 yields a search tree T' such that π is a prefix of a topological ordering of T'. By lemma 2.15, there is a prefix P of T with $V(P) = V(\pi) = V(T^H)$, and π is a topological ordering of P.

Theorem 2.21 implies that P is a search tree on $\operatorname{torso}_G(V(P)) = H$. Since π is a topological ordering of both T^H and P, the two search trees are identical.

It remains to bound the number t of performed rotations. By lemma 11.19, we have

$$t \leq \sum_{v \in A} |V(T'_v)| - 1.$$

From lemma 11.20, we get $|V(T'_v)| \leq w_{G,A}(T^H_v)$. This implies that $\sum_{v \in A} |V(T'_v)| \leq \operatorname{cost}(T^H, w_{G,A})$ by observation 3.11. The required bound thus follows. □

To finish the proof of theorem 11.6, we need the following basic observations, which will continue to be useful in chapter 12.

Lemma 11.21. *Let T be a search tree on a graph G, and let $A \subseteq V(G)$ induce a prefix of T. Then, each component of $G - A$ induces a rooted subtree of T.*

Proof. Let C be a component of $G - A$ and let $v = \mathrm{LCA}_T(V(C))$. By definition, we have $V(C) \subseteq V(T_v)$. Since A induces a prefix of T and $v \in V(C)$ by lemma 2.4, we have $A \cap V(T_v) = \emptyset$. Since A separates C from the rest of G and $G[T_v]$ is connected, we have $V(T_v) \subseteq V(C)$. Overall, we have $V(T_v) = V(C)$, so indeed C induces a rooted subtree of T. □

Lemma 11.22. *Let T be a search tree on a graph G. Let S be a rooted subtree of T, let S' be a search tree on $G[S]$, and let T' be the search tree obtained by replacing S with S' in T. Then $d(T, T') = d(S, S')$.*

Proof. Let X be a minimum-length sequence of rotations transforming S into S'. Clearly we can apply X to T (by lemma 2.10, no nodes outside $V(C)$ are touched), which produces T'. □

Lemma 11.23. *Let T, T' be search trees on a graph G with a common prefix P. Then*

$$d(T, T') \leq \sum_{C \in \mathbb{C}(G - V(P))} d(T|_C, T'|_C).$$

Proof. Consider a component C of $G - V(P)$. By lemma 11.21, both $S = T|_C$ and $S' = T'|_C$ are rooted subtrees of T, resp. T'. Using lemma 11.22, we can replace S by S' in T with $d(S, S')$ rotations, yielding a search tree T'' with prefix P and where $T''|_C = T'|_C$.

Repeating this for each component of $G - V(P)$ produces T', using the desired number of rotations. □

Proof of theorem 11.6. Let S and T be arbitrary search trees on G, and let T^H be an optimal search tree on the weighted graph $(H, w_{G,A})$. We first apply lemma 11.18 to S and T to obtain search trees S' and T', both of which have T^H as a prefix. The rotation distance between S' and T' can then be bounded using lemma 11.23. The overall number of rotations is at most

$$\begin{aligned} &2 \cdot (\mathrm{cost}(T^H, w_{G,A}) - |A|) + \sum_{C \in \mathbb{C}(G-A)} d(S'|_C, T'|_C) \\ &\leq 2 \cdot (\mathrm{StOPT}(H, w_{G,A}) - |A|) + \sum_{C \in \mathbb{C}(G-A)} \mathrm{diam}(\mathcal{R}(C)), \end{aligned}$$

as desired. □

Observe that if G is a subdivision of H, then $\mathrm{torso}_G(V(H)) = H$, and the components of $G - H$ are all paths. Hence, we have $\mathrm{diam}(\mathcal{R}(C)) \leq 2 \cdot |V(C)|$ for each $C \in \mathbb{C}(G - H$ (theorem 11.3), and we obtain theorem 11.7 as a direct consequence of theorem 11.6.

Theorem 11.7. *Let H be a connected graph, and let G be a subdivision of H. Let $s_e \in \mathbb{N}_0$ be the number of times the edge $e \in E(H)$ was subdivided to obtain G, and let $w(v) = \sum_{e \in E_v} s_e$ for each $v \in V(H)$, where E_v is the set of edges incident to v. Then*

$$\mathrm{diam}(\mathcal{R}(G)) \leq 2 \cdot \mathrm{StOPT}(H, w + \mathbb{1}) + 2|V(G)| - 4|V(H)|.$$

12. Rotation distance and dynamic search trees

In this chapter, we connect the dynamic search tree model with the diameter of rotation graphs to prove strong upper and lower bounds on the diameter of tree associahedra. Our main result is the following.

Theorem 11.9. *Given a tree G on n vertices with path-width k, we can approximate* $\operatorname{diam}(\mathcal{R}(G))$ *up to a factor of four and an additive term of $\mathcal{O}(kn)$, in time $\mathcal{O}(kn^2)$.*

Since we always have $\operatorname{diam}(\mathcal{R}(G)) \in \Omega(n)$ (lemma 11.2), this yields a constant-factor approximation for trees of bounded path-width.

The algorithm of theorem 11.9 is exceedingly simple. The interesting part are the structural observations proving its quality of approximation. We are mainly interested in finding a "formula" for the diameter of tree associahedra – but since the structure of trees (even with bounded path-width) is likely too complicated for any sort of closed-form expression, a simple algorithm is the next best thing.

Note that trees with bounded path-width k are (approximately) characterized by not containing subdivisions of (unrootings of) binary trees of height $\approx k$ (see section 12.5.2). These graphs seem to be the main obstacle towards a full approximate characterization of the diameter of tree associahedra. The author conjectures that theorem 11.7 is tight for these graphs, but has been unable to prove any non-trivial lower bound.

Conjecture 12.1. Let H be a binary tree, and let G be a subdivision of H. Let $s_e \in \mathbb{N}_0$ be the number of times the edge $e \in E(H)$ was subdivided to obtain G, and let $w(v) = \sum_{e \in E_v} s_e$ for each $v \in V(H)$, where E_v is the set of edges incident to v. Then

$$\operatorname{diam}(\mathcal{R}(G)) \in \Theta(\operatorname{StOPT}(H, w + \mathbb{1}) + |V(G)|).$$

Caterpillar associahedra and dynamic search trees. We illustrate the connection between rotation distance and the dynamic search tree model with the following example. Consider a *caterpillar* G, i.e., a tree consisting of a path P (the *spine*) and a collection of leaves attached directly to the spine (the *legs*). See figure 12.1 for an example.

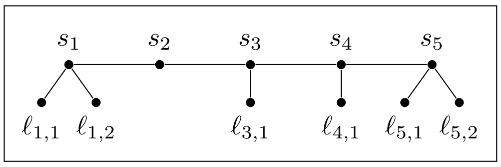


Figure 12.1.: A caterpillar with five spine vertices and seven leg vertices.

We start by making some simple observations about search trees on G. First, since P is a path, the projection $G|_P$ is a *binary search tree*, called the *spine search tree*. Second, let ℓ be a leg node and let s be the adjacent spine vertex in G. Then ℓ has at most one child in the search tree (since ℓ is a leaf in G). Further, if ℓ has descendants, then s must be among them. If it does not have descendants, it must be a child of s.

We can see from these observations that every search tree T on G has roughly the following form. Start with some binary search tree S on P. Then put each leg node in one of three places: Either make it the child of its adjacent spine node; or put it *on* some edge in S, to form a degenerate subtree with other leg nodes there; or put it above S to form a degenerate prefix of T with other leg nodes. Figure 12.2 shows three examples.

We study the rotation distance between the following two search trees on G. First, a search tree T where all legs are above the spine search tree. Second, a search tree T' where all leg nodes are leaves. See figure 12.2 (center, right) for illustrations.

Consider the task of transforming T into T', with the following restrictions. First, rotations between leg nodes are not allowed. Second, there can be at most one leg node that has both spine ancestors and spine descendants (i.e., that is "on" a spine search tree edge). All other leg nodes are at the top or bottom.

Let π be the sequence of leg nodes in T, read from bottom to top, and let σ be the sequence obtained by replacing each leg node in π with its adjacent spine node. Observe that transforming T into T' with the mentioned restrictions is almost exactly the same as *accessing* σ in the dynamic search tree model. Indeed, we can alternate between performing some rotations in the spine search tree and moving the next leg node to the bottom (i.e., below its adjacent spine node). See figure 12.3. Thus, we have $d(T, T') \leq \text{DynOPT}(H, \sigma) + \mathcal{O}(n)$.[1]

Unfortunately, this observation alone does not let us bound $\text{diam}(\mathcal{R}(G))$. Lower bounds on $\text{DynOPT}(H, \sigma)$ are not guaranteed to hold for $d(T, T')$, and T, T' may not be a maximum-distance pair of search trees.

[1]The $\mathcal{O}(n)$ term comes from the fact that in the dynamic search tree model, we are allowed to choose the initial search tree and the final search tree is arbitrary, but here, both are given.

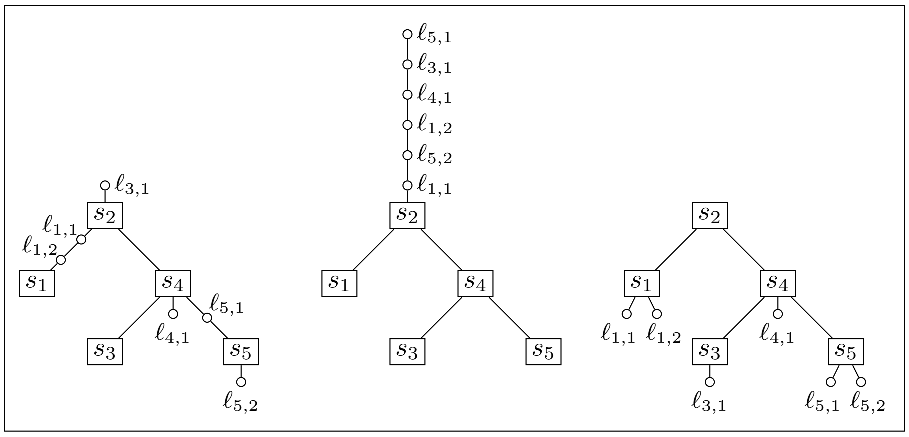


Figure 12.2.: Three search trees on the caterpillar in figure 12.1.

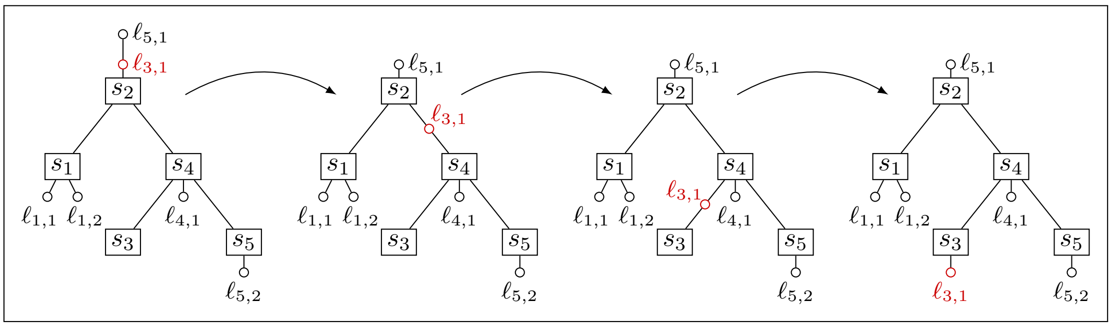


Figure 12.3.: Rotating a leg node downwards simulates a search. See figure 12.1 for the underlying caterpillar graph.

However, the general idea of using dynamic search tree theory is still useful. On one hand, we show that a well-known dynamic BST lower bound due to Wilber [Wil89], the *interleave lower bound*, still holds for $d(T, T')$, and thus for $\operatorname{diam}(\mathcal{R}(G))$. On the other hand, we show that $\operatorname{diam}(\mathcal{R}(G)) \in \mathcal{O}(d(T, T') + n)$, for a worst-possible such pair T, T'. This gives us bounds that are tight up to a constant factor. Choosing a worst-possible pair T, T' involves choosing π (and thus σ) such that $\operatorname{DynOPT}(H, \sigma) \approx \operatorname{StOPT}(H, w)$, where the weight $w(v)$ of each spine node is the number of adjacent leaves, i.e., the number of times v appears in σ.

The author proved these bounds in a recent paper [Ber22] that is the base of this chapter. We expand the result considerably. Instead of single leg nodes, we allow whole connected subgraphs that are attached to a single spine vertex. The following theorem captures the upper and lower bound.

Theorem 12.1. *Let G be a connected graph and let H be a path in G such that each component of $G - H$ is 1-cut and convex in G. For each vertex $v \in V(H)$, let $\mathcal{C}_v \subseteq \mathbb{C}(G - H)$ be the set of components of $G - H$ that are adjacent to v, and let $w(v) = \sum_{C \in \mathcal{C}_v} |V(C)|$. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \geq \tfrac{1}{2} \operatorname{StOPT}(H, w) - \tfrac{3}{2} n + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C)), \textit{ and}$$

$$\operatorname{diam}(\mathcal{R}(G)) \leq 2 \operatorname{StOPT}(H, w) + 10n + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C)).$$

Repeatedly applying theorem 12.1 gives the algorithm of theorem 11.9. The proof involves a graph invariant called the *Strahler number*, and its connection to path-width (see section 12.5).

A single application of theorem 12.1 yields theorem 11.10, the bound for *caterpillar graphs*. Indeed, if H is the spine of a caterpillar G, then each $C \in \mathbb{C}(G - H)$ consists of a single vertex, hance $\operatorname{diam}(\mathcal{R}(C)) = 0$.

It turns out that the lower bound of theorem 12.1 can be generalized by using more general subgraphs as spines (see section 12.2 for the precise statement). Most of the observations above still hold, just that instead of using dynamic *BSTs*, we need use dynamic *STTs*, which makes some proofs more technically difficult. The upper bound does not work in this setting.

Still, the generalized lower bound can be used in tandem with the simple static optimum upper bound from the previous chapter (theorem 11.5) to prove the following result (already mentioned in chapter 11).

Theorem 11.8. *Let G be a tree with no vertices of degree two. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \in \Theta(\operatorname{StOPT}(G, \mathbb{1})).$$

Organization of this chapter. Section 12.1 introduces definitions and observations that are useful throughout the chapter. In section 12.2, we prove the lower bound of theorem 12.1, using a generalization of the interleave bound to STGs. In section 12.3, we prove theorem 11.8 as a relatively simple application of the lower bound. In section 12.4, we prove the upper bound of theorem 12.1, and in section 12.5, we prove our main theorem 11.9, using theorem 12.1.

Finally, in section 12.6, we show that our generalization of the interleave bound also applies to the dynamic STG model. We then use that lower bound to show that the *static finger bound* is unachievable in the general STT setting (see section 8.1).

12.1. Preliminaries

spine

limb

Spines, joints, and limbs. Let G be a connected graph and H be a connected subgraph of G. We call H a *spine* of G if every component $C \in \mathbb{C}(G - H)$ is 1-cut and convex in G. Observe that the spine itself is also convex. We call the components of $G - H$ *limbs*.

spine search tree

joint

With a fixed spine H, vertices of H are called *spine vertices*, and all other vertices are called *limb vertices*. Similarly, nodes of search trees on G or subgraphs of G are called *spine nodes* and *limb nodes* accordingly. Given a search tree T on G, the search tree $T|_H$ is called the *spine search tree* of T. For each limb vertex ℓ, the limb C containing ℓ is the *limb of* ℓ. The unique vertex $v \in V(H)$ that is adjacent to C is called the *joint* of ℓ (or C). See figure 12.4 for an illustration.

Observation 12.2. *Every connected subgraph of a tree G is a spine of G.*

$\sigma_{G/H}(\pi)$

$w_{G/H}$

Let H be a spine of G and let $\pi \in V(G - H)^m$ be a sequence of limb vertices. Then $\sigma_{G/H}(\pi)$ denotes the sequence obtained from π by replacing each vertex v with its joint. Let $w_{G/H} \colon V(H) \to \mathbb{N}_0$ be the weight function on H that indicates the number of limb vertices in G whose joint is a given vertex. In other words, if $\mathcal{C}_v$ is the set of limbs adjacent to v, then $w_{G/H}(v) = \sum_{C \in \mathcal{C}_v} |V(C)|$. See again figure 12.4. Note that the weight function w in theorem 12.1 is precisely $w_{G/H}$.

Rotations in prefixes and suffixes. Our upper bounds sometimes involve “isolating” a set of vertices into a prefix or suffix of the search tree to handle them separately (cf. lemma 11.23). The following lemma will be useful.

Lemma 12.3. *Let T be a search tree on G and let $U \subseteq V(G)$ induce a prefix or suffix of T. After a rotation between two nodes in U, the set U still induces a prefix (resp., suffix) of T.*

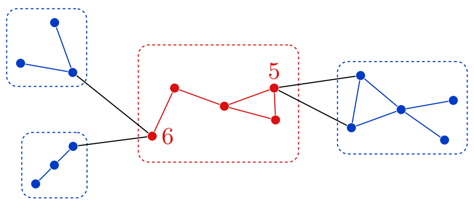


Figure 12.4.: A graph with spine (red) and three limbs (blue). The small numbers indicate the values of $w_{G/H}$.

Proof. By lemma 2.10, no node outside U can gain or lose descendants from the rotation. If U induces a prefix, then no node outside U is an ancestor of a node in U in T, and the rotation does not change that.

Now assume U induces a suffix, and the rotated nodes are u, u'. By assumption, we have $V(T_u), V(T_{u'}) \subseteq U$. By lemma 2.10, the same is true after the rotation, and for all other nodes, the set of descendants does not change. $\square$

Projected rotation distance. Let T, T' be search trees on a graph G, and let $U \subseteq V(G)$. Let the *U-projected rotation distance* $d_U(T,T')$ between T and T' be the minimum number of rotations between two nodes in U in a sequence of rotations transforming T to T'.

U-projected rot. distance

$d_U(T,T')$

Note that we consider arbitrary sequences of rotations here – we just do not count rotations that involve nodes outside of U. In particular, we have $d_U(T,T') = 0$ if T can be transformed into T' without any rotations between nodes in U.

Observe that always $d(T,T') \geq d_U(T,T') + d_{V(G)\setminus U}(T,T')$, but that may be a weak lower bound. It turns out that the projected rotation distance is strongly related to the rotation distance between projections, in the following way.

Lemma 12.4. *Let T, T' be search trees on a graph G, and let $A \subseteq V(G)$ induce a convex subgraph. Then $d_A(T,T') \geq d(T|_A, T'|_A)$.*

Proof. Let R be a sequence of rotations transforming T into T'. Let R_A be the restriction of R to rotations between nodes in A. By lemma 2.20, applying R_A to $T|_A$ is possible and yields $T'|_A$. Thus $d(T|_A, T'|_A) \leq d_A(T,T')$. $\square$

It can be seen that lemma 12.4 holds with equality; we omit the proof. The following is an easy consequence.

Corollary 12.5. *Let T, T' be search trees on a graph G, and let $A \subseteq V(G)$ such that each component of $G[A]$ is a convex subgraph of G. Then*

$$d_A(T,T') \geq \sum_{C \in \mathbb{C}(G[A])} d(T|_C, T'|_C).$$

Proof. We clearly have $d_A(T,T') \geq \sum_{C \in \mathbb{C}(G[A])} d_{V(C)}(T,T')$, and lemma 12.4 implies $d_{V(C)}(T,T') \geq d(T|_C, T'|_C)$. $\square$

12.2. Lower bound

In this section, we show the following lower bound.

Theorem 12.6. *Let G be a connected graph with n vertices and H be a spine of G that is a tree. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \geq \tfrac{1}{2}\operatorname{StOPT}(H, w_{G/H}) - \tfrac{3}{2}n + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C))$$

Observe that in theorem 12.6, the spine may be an arbitrary tree, not just a path; the lower bound of theorem 12.1 follows as a special case. As we will see later, the restriction of H to a tree is necessary.

The proof is based on a technique involving the so-called *alternation number*, introduced by Cardinal, Langerman, and Pérez-Lantero [CLPL18] in their $\Omega(n \log n)$ lower bound for the class of trees (theorem 11.11). We roughly follow their proof, which corresponds to the special case where the underlying graph is a *perfect binary tree*. Extending the proof to arbitrary graphs with tree spines makes it considerably more complicated. In particular, it involves the *interleave lower bound*, one of the two famous dynamic BST lower bounds due to Wilber [Wil89] (see section 8.1). We use a generalization of the interleave lower bound for the rotation distance between certain search trees.

We start by formalizing the alternation number technique [CLPL18, CPVP22] in section 12.2.1. Based on this technique, we prove the interleave lower bound for rotation distance in section 12.2.2. This allows us to bound the rotation distance between arbitrary search trees. In section 12.2.3, we determine search trees that maximize the interleave lower bound, thus providing a good lower bound for the diameter. Finally, in section 12.2.4, we put everything together to prove theorem 12.6.

As we have sketched in the beginning of the chapter, we can simulate a dynamic search tree access as a series of rotations; this allows us to show that the interleave lower bound also holds in the dynamic *STG* model in section 12.6 (although it will turn out to be useless in very dense graphs). The interleave lower bound was previously generalized to the *STT* model [BCI$^+$20] with a different proof.

12.2.1. Rotation distance and alternation number

$\operatorname{alt}(\sigma, \mathcal{X})$

We start with some definitions. Let σ be a sequence over a set A, and let $\mathcal{X}$ be a partition of A. We define $\operatorname{alt}(\sigma, \mathcal{X})$ to be the number of adjacent pairs (x, y) in σ such that x and y are in different parts of the partition $\mathcal{X}$.

alternation number $\operatorname{alt}(T, \mathcal{X})$

Let T be a search tree on a graph G and let $\mathcal{X}$ be a partition of $V(G)$. Define the *alternation number* of T as $\operatorname{alt}(T, \mathcal{X}) = \max_{v \in V(T)} \operatorname{alt}(\pi_{T,v}, \mathcal{X})$, where $\pi_{T,v}$ is the root path of v in T. The partition $\mathcal{X}$ can be seen as a *coloring* of $V(G)$; see figure 12.5 for an example. We now relate alternation number and rotation distance.

Lemma 12.7 (Cardinal, Pournin, and Valencia-Pabon [CPVP22]). *Let T be search tree on a graph G and let T' be the search tree obtained from T by rotating two nodes u, v. Let $\mathcal{X}$ be a partition of $V(G)$. If u and v belong to the same part of $\mathcal{X}$, then* $\operatorname{alt}(T, \mathcal{X}) = \operatorname{alt}(T', \mathcal{X})$. *Otherwise,* $|\operatorname{alt}(T, \mathcal{X}) - \operatorname{alt}(T', \mathcal{X})| \leq 2$.

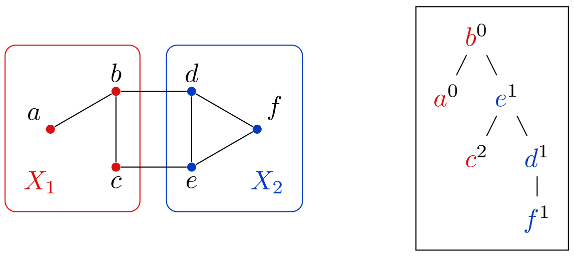


Figure 12.5.: A graph G with a partition $\mathcal{X} = \{X_1, X_2\}$ (left), and a search tree T on G (right). The numbers next to the the search tree nodes indicate the alternation number $\operatorname{alt}(\pi_{T,v}, \mathcal{X})$ of that node. We have $\operatorname{alt}(T, \mathcal{X}) = 2$.

Proof. Consider a node x. The root path of x before and after the rotation can only differ if it contains u or v (or both), and only in one of the following ways:

- One of u and v is inserted, such that u, v are neighbors in T'.
- One u and v is deleted, and u, v are neighbors in T.
- u and v are swapped, and are neighbors (in both search trees).

Our claim easily follows in each case. □

Lemma 12.8. *Let T, T' be search trees on a connected graph G, and let $\mathcal{X}$ be a partition of $V(G)$ such that $G[X]$ is convex for each $X \in \mathcal{X}$. Then*

$$d(T, T') \geq \tfrac{1}{2}|\operatorname{alt}(T, \mathcal{X}) - \operatorname{alt}(T', \mathcal{X})| + \sum_{X \in \mathcal{X}} d(T|_X, T'|_X).$$

Proof. Let R be a sequence of rotations that transforms T into T'. We first consider rotations between nodes of two different parts of $\mathcal{X}$. By lemma 12.7, there must be at least $\frac{1}{2}|\operatorname{alt}(T, \mathcal{X}) - \operatorname{alt}(T', \mathcal{X})|$ such rotations.

Now consider rotations between nodes within some part $X \in \mathcal{X}$. There must be at least $d_X(T, T')$ such rotations, and we have $d_X(T, T') \geq d(T|_X, T'|_X)$ by lemma 12.4. Summing up everything yields the desired lower bound. □

Remark. The previous two lemmas can be strengthened by replacing the alternation number $\operatorname{alt}(T, \mathcal{X})$ by the *number of bichromatic edges* in T, i.e., the number of edges between a parent from one part of $\mathcal{X}$ and a child from another. This only works if the underlying graph is a tree; otherwise, a single rotation could create or destroy an arbitrary number of bichromatic edges. It is not clear if we can prove stronger bounds with counting bichromatic edges; we use the alternation number here, since it works for general graphs.

12.2.2. The interleave lower bound

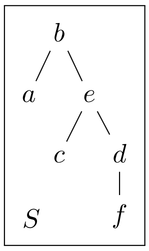
In this section, we introduce the interleave lower bound and show how it applies to rotation distance. Let S be an arbitrary rooted tree and let σ be a sequence of nodes in $V(G)$ (possibly with repetition). Let $v \in V(S)$ and K be the set of children of v in S.

$\lambda(S,v,\sigma)$

$\Lambda(S,\sigma)$

interleave bound

Then we define $\lambda(S,v,\sigma) = \text{alt}(\sigma|_{V(S_v)}, \mathcal{X})$, where $\mathcal{X} = \{\{v\}\} \cup \{V(S_c) \mid c \in K\}$. Let $\Lambda(S,\sigma) = \sum_{v \in V(S)} \lambda(S,v,\sigma)$. We call $\Lambda(S,\sigma)$ the *interleave bound.* See below for an example calculation.



$\sigma = a\,b\,c\,d\,e\,a$
$\lambda(S,b,\sigma) = \text{alt}(a\,b\,c\,d\,e\,a, \{\{b\},\{a\},\{c,d,e,f\}\}) = 3$
$\lambda(S,e,\sigma) = \text{alt}(c\,d\,e, \{\{e\},\{c\},\{d,f\}\}) = 2$
$\lambda(S,d,\sigma) = \text{alt}(d, \{\{d\},\{f\}\}) = 0$
$\lambda(S,a,\sigma) = \lambda(S,c,\sigma) = \lambda(S,f,\sigma) = 0$
$\Lambda(S,\sigma) = 3 + 2 = 5$

In the remainder of the section, we prove:

Lemma 12.9. *Let G be a connected graph on n vertices and let H be a spine of G. Let S be a search tree on H and let T, T' be search trees on G such that S is a suffix of T and a prefix of T'. Suppose that $T|_{G-H}$ is a degenerate search tree, and let π be a topological ordering of $T|_{G-H}$. Then*

$$d(T,T') \geq \tfrac{1}{2}\Lambda(S, \sigma_{G/H}(\pi)) + \sum_{C \in \mathbb{C}(G-H)} d(T|_C, T'|_C).$$

Observe that lemma 12.9 applies to the process discussed in the beginning of the chapter (pages 175 to 177): The search trees T and T' correspond to the “extremal” search trees in figure 12.2 (center, right).

We prove lemma 12.9 in multiple steps. First, we define a partition $\mathcal{X}$ of $V(G)$ so that we can apply lemma 12.8.

Fix H, G, S, T, T', and π as in the lemma. Let $r = \text{root}(S)$ and let $X_r \subseteq V(G)$ consist of r and all limb vertices with joint r, i.e., all vertices in a components $C \in \mathbb{C}(G-H)$ such that C is adjacent to r. For each component D of $H - r$, let $X_D \subseteq V(G)$ consist of $V(D)$ and all limb vertices whose joint is contained in $V(D)$. Observe that X_D induces a connected component of $G - r$. See figure 12.6 for an illustration.

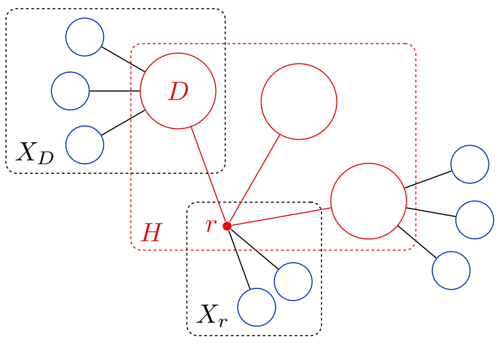
Let $\mathcal{X} = \{X_r\} \cup \{X_D \mid D \in \mathbb{C}(H-r)\}$, and observe that $\mathcal{X}$ is a partition of $V(G)$. Further observe that there are no edges between X_D and $X_{D'}$ for $D, D' \in \mathbb{C}(H-r)\}$, and there is precisely one edge between X_r and each X_D. Thus, all $X \in \mathcal{X}$ are convex and we can apply lemma 12.8 to obtain

$$d(T,T') \geq \tfrac{1}{2}|\text{alt}(T,\mathcal{X}) - \text{alt}(T',\mathcal{X})| + \sum_{X \in \mathcal{X}} d(T|_X, T'|_X). \tag{12.1}$$

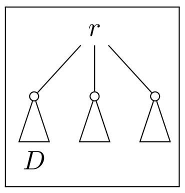
We first consider the quantity $|\text{alt}(T,\mathcal{X}) - \text{alt}(T',\mathcal{X})|$. In the following, let $\sigma = \sigma_{G/H}(\pi)$. Recall that σ is obtained from π by mapping each limb vertex to its joint. Since each limb vertex belongs to the same part of $\mathcal{X}$ as its joint, we have $\text{alt}(\sigma,\mathcal{X}) = \text{alt}(\pi,\mathcal{X})$.

Claim 12.10. $\text{alt}(T,\mathcal{X}) \geq \lambda(S,r,\sigma)$, *and* $\text{alt}(T,\mathcal{X}) \geq \lambda(S,r,\sigma) + 1$ *if* $X_r = \{r\}$.

Proof. By definition, we have $\lambda(S,r,\sigma) = \text{alt}(\sigma,\Pi)$, where Π is the partition of $V(S) = V(H)$ into $\{r\}$ and the components of $H - r$. Note that Π is precisely the refinement of $\mathcal{X}$ to $V(H) \supseteq V(\sigma)$, so we have $\lambda(S,r,\sigma) = \text{alt}(\sigma,\Pi) = \text{alt}(\sigma,\mathcal{X}) = \text{alt}(\pi,\mathcal{X})$.



(a) Underlying graph G



(b) Spine search tree S

Figure 12.6.: Example of G, H, and S in the proof of lemma 12.9.

Now recall that π is a topological ordering of the degenerate search tree $T|_{G-H}$. Clearly, some root path of T contains π as a subsequence, so $\operatorname{alt}(T, \mathcal{X}) \geq \operatorname{alt}(\pi, \mathcal{X})$.

For the case $X_r = \{r\}$, we make the following additional observations. Since S, a search tree on H, is a suffix of T, we know that $T|_{G-H}$ must be a *prefix* of T. Moreover, the child of the lowest node of $T|_{G-H}$ is r, which gives us one additional alternation (since the lowest node of $T|_{G-H}$ is clearly not in X_r). Thus, we have $\operatorname{alt}(T, \mathcal{X}) \geq \lambda(S, r, \sigma) + 1$. ☐

Claim 12.11. $\operatorname{alt}(T', \mathcal{X}) \leq 1$.

Proof. Since S is a prefix of T', we have $\operatorname{root}(T') = r$. The child subtrees of r in T' are the components of $G - r$, consisting of

- $G[X_D]$ for each $D \in \mathbb{C}(H - r)$, and
- the components of $G - H$ that are adjacent to r, all of which are contained in X_r.

Thus, the only alternations in T' are between r and some of its children, hence $\operatorname{alt}(T', \mathcal{X}) \leq 1$. ☐

Claims 12.10 and 12.11 together imply that

$$|\operatorname{alt}(T, \mathcal{X}) - \operatorname{alt}(T', \mathcal{X})| \geq \begin{cases} \lambda(S, r, \sigma), & \text{if } X_r = \{r\}, \\ \lambda(S, r, \sigma) - 1, & \text{otherwise.} \end{cases} \tag{12.2}$$

We now move on to the second part of eq. (12.1) and bound $d(T|_X, T'|_X)$ for each $X \in \mathcal{X}$. We start with $X = X_r$.

Claim 12.12. *If $X := X_r = \{r\}$, then $d(T|_X, T'|_X) = 0$. Otherwise,*

$$d(T|_X, T'|_X) \geq 1 + \sum_{C \in G[X]-r} d(T|_C, T'|_C).$$

Proof. If $X = \{r\}$, then $T|_X = T'|_X$, so the first part is trivially true. Otherwise, there must be at least one rotation between r and some other node in X (since r is not the root of $T|_X$, but *is* the root of $T'|_X$). Thus, we have

$$d(T|_X, T'|_X) \geq 1 + d_{X \setminus \{r\}}(T|_X, T'|_X).$$

The claim follows by applying corollary 12.5. (Observe that $X \setminus \{r\}$ induces a collection of limbs, which are all convex by definition.) □

We now turn to the parts $X \in \mathcal{X} \setminus \{X_r\}$. Here, we use induction.

Claim 12.13. *Let c be a child of r in S, and let $D = G[S_c]$. Then*

$$d(T|_{X_D}, T'|_{X_D}) \geq \tfrac{1}{2}\Lambda(S_c, \sigma|_D) + \sum_{C \in \mathbb{C}(G[X_D]-D)} d(T|_C, T'|_C).$$

Proof. We want to inductively apply lemma 12.9 with $G \leftarrow G[X_D]$, $H \leftarrow D$, $S \leftarrow S|_D$, $T \leftarrow T|_{X_D}$, $T' \leftarrow T'|_{X_D}$, and $\pi \leftarrow \pi|_{X_D}$. First, we need to show that the necessary conditions hold.

Indeed, observe that $S|_D$ is a suffix of $T|_{X_D}$ and a prefix of $T'|_{X_D}$, since S is a suffix of T and a prefix of T'. Moreover, clearly $(T|_{X_D})|_{X_D \setminus V(D)} = T|_{X_D \setminus V(D)}$ is degenerate, since $T|_{G-H}$ is degenerate and $X_D \setminus V(D) \subseteq V(G-H)$. Similarly, we can show that $\pi|_{X_D}$ is a topological ordering of $T|_{X_D \setminus V(D)}$. Finally, D is clearly a spine of $G[X_D]$.

Applying lemma 12.9 yields the following. We write $X = X_D$ for brevity.

$$d(T|_X, T'|_X) \geq \tfrac{1}{2}\Lambda(S|_D, \sigma_{G[X]/D}(\pi|_X)) + \sum_{C \in \mathbb{C}(G[X]-D)} d(T|_C, T'|_C).$$

Now only some small observations remain. First, observe that $S_c = S|_D$ by definition. Second, observe that $\sigma_{G[X]/D}(\pi|_X)) = \sigma_{G/H}(\pi|_X)$; this is because for each $x \in X \setminus V(D)$, the joint of x in $G[X]$ w.r.t. spine D is the same as the joint of x in G w.r.t. spine H, hence $\sigma_{G[X]/D}$ and $\sigma_{G/H}$ map x to the same vertex. Third, we have $\sigma_{G/H}(\pi|_X) = \sigma_{G/H}(\pi)|_D$ for similar reasons, and recall that $\sigma = \sigma_{G/H}(\pi)$. The claimed inequality follows. □

We are now ready to finish the proof. First, by combining eq. (12.2) with claim 12.12 and separately treating the cases $X_r = \{r\}$ and $X_r \supsetneq \{r\}$, we obtain

$$\tfrac{1}{2}|\mathrm{alt}(T, \mathcal{X}) - \mathrm{alt}(T', \mathcal{X})| + d(T|_{X_r}, T'|_{X_r}) \geq \tfrac{1}{2}\lambda(S, r, \sigma) + \sum_{C \in G[X_r]-r} d(T|_C, T'|_C). \tag{12.3}$$

Second, if K is the set of children of r in S, the following holds by definition of Λ.

$$\Lambda(S, \sigma) = \lambda(S, r, \sigma) + \sum_{c \in K} \Lambda(S_c, \sigma|_{V(S_c)}). \tag{12.4}$$

Overall, we have

$$\begin{aligned}
d(T, T') &\geq \tfrac{1}{2}|\mathrm{alt}(T, \mathcal{X}) - \mathrm{alt}(T', \mathcal{X})| + \sum_{X \in \mathcal{X}} d(T|_X, T'|_X). \\
&\geq \tfrac{1}{2}\lambda(S, r, \sigma) + \sum_{C \in G[X_r]-r} d(T|_C, T'|_C) + \sum_{X \in \mathcal{X} \setminus \{X_r\}} d(T|_X, T'|_X) && \text{by eq. (12.3)} \\
&\geq \tfrac{1}{2}\Lambda(S, \sigma) + \sum_{C \in G[X_r]-r} d(T|_C, T'|_C) + \sum_{\substack{D \in \mathbb{C}(H-r) \\ C \in \mathbb{C}(G[X_D]-D)}} d(T|_C, T'|_C) && \text{by claim 12.13, eq. (12.4)} \\
&= \tfrac{1}{2}\Lambda(S, \sigma) + \sum_{C \in G-H} d(T|_C).
\end{aligned}$$

This concludes the proof of lemma 12.9.

12.2.3. Maximum interleave sequences

In this section, we show how to find search trees S and sequences σ that maximize the lower bound in lemma 12.9. We start with a way of interleaving sequences such that the alternation number is maximized.

maximum interleave sequence

The *maximum interleave sequence* of sequences $\sigma_1, \sigma_2, \ldots, \sigma_k$ is constructed as follows. Start with an empty sequence. In each step, we pick the longest sequence that we did not choose in the last step (break ties by picking the sequence with lowest index), remove its first element, append that element to our result, and continue. At some point, we may just have one sequence σ'_i left from which we just picked an element. In that case, append the whole remaining sequence σ'_i.

Lemma 12.14. *Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be sequences over disjoint sets $A_1, A_2, \ldots, A_k$, and let μ be their maximum interleave sequence. Let $m_i = |\sigma_i|$, let $m = |\mu| = \sum_{i=1}^{k} m_i$. Then*

$$\mathrm{alt}(\mu, \{A_1, A_2, \ldots, A_k\}) \geq \min\left(m - 1, 2 \cdot (m - \max_{i \in [k]} m_i)\right).$$

Proof. We write $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ for convenience.

Suppose first that each step *succeeds* in the sense that we can always pick an element from a sequence that was not used in the previous step. Then any two adjacent elements come from different sequences, so we have $\mathrm{alt}(\mu, \mathcal{A}) = m - 1$ and are done.

Now suppose that after some step $t < m$, only one sequence, say the i-th, has $k = m - t$ elements left. Since all previous steps where successful, this means that $\mathrm{alt}(\mu, \mathcal{A}) = t - 1$.

We denote by σ_j^s the remainder of the j-th sequence σ_j after step s. We claim that after every step $s \leq t$, we have $|\sigma_i^s| > |\sigma_j^s|$ for all $j \neq i$. Suppose this is true. Then the construction process picks an element of σ_i whenever possible; this includes the first step, each odd-numbered step afterwards, and in particular step t. Thus t is odd and we have

$$m_i = |\sigma_i| = k + \lceil \tfrac{t}{2} \rceil = (m - t) + \tfrac{t+1}{2} = m - \tfrac{t-1}{2}.$$

This implies that $2(m - m_i) = t - 1 = \mathrm{alt}(\mu, \mathcal{A})$, as desired.

We now prove the claim by induction on the step number s. We actually show the following more specific fact: For all $j \neq i$, we have $|\sigma_i^s| \geq |\sigma_j^s| + 1$ if s is odd and $|\sigma_i^s| \geq |\sigma_j^s| + 2$ if s is even.

If $s = t$, then s is odd, and we have $|\sigma_i^s| = k \geq 1$ and $|\sigma_j^s| = 0$ for all $j \neq s$.

Now suppose the claim holds for s. We prove the claim for $s - 1$. If s is odd, then step s removes an element from σ_i, thus $|\sigma_i^{s-1}| = |\sigma_i^s| + 1 \geq |\sigma_j^s| + 2 = |\sigma_j^{s-1}| + 2$ for all $j \neq i$. If s is even, then step s removes an element from some σ_j with $j \neq i$, but we still have $|\sigma_i^{s-1}| = |\sigma_i^s| \geq |\sigma_j^s| + 2 \geq |\sigma_j^{s-1}| + 1$ for all $j \neq i$. □

We now apply lemma 12.14 to maximize the alternation bound. Curiously, the *centroid tree* (see chapter 4) makes another appearance here.

$\lambda'(S, v, \sigma)$

$\Lambda'(S, \sigma)$

For convenience, define $\lambda'(S, v, \sigma)$ as $\lambda(S, v, \sigma)$ plus the number of occurrences of v in σ, and let $\Lambda'(S, \sigma) = \sum_{v \in V(S)} \lambda'(S, v, \sigma) = \Lambda(S, \sigma) + |\sigma|$.

Lemma 12.15. *Let (G, w) be a weighted tree with only integer weights and let $m = \sum_{v \in V(G)} w(v)$. Let T be a centroid tree of (G, w). Then there is a sequence σ of length m where each $v \in V(G)$ occurs precisely $w(v)$ times, such that $\Lambda'(T, \sigma) \geq \mathrm{cost}(T, w) - |V(G)|$.*

Proof. We recursively construct a sequence σ_v on $V(T_v)$ for each $v \in V(T)$, and show that $\Lambda'(T_v, \sigma_v) \geq \mathrm{cost}(T_v, w) - |V(T_v)|$. In the end, setting $\sigma = \sigma_{\mathrm{root}(T)}$ yields our claim.

Let $v \in V(T)$. If v is a leaf in T, then we let σ_v be the sequence consisting of $w(v)$ times v. Clearly, we have $\Lambda'(T_v, \sigma_v) = \lambda'(T, v, \sigma_v) = |\sigma_v| = w(v) = \mathrm{cost}(T_v, w)$.

Otherwise, let K be the set of children of v. Let σ'_v be the sequence consisting of $w(v)$ times v, and let σ_v be the maximum interleave sequence of σ'_v and all σ_c for $c \in K$. Note that, since T is a centroid tree, we have $|\sigma_c| = w(T_c) \leq \frac{1}{2}w(T_v)$. We now apply lemma 12.14 and distinguish two cases. First, if $w(v) \leq \frac{1}{2}w(T_v)$, then

$$\lambda(T, v, \sigma_v) \geq \min(w(T_v) - 1, 2(w(T_v) - \tfrac{1}{2}w(T_v))) = w(T_v) - 1.$$

If instead $w(v) > \frac{1}{2}w(T_v)$, then

$$\lambda(T, v, \sigma_v) \geq \min(w(T_v) - 1, 2(w(T_v) - w(v))) \geq w(T_v) - w(v) - 1.$$

In both cases, we have $\lambda'(T, v, \sigma_v = \lambda(T, v, \sigma_v) + w(v) \geq w(T_v) - 1$.

Note that for each $c \in K$, we have $\sigma_c = \sigma_v|_{V(T_c)}$, and thus $\Lambda(T_c, \sigma_v) = \Lambda(T_c, \sigma_c)$. By induction, we have

$$\begin{aligned}\Lambda'(T, v, \sigma_v) &= \lambda'(T, v, \sigma_v) + \sum_{c \in K} \Lambda'(T_c, \sigma_v)\\ &\geq w(T_v) - 1 + \sum_{c \in K} \mathrm{cost}(T_c, w) - |V(T_c)| = \mathrm{cost}(T_v, w) - |V(T_v)|.\end{aligned}$$

This concludes the proof. □

12.2.4. Completing the lower bound

In this section, we prove:

Theorem 12.6. *Let G be a connected graph with n vertices and H be a spine of G that is a tree. Then*

$$\mathrm{diam}(\mathcal{R}(G)) \geq \tfrac{1}{2}\,\mathrm{StOPT}(H, w_{G/H}) - \tfrac{3}{2}n + \sum_{C \in \mathbb{C}(G-H)} \mathrm{diam}(\mathcal{R}(C))$$

Fix a connected graph G with spine H. To prove the lower bound, we construct two search trees T and T' akin to the two "extremal" search trees in figure 12.2 (center, right). The limb vertices $V(G-H)$ form a degenerate prefix of T and a suffix of T'. The spine search tree of both T and T' will be the centroid tree of $(H, w_{G/H})$ and the order of the vertices in the degenerate prefix of T are chosen with lemma 12.15. We then use lemma 12.9 to lower bound the distance between T and T'.

The construction of T in particular is more complicated than in the case where G is a caterpillar. The main reason is that the degenerate prefix no longer simply consists of *leaves* of G.

We now proceed with the construction of the two search trees. Fix G and its spine H for the remainder of the section, and let $n = |V(G)|$, $k = |V(H)|$, and $m = n-k = |V(G-H)|$. Figure 12.7 shows an example for both constructions.

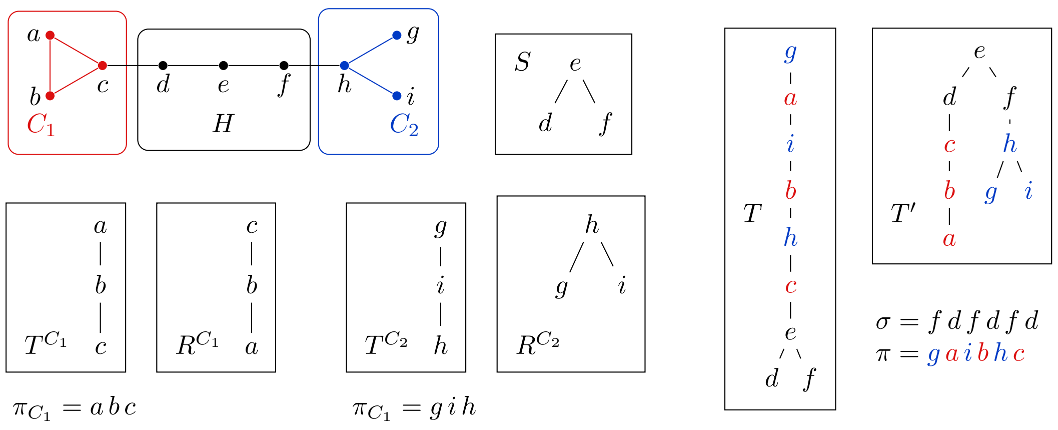


Figure 12.7.: Construction of search trees T and T'.

The first search tree. Let the following be given:

- A search tree S on H.
- A sequence σ of length m such that each $v \in V(H)$ occurs $w_{G/H}$ times in σ.
- For each limb $C \in \mathbb{C}(G-H)$, a degenerate search tree T^C on C, such that the unique leaf of T^C is adjacent to $V(H)$ in G.

For each limb $C \in \mathbb{C}(G-H)$, let π_C be the unique topological ordering of T^C, and let v_C be the joint of C. Obtain a permutation π of $V(G)$ as follows. For each limb C and each $i \in [|V(C)|]$, we replace the i-th occurrence of v_C in σ with the i-th element in π_C. Let the permutation τ be obtained by concatenating π with an arbitrary topological ordering of S, and define $A(S, \sigma, (T^C)_{C \in \mathbb{C}(G-H)})$ as the search tree with topological ordering τ. $\boxed{A(\cdot,\cdot,\cdot)}$

We now show some facts about T in preparation of applying lemma 12.9. Let $T = A(S, \sigma, (T^C)_{C \in \mathbb{C}(G-H)})$ in the following.

Observation 12.16. *S is a suffix of T.*

Observation 12.17. *For each limb $C \in \mathbb{C}(G-H)$, we have $T|_C = T^C$.*

Lemma 12.18. *$V(G-H)$ induces a degenerate prefix P of T.*

Proof. Since $V(G-H)$ induces a prefix of the topological ordering τ of T by construction, it also induces a prefix P of T (lemma 2.15). It remains to show that P is degenerate.

Suppose P is not degenerate. Then there must be two nodes $u, v \in V(P)$ that are incomparable under $\prec_P$ (and thus under $\prec_T$). First suppose that u, v are contained in the same limb $C \in \mathbb{C}(G-H)$. Since $P' = P|_C$ is degenerate, we have $u \prec_{P'} v$ or $v \prec_{P'} u$. But then also $u \prec_P v$ or $v \prec_P u$ by lemma 2.18, contradicting our assumption.

Now suppose that $u \in V(C_u)$ and $v \in V(C_v)$ for two distinct limbs $C_u, C_v \in \mathbb{C}(G-H)$. Without loss of generality, assume that u is the unique leaf of $T|_{C_u}$ and v is the unique leaf of $T|_{C_v}$. This means that u and v are both adjacent to $V(H)$ in G by assumption.

Recall that $V(G-H)$ induces a prefix of the topological ordering τ. Since u, v are incomparable under $\prec_T$, corollary 2.13 implies that there is a vertex set A that separates

u from v in G and precedes both in τ. However, since u and v are both adjacent to H, we know that A contains at least one vertex from H. This is a contradiction, since then A cannot precede u and v in τ. □

Lemma 12.19. *Let P be the degenerate prefix of T induced by $V(G-H)$, and let π be the unique topological ordering of T. Then $\sigma = \sigma_{G/H}(\pi)$.*

Proof. The permutation π in the construction is the topological ordering of P, and it is easy to see that $\sigma = \sigma_{G/H}(\pi)$. □

The second search tree. Let the following be given:

- A search tree S on H.
- For each limb $C \in \mathbb{C}(G-H)$, an arbitrary search tree R^C on C.

Intuitively, we construct the second search tree as follows: Take S as a prefix. Then each component C corresponds to its own rooted subtree, which we set to R^C. A formal construction follows.

For each limb C, let ρ_C be a topological ordering of R^C. Let τ' be the concatenation of some topological ordering of S with all ρ_C (the permutations ρ_C can be appended in any order). Finally, let $B(S, (R^C)_{C\in\mathbb{C}(G-H)})$ be the search tree with topological ordering τ'.

Let $T' = B(S, (R^C)_{C\in\mathbb{C}(G-H)})$ in the following.

Observation 12.20. *S is a prefix of T'.*

Observation 12.21. *For each limb $C \in \mathbb{C}(G-H)$, we have $T'|_C = R^C$.*

More about degenerate search trees. The astute reader may have noticed that when applying lemma 12.9 to T, T', the lower bound will contain the distance $d(T^C, R^C)$ for each limb C. Hence, we want to maximize this distance. We cannot just take two search trees, since T^C must be degenerate and its leaf cannot be freely chosen. We now show that this restriction does note strongly limit us.

Lemma 12.22. *Let G be a connected graph on n vertices. For each search tree T on G and each vertex $v \in V(G)$, there is a degenerate search tree T' on G such that $d(T,T') \le n-1$ and v is the unique leaf of T'.*

Proof. We transform T into T' as follows. If $|\operatorname{Path}_T(v)| = n$, then we are done. Otherwise, there are two cases. If v is not a leaf node, rotate some child c of v with v. This increases $|\operatorname{Path}_T(v)|$ by one. Repeat until v is a leaf.

If v is a leaf node and $|\operatorname{Path}_T(v)| < n$, then there must be a node u with $u \notin \operatorname{Path}_T(v)$ and $\operatorname{parent}_T(u) \in \operatorname{Path}_T(v)$. In that case, rotate u with its parent and repeat. Note that this rotation also increases $|\operatorname{Path}_T(v)|$ by one. Thus, we are done after at most $n-1$ steps. □

The proof. We now finally prove:

Theorem 12.6. *Let G be a connected graph with n vertices and H be a spine of G that is a tree. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \geq \tfrac{1}{2}\operatorname{StOPT}(H, w_{G/H}) - \tfrac{3}{2}n + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C))$$

Proof. Let $k = |V(H)|$ and $m = |V(G-H)| = n - k$. For each limb $C \in \mathbb{C}(G-H)$, let T^C be a degenerate search tree on C such that the leaf of T^C is adjacent to $V(H)$ in G, and let R^C be an arbitrary search tree on C. Choose T^C and R^C such that $d(T^C, R^C)$ is maximized, subject to the restrictions on T^C. By lemma 12.22 and the triangle inequality, we have

$$d(T^C, R^C) \geq \operatorname{diam}(\mathcal{R}(C)) - |C| + 1. \tag{12.5}$$

Let S be a centroid tree on $(H, w_{G/H})$. Since H is a tree, S exists. Let σ be the sequence obtained from lemma 12.15. Then $V(\sigma) \subseteq V(H)$, each vertex $v \in V(H)$ occurs precisely $w_{G/H}(v)$ times in σ, and

$$\Lambda'(S, \sigma) \geq \operatorname{cost}(S, w_{G/H}) - |V(H)| \geq \operatorname{StOPT}(H, w_{G/H}) - k. \tag{12.6}$$

Now define $T = A(S, \sigma, (T^C)_{C \in \mathbb{C}(G-H)})$ and $T' = B(S, (R^C)_{C \in \mathbb{C}(G-H)})$. By observations 12.16 and 12.20 and lemmas 12.18 and 12.19, we can apply lemma 12.9 to T and T' and obtain:

$$\begin{aligned}
d(T, T') &\geq \tfrac{1}{2}\Lambda(S, \sigma) + \sum_{C \in \mathbb{C}(G-H)} d(T|_C, T'|_C) \\
&= \tfrac{1}{2}\Lambda(S, \sigma) + \sum_{C \in \mathbb{C}(G-H)} d(T^C, R^C) && \text{by observations 12.17 and 12.21} \\
&\geq \tfrac{1}{2}\Lambda(S, \sigma) + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C)) - |C| && \text{by eq. (12.5)} \\
&= \tfrac{1}{2}\Lambda'(S, \sigma) - \tfrac{3}{2}m + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C)) && \text{by def. of } \Lambda' \\
&= \tfrac{1}{2}\operatorname{StOPT}(H, w_{G/H}) - \tfrac{1}{2}k - \tfrac{3}{2}m + \sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C)) && \text{by eq. (12.6)}
\end{aligned}$$

Since $n = m + k$, this finishes the proof. □

12.3. Trees with no vertices of degree two

Before proving our more complicated lower bounds, we combine the lower bound theorem 12.6 with theorem 11.5 to obtain the following result.

Theorem 11.8. *Let G be a tree with no vertices of degree two. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \in \Theta(\operatorname{StOPT}(G, \mathbb{1})).$$

This is one of the cases where theorem 11.5 is tight. We can re-use the following technical lemma from section 9.3.6.

Lemma 9.32. *Let (G, w) be a connected weighted graph with integral weights. If G has no vertex v with both $\deg_G(v) \leq 2$ and $w(v) = 0$, then $\mathrm{StOPT}(G, w + \mathbb{1}) \leq 10 \cdot \mathrm{StOPT}(G, w)$.*

Proof of theorem 11.8. The upper bound follows from theorem 11.5. For the lower bound, let H be the graph obtained by G by removing each leaf, and define $w \colon V(H) \to \mathbb{N}$ such that $w(v)$ is the number of leaves adjacent to v in G.

Theorem 12.6 implies that $\mathrm{diam}(\mathcal{R}(G)) + \frac{3}{2}n \geq \frac{1}{2}\,\mathrm{StOPT}(H, w)$. Since $\mathrm{diam}(\mathcal{R}(G)) \geq 2n - 20$ by lemma 11.2, this implies $\mathrm{diam}(\mathcal{R}(G)) \in \Omega(\mathrm{StOPT}(H, w))$. It remains to show that $\mathrm{StOPT}(H, w) \in \Omega(\mathrm{StOPT}(G, \mathbb{1})$.

Let $w' = w + \mathbb{1}$. Observe that every vertex $v \in V(H)$ with $w(v) = 0$ has degree at least three. Indeed, $v \in V(H)$ implies that v is not a leaf in G, and $w(v) = 0$ implies that it has no adjacent leaves in G, so $\deg_H(v) = \deg_G(v) \geq 3$. Hence, lemma 9.32 implies that $\mathrm{StOPT}(H, w) \in \Omega(\mathrm{StOPT}(H, w'))$.

Finally, we show that $\mathrm{StOPT}(H, w') \geq \mathrm{StOPT}(G, \mathbb{1})$. Let T' be an optimal search tree on H, and let T be the search tree on G obtained by attaching each leaf ℓ of G as a child to the node $v \in V(H)$ that is adjacent to ℓ in G. Observe that T is indeed a valid search tree on G.

It is easy to see that $\mathrm{cost}(T', w') = \mathrm{cost}(T, \mathbb{1}) + |V(G) \setminus V(H)|$. Essentially, for each node $v \in V(T')$, we transfer $w(v)$ of its weight to its children. This implies that $\mathrm{StOPT}(H, w') > \mathrm{StOPT}(G, \mathbb{1})$, and thus concludes the proof. □

12.4. Upper bound

In this section, we prove the upper bound part of theorem 12.1.

Theorem 12.23. *Let G be a connected graph with n vertices and let H be a spine of G that is a path. Then*

$$\mathrm{diam}(\mathcal{R}(G)) \leq 2\,\mathrm{StOPT}(H, w_{G/H}) + 10n + \sum_{C \in \mathbb{C}(G-H)} \mathrm{diam}(\mathcal{R}(C)).$$

Observe that theorem 12.23 is *almost* a special case of the torso bound (theorem 11.6). Indeed, clearly $H = \mathrm{torso}_G(V(H))$. However, the weight function in theorem 11.6 is actually $w_{G/H} + \mathbb{1}$. We always have $\mathrm{StOPT}(H, \mathbb{1}) \in \Omega(|V(H)| \log |V(H)|)$, so theorem 11.6 may be much weaker than theorem 12.23.

We now outline the proof of theorem 12.23, deferring the details of each step to a later section. Given two search trees T and T' on G, we upper bound $d(T, T')$. We give a series of steps that are applied to both search trees in parallel, until the last step transforms one into the other. See figure 12.8 for an illustration.

First (section 12.4.1), we move the limb nodes “to the top”. Afterwards, the the limb nodes form a prefix, which implies the spine nodes form a suffix (or, more specifically, a rooted subtree). This is possible with only $\mathcal{O}(n)$ rotations.

Lemma 12.24. *Let G be a connected graph with n vertices and let H be a spine of G that is a path. Let T be a search tree on G. Then, with at most $3n$ rotations, we can transform T into a search tree T' such that $V(G - H)$ induces a prefix of T'.*

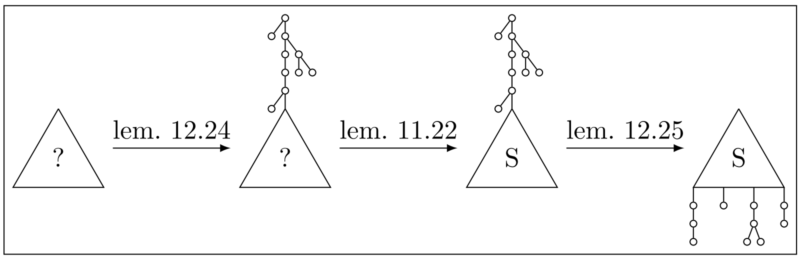


Figure 12.8.: The transformation process for theorem 12.23.

Note that lemma 12.24 is the only part of our proof that requires H to be a path.

The next step is to transform the (now) suffix of G induced by H into a certain search tree S, to be determined later. This is easily done with lemma 11.22.

Now (section 12.4.2), we want to move the limb nodes to the "bottom", i.e., make them a suffix. This may seem counter-intuitive, since we just moved them up (made them a prefix), but that was to enable lemma 11.22, which turns out to be a good preparation for the current step. Intuitively, while the limb nodes are interspersed throughout the tree, they are hard to move to the bottom of the tree, but easy to move to the top. Once there are at the top, we can transform the spine search tree in a way that we can efficiently move the limb nodes to the bottom.

Lemma 12.25. *Let G be a connected graph and H be a spine of G. Let S be a search tree on H and let T be a search tree on G such that S is a suffix of T. Then, we can transform T into a tree T' such that S is a prefix of T', with at most* $\operatorname{cost}(S, w_{G/H})$ *rotations.*

To minimize $\operatorname{cost}(S, w_{G/H})$, we choose S to be an *optimal* static search tree on $(H, w_{G/H})$. Finally, since S induces a prefix of both trees, we can use lemma 11.23 to bound their distance. We now condense the outline above into a formal proof.

Proof of theorem 12.23. Let T, T' be search trees on G. Apply lemma 12.24 to both to obtain trees T_1, T_1'. This requires $6n$ rotations in total. Now $V(G-H)$ induces a prefix of both T_1 and T_1', so $V(H)$ induces a suffix (specifically, a rooted subtree, since H is connected). Let S be an optimal search tree on the weighted path $(H, w_{G/H})$. We now apply lemma 11.22 to make S a suffix of T_1 and T_1', obtaining T_2, T_2' with $2\operatorname{diam}(\mathcal{R}(H)) \leq 4n$ rotations (recall that the diameter of the path associahedron is less than $2n$ by theorem 11.3). Now we apply lemma 12.25 to T_2, T_2' and obtain T_3, T_3'. This needs at most $2\operatorname{cost}(Sw_{G/H}) = 2\operatorname{StOPT}(H, w_{G/H})$ rotations in total, and S is a prefix of both T_3 and T_3'. Finally, the rotation distance between T_3 and T_3' is at most

$$\sum_{C \in \mathbb{C}(G-H)} \operatorname{diam}(\mathcal{R}(C))$$

by lemma 11.23. Adding up all rotation counts yields the desired bound. □

It remains to prove lemmas 12.24 and 12.25.

12.4.1. Moving limb nodes to the top

In this section, we show lemma 12.24. The following lemma on *BSTs* will be useful.

Lemma 12.26. *Let T be a binary search tree. There exists a sequence of $3n$ rotations on T such that*

(i) every rotation involves only nodes at depth at most 3; and

(ii) every node becomes the root of T at some point.

A version of lemma 12.26 with a worse constant can be easily derived from a result by Cleary [Cle02]. That proof uses algebraic techniques; we provide a simple elementary proof instead.

Proof of lemma 12.26. We proceed in two phases. In the first phase, we repeatedly rotate the root of T with its left child, until the root has no left child. This requires at most $n-1$ rotations.

In the second phase, we repeat the following step as long as the root has a right child u. If u has a left child v, then rotate v with u. Otherwise, rotate u with the root. See figure 12.9 for an example.

We now bound the number of rotations in the second phase. Let $L(T)$ be the *left path* of T, i.e., the maximal sequence $v_1, v_2, \dots$ where v_1 is the root and v_{i+1} is the left child of v_i. Let the *right path* $R(T)$ be defined similarly. Let $f(T) = 2|L(T)| + |R(T)|$. Observe that each rotation in the second phase increases $f(T)$ by one. Since $3 \leq f(T) \leq 2n+1$ for every binary search tree T, the total number of rotations is at most $2n-2$. Both phases together take $3n-3 \leq 3n$ rotations, as desired.

The rotations only involve the root, its children, and its grandchildren, so (i) holds. To show (ii), suppose a node v never becomes the root. At the start of the second phase, v is in the right subtree of the root, and at the end, v is in the left subtree of the root. Thus, v must pass from the right subtree to the left subtree of the root at some point. The only way this can happen without v becoming the root is that v is in the left subtree of the right child u of the root, and we rotate at u. But in the second phase we only rotate at u when u has no left child, a contradiction. Thus, (ii) holds. □

light edge

In the following, fix a tree G and a spine H in G that is a path. Let T be a search tree on G and let $S = T|_H$. An edge from child c to parent p in S is called *light* if p is also a parent of c in T. (Note that in any case p must be an *ancestor* of c in T, by lemma 2.18.) Recall that a search tree on H is called a *spine search tree*. The following observation is immediate from the definitions.

Observation 12.27. *A rotation in a spine search tree $S = T|_H$ can be applied to T if and only if the rotated edge is light.*

The proof of lemma 12.24 is algorithmic. The idea is to apply lemma 12.26 to the spine search tree $T|_H$. Of course, the rotation sequence from lemma 12.26 cannot be directly applied to T; we need to make sure that the relevant edge is light before applying a rotation. For this, part (i) of lemma 12.26 is useful. Since we only rotate at depth at most three in the spine search tree, we first do a "cleanup" step that rotates each limb

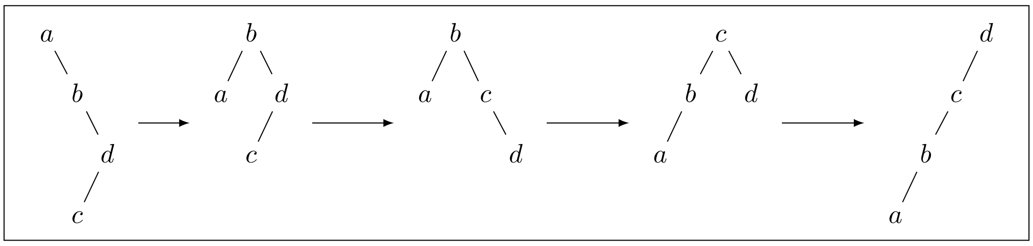


Figure 12.9.: The second phase in the proof of lemma 12.26.

node with one or two spine ancestors upwards, until it has no spine ancestors anymore. This will make the rotated edge light, and, conveniently, brings all limb nodes to the top of the tree in the end, as desired. With a little care, we can make sure that each limb node is only rotated twice, ensuring that the number of rotations is linear overall. We now proceed with the proof, starting with the “cleanup” step.

Let T be a search tree on G. Let `cleanup`(T) denote the following sequence of rotations: As long as there is a limb node v with a spine parent p and at most two spine ancestors in total, rotate ℓ with p. Arbitrarily resolve conflicts.

We now show a few basic facts about the `cleanup` procedure.

Lemma 12.28. *Let T' be the search tree obtained from applying* `cleanup`(T) *to T.*

(i) In T', each limb node has either zero or at least three spine ancestors.

(ii) Each limb node with at most two spine ancestors in T has no spine ancestors in T'.

(iii) Each limb node touched by `cleanup`(T) *has no spine ancestors in T'.*

(iv) Each limb node touched at most twice by `cleanup`(T)*.*

Proof. Let ℓ be a limb node.

(i) If ℓ has one or two spine ancestors, then it has a limb ancestor ℓ' that is the child of one of these two spine nodes. But then `cleanup`(T) would not stop before rotating ℓ'.

(ii) All rotations in `cleanup`(T) are of a limb child with a spine parent. No node can ever *gain* a spine ancestor with such a rotation. Thus, by (i), the limb node has no spine ancestors in T'.

(iii) When first being touched by `cleanup`(T), the limb node has at most two spine ancestors, so (ii) applies.

(iv) Whenever a limb node is rotated, it loses a spine ancestor. Directly before the rotation, it has at most two spine ancestors, and (as observed above), it cannot ever gain a spine ancestor. Thus, it cannot be involved in more than two rotations. □

We are now ready to prove lemma 12.24.

Lemma 12.24. *Let G be a connected graph with n vertices and let H be a spine of G that is a path. Let T be a search tree on G. Then, with at most $3n$ rotations, we can transform T into a search tree T' such that $V(G-H)$ induces a prefix of T'.*

Proof. Let R be the sequence of rotations obtained by applying lemma 12.26 to $T|_H$, called the *spine rotations*. We transform T as follows. Start with a `cleanup` step, then apply R with a `cleanup` step inserted after each spine rotation.

Each rotation in R is between two spine nodes of depth at most three. By lemma 12.28 (i), the relevant edge must be light after the preceding `cleanup` step. Thus, the sequence of rotations can indeed be applied to T.

We next bound the number of rotations. We have $|R| \leq 3|V(H)|$ spine rotations by lemma 12.26. All other rotations involve limb nodes. A single `cleanup` step touches each limb node at most twice by lemma 12.28 (iv). When a limb node is touched in a `cleanup` step, it loses all spine ancestors in that cleanup step by lemma 12.28 (iii). It cannot gain new spine ancestors in any spine rotation, nor in any later `cleanup` step (by lemma 12.28 (ii)). Thus, each limb node is touched at most twice overall, and the number of rotations is at most $2|V(G) \setminus V(H)| + 3|V(H)| \leq 3n$.

Finally, we show that $V(G-H)$ indeed induces a prefix of the final tree T'. The idea is that each limb node is *transported* near the spine root by the spine rotations, and a `cleanup` step then moves it to the top. The formal argument follows.

Let ℓ be a limb node, let $C \in \mathbb{C}(G-H)$ be the limb that contains ℓ, and let $s \in V(H)$ be the joint of C. Recall that at some point, we will rotate s to the root of the spine search tree. Let T'' be the first search tree where s is the root of the spine search tree. We show that ℓ either has no spine ancestors in T'', or it has no spine ancestors after the following `cleanup`(T'') step.

- Suppose s is an ancestor of ℓ in T''. Let T_c be the child subtree of s that contains ℓ. Since s separates $V(C)$ from the rest of the graph, and $G[T_c]$ is connected, we have $V(T_c) \subseteq V(C)$, so T_c consists entirely of limb nodes. This means that s is the nearest spine ancestor of ℓ. Since s itself has no spine ancestors, we know that ℓ has only one spine ancestor, and the next `cleanup` step will remove that spine ancestor by lemma 12.28 (ii).

- Suppose s is a descendant of ℓ in T''. Then ℓ has no spine ancestor in T'', since s is the root of the spine search tree $T''|_H$.

- Suppose s is neither an ancestor nor a descendant of ℓ in T''. We argue that then ℓ again has no spine ancestors. Suppose ℓ has a spine ancestor s'. Since s has no spine ancestor, the rooted subtree $T''_{s'}$ does not contain s. But since s separates ℓ from s' by definition, the subgraph $G[T''_{s'}]$ must be disconnected, contradicting the characterization of search trees (lemma 2.3).

We have shown that ℓ has no spine ancestors at some point, and recall it cannot gain any new ones after that. This concludes the proof. □

Remark. The reason that the proof of lemma 12.24 does not generalize to more general spines is that lemma 12.26 is specific to paths. Indeed, take the star G on n vertices and a search tree T on G where the center c of G is the unique leaf. Observe that the only way to decrease the depth of c in T is to rotate at c – which obviously is a rotation involving a node at depth n and thus egregiously violates condition (i) of lemma 12.26. Thus, lemma 12.26 does not even generalize to arbitrary trees.

12.4.2. Moving limb nodes down again

We now show lemma 12.25, starting with some technical lemmas. Essentially, we need a generalization of the observations about search trees on *caterpillars* at the start of this chapter (pages 175 to 177). Fix a connected graph G and a spine H of G.

Lemma 12.29. *Let T be a search tree on G. Let ℓ be a limb vertex and v be the joint of ℓ. If T_ℓ contains a spine node, then it must contain v.*

Proof. Suppose T_ℓ contains some spine node s, but not v. Since v separates ℓ from s by definition, we have that $G[T_\ell]$ is not connected, contradicting lemma 2.3. □

Let ℓ be a limb node in a search tree T on G. Define the rotation sequence $\texttt{serve}(T, \ell)$ as follows. As long as ℓ has a spine child s, rotate s with ℓ. Observe that $\texttt{serve}(T, \ell)$ corresponds to serving an *access* to v, as described in the beginning of this chapter.

$\texttt{serve}(T, \ell)$

Lemma 12.30. *Let ℓ be a limb node in a search tree T on G, and let v be the joint of ℓ. Then $\texttt{serve}(T, \ell)$ consists of at most $\operatorname{depth}_{T|_H}(v)$ rotations.*

Proof. By lemma 12.29, as long as ℓ has any spine descendant, its joint v is a descendant. Consider the number of spine nodes that are descendants of ℓ and ancestors of v (including v). This number is clearly bounded by $\operatorname{depth}_{T|_H}(v)$, and it is decreased by each rotation in $\texttt{serve}(T, \ell)$. □

Lemma 12.31. *Let T be search tree on G such that each limb node in T has either no spine ancestors, or no spine descendants. Let ℓ be a limb node with a spine child, and let T' be obtained by applying $\texttt{serve}(T, \ell)$ to T. Then ℓ has no spine descendants in T', and each other limb node has either no spine ancestors or no spine descendants in T'.*

Proof. Consider a limb node $\ell' \neq \ell$. We show that ℓ' has either no spine ancestors of no spine descendants in T' (see lemma 2.10).

Assume first that ℓ' has no spine descendants in T. Since ℓ' is not involved in any rotation, clearly ℓ' does not gain spine descendants at any point during the process, so it has no spine descendants in T'.

Now assume ℓ' does have some spine descendant s', so it does not have any spine ancestors. We claim that ℓ' cannot be a descendant of ℓ in T. It is easy to see that then ℓ' cannot gain any ancestors via $\texttt{serve}(T, \ell)$, and in particular, it still has no spine ancestors in T'.

Towards our claim, let s be a spine child of ℓ in T (which exists by assumption). Suppose ℓ' is a descendant of ℓ. Since ℓ' cannot be a descendant of s, the two rooted subtrees T_s and $T_{\ell'}$ are disjoint and their vertex sets are separated by ℓ in $G[T_\ell]$ by observation 2.5. Since $T_{\ell'}$ contains the spine node s', in particular s and s' are separated by ℓ in $G[T_\ell]$. This means that some path between s and s' in G goes through ℓ, which is impossible, since the limb containing ℓ is 1-cut by definition.

Now consider ℓ. After applying the process, its former spine child s must be an ancestor of ℓ. Moreover, all children of ℓ are limb nodes. Observe that s is also an ancestor of each of these children, implying that they cannot have spine descendants by the observations above. Hence ℓ also has no spine descendants. □

We are now ready to show lemma 12.25.

Lemma 12.25. *Let G be a connected graph and H be a spine of G. Let S be a search tree on H and let T be a search tree on G such that S is a suffix of T. Then, we can transform T into a tree T' such that S is a prefix of T', with at most* $\operatorname{cost}(S, w_{G/H})$ *rotations.*

Proof. We transform T into T' with the following process. As long as there is some limb node ℓ with a spine child, we apply $\texttt{serve}(\,\cdot\,, \ell)$. By lemma 12.31, after each $\texttt{serve}(\,\cdot\,, \ell)$, each limb node has no spine ancestors or no spine descendants, and ℓ in particular has no spine descendants. Moreover observe that no limb node ever *gains* a spine descendant, since all rotations only move spine nodes "upwards". Thus, $\texttt{serve}(\,\cdot\,, \ell)$ is called only once for each limb node ℓ, and eventually, all limb nodes have no spine descendants.

To bound the number of rotations, observe first that $\texttt{serve}(\,\cdot\,, \ell)$ does not perform rotations between spine nodes, so we have $S = T''|_H$ for each intermediate tree T'' (by lemma 2.20).

Further recall that by lemma 12.30, each $\texttt{serve}(T'', \ell)$ needs $\operatorname{depth}_{T''|_H}(v) = \operatorname{depth}_S(v)$ rotations, where v is the joint of ℓ. Since we apply $\texttt{serve}$ to each limb node at most once, this adds up to at most $\operatorname{cost}(S, w_{G/H})$. □

Having showed lemmas 12.24 and 12.25, this concludes the proof of theorem 12.23.

12.5. Diameter bounds for trees with bounded path-width

In this section, we show:

Theorem 11.9. *Given a tree G on n vertices with path-width k, we can approximate* $\operatorname{diam}(\mathcal{R}(G))$ *up to a factor of four and an additive term of $\mathcal{O}(kn)$, in time $\mathcal{O}(kn^2)$.*

Let G be a tree, and consider the following process. In each round, choose a path in each remaining component of G, and delete those paths. Suppose we need k rounds to delete the whole graphs this way. This process mimics the recursive application of theorem 12.1.

Observe that each application introduces two kinds of errors. First, we have a *multiplicative* error of four, since the constant factor of the term $\operatorname{StOPT}(\dots)$ is $\frac{1}{2}$ in the lower bound and two in the upper bound. On the other hand, we have the *additive* error of $11.5n$.

The additive error accumulates over each of the k rounds. Thus, if k is large, the additive error may dominate the bounds, rendering them useless. This is in particular the case if $k \in \Omega(\log n)$, since the diameter of $\mathcal{R}(G)$ is always $\mathcal{O}(n \log n)$. On the other hand, if k is constant, then we always get a constant-factor approximation of $\operatorname{diam}(\mathcal{R}(G))$, since $\operatorname{diam}(\mathcal{R}(G)) \in \Omega(n)$.

It turns out this value k is approximately equal to the *path-width* of a tree. Trees with constant path-width include many well-known classes like caterpillars, spiders, and lobsters. On the other hand, the path-width of a tree can be logarithmic (e.g., for binary trees), so our technique does not give non-trivial bounds for all trees.

In this section, we formalize this idea to prove theorem 11.9.

12.5.1. Path-deletion trees

In this section, we define an analog to STGs based on repeated removal of *paths* instead of vertices. A *path-deletion tree (PDT)* on a tree G is constructed as follows. First, choose a path P and assign it to the root node r of the PDT. Then, recursively construct PDTs on each component of $G - H$ and make them child subtrees of r.

path-deletion tree (PDT)

More formally, a path-deletion tree on a tree G is a tuple (T, P), where T is a rooted tree (with $V(T)$ disjoint from $V(G)$), and P is a function mapping the vertices of T to paths within G. As usual, we allow the domain of P to be a superset of $V(T)$ whenever convenient. For each node $v \in V(T)$, define $U(v) = \bigcup_{u \in V(T_v)} V(P(u))$ to be the set of vertices appearing in paths assigned to some node in T_v. The following conditions must hold if (T, P) is a path-deletion tree.

(i) Each vertex of G occurs in some path, i.e., we have $U(\operatorname{root}(T)) = V(G)$.

(ii) The root r of T has precisely one child c for each component C of $G - P(r)$, so that (T_c, P) is a PDT on C.

Observe that condition (ii) implies that for each pair of distinct nodes $u, v \in V(T)$, the two paths $P(u)$ and $P(v)$ are vertex-disjoint. Moreover, each $U(v)$ induces a connected subgraph of G.

Given a PDT on G, we can compute upper and lower bounds on G, using theorem 12.1.

Lemma 12.32. *Let G be a tree on n vertices and let (T, P) be a PDT on G. Then*

$$\operatorname{diam}(\mathcal{R}(G)) \geq \sum_{v \in V(T)} \tfrac{1}{2} \operatorname{StOPT}\left(P(v), w_{G[U(v)]/P(v)}\right) - \tfrac{3}{2}|U(v)|, \text{ and}$$
$$\operatorname{diam}(\mathcal{R}(G)) \leq \sum_{v \in V(T)} 2 \operatorname{StOPT}\left(P(v), w_{G[U(v)]/P(v)}\right) + 10|U(v)|.$$

Proof. Suppose $|V(T)| = 1$ and let r be the unique node in T. Recall that $0 \leq \operatorname{diam}(\mathcal{R}(P(r))) \leq 2n$. Since $w_{G/P(v)}$ is the all-zero weight function, we further have $\operatorname{StOPT}(P(v), w_{G/P(v)}) = 0$. Thus 0 and $2n$ are both within the required bounds.

Now suppose $|V(T)| > 1$. Let r be the root of T, and let K be the set of its children. By theorem 12.1 and induction, we have

$$\begin{aligned}\operatorname{diam}(\mathcal{R}(G)) &\leq 2 \operatorname{StOPT}\left(P(r), w_{G/P(r)}\right) + 10n + \sum_{C \in \mathbb{C}(G - P(r)} \operatorname{diam}(\mathcal{R}(C)) \\ &\leq 2 \operatorname{StOPT}\left(P(r), w_{G/P(r)}\right) + 10n + \sum_{c \in K} \sum_{v \in V(T_c)} 2 \operatorname{StOPT}\left(P(v), w_{G[U(v)]/P(v)}\right) + 10|U(v)|,\end{aligned}$$

which equals the desired upper bound. The lower bound is proved in the same way. □

We can easily bound the additive error in lemma 12.32 using the height of the PDT, which yields:

Lemma 12.33. *Given a PDT (T, P) of height k on a tree G with n vertices, in time $\mathcal{O}(kn^2)$, we can compute a number d such that $d \leq \operatorname{diam}(\mathcal{R}(G)) \leq 4d + 16kn$.*

Proof. First, observe that $\sum_{v \in V(T)} |U(v)| \leq \operatorname{height}(T) \cdot n = kn$. Given a weighted path (P, w), we can compute $\operatorname{StOPT}(P, w)$ in quadratic time [Knu71]. The quantity $d' = \sum_{v \in V(T)} \operatorname{StOPT}\left(P(r), w_{G/P(r)}\right)$ can thus be computed in time $\mathcal{O}(kn^2)$. Moreover, by lemma 12.32, we have

$$\tfrac{1}{2}d' - \tfrac{3}{2}kn \leq \operatorname{diam}(\mathcal{R}(G)) \leq 2d' + 10kn.$$

Setting $d = \frac{1}{2}d' - \frac{3}{2}kn$ thus satisfies the required bounds. □

What remains to discuss now is the connection to path-width and how to compute PDTs with low height.

12.5.2. Strahler number

Strahler number

Let S be a rooted tree. The *Strahler number* $\operatorname{sn}(S)$ of S is defined as follows. Let $\mathcal{P}$ be the set of root paths in S. Then $\operatorname{sn}(S) = 1$ if S is degenerate, and otherwise

$$\operatorname{sn}(S) = 1 + \min_{P \in \mathcal{P}(G)} \left(\max_{C \in \mathbb{C}(S-P)} \operatorname{sn}(C) \right).$$

The Strahler number was originally developed to measure complexity of river systems [Hor45, Str57], but has since been used in many different contexts; we refer to Esparza, Luttenberger, and Schlund [ELS14] for more information.

The following characterization of the Strahler number is easy to prove.

Observation 12.34. *Let S be a rooted tree. Then $\operatorname{sn}(S)$ is the maximum k such that S contains a subdivision of a perfect binary tree of height k.*

The Strahler number is related to the path-width (see section 2.4) as follows:[2]

Lemma 12.35. *Let S be a rooting of a tree G. Then*

$$\operatorname{pw}(G) \leq \operatorname{sn}(S) \leq 2\operatorname{pw}(G) + 1.$$

Proof sketch. The lower bound is easy to show via the equivalence of path-width and the so-called *vertex separation number* [Kin92].

For the upper bound, suppose $\operatorname{sn}(S) \geq k$. Observation 12.34 implies that S contains a subdivision S' of a perfect binary tree of height k. Let G' be the unrooting of S'. It is known that $\operatorname{pw}(G') \geq \lceil \frac{k-1}{2} \rceil$ [Sch89a, Bod98], implying that $2\operatorname{pw}(G) + 1 \geq 2\operatorname{pw}(G') + 1 \geq k$, as desired. □

For us, the main use of the Strahler number is that it implies the existence of PDTs with low height. Formally, let S be a rooting of a tree G. There always exists a PDT on G with height $\operatorname{sn}(S)$: Simply take the path $P \in \mathcal{P}(G)$ that minimizes the maximum Strahler number of the components of $S - P$.

Further, the Strahler number (and an associated PDT) is easy to compute in linear time. Given a rooted tree S, compute the Strahler number of each child subtree of the root. Let k be the maximum Strahler number of child subtrees. If a unique subtree S_c attains

[2]A slightly weaker version of lemma 12.35 was observed by Esparza et al. [ELS14].

$\mathrm{sn}(S_c) = k$, then $\mathrm{sn}(S) = k$. Otherwise, we have $\mathrm{sn}(S) = k + 1$. It is easy to compute a PDT with height $\mathrm{sn}(S)$ in a similar recursive way, in linear time.

We can combine these observations with lemmas 12.33 and 12.35: Given a tree G, take an arbitrary rooting S of G, compute a PDT on G with height $\mathrm{sn}(G) \leq 2\,\mathrm{pw}(G) + 2$, and apply lemma 12.33. This implies:

Theorem 11.9. *Given a tree G on n vertices with path-width k, we can approximate $\mathrm{diam}(\mathcal{R}(G))$ up to a factor of four and an additive term of $\mathcal{O}(kn)$, in time $\mathcal{O}(kn^2)$.*

Remark. The astute reader may have noticed that while the Strahler number gives us *some* PDT, it may not necessarily give us the *best* PDT. In particular, even when G is a path, an unlucky rooting of G can have Strahler number two. One could define the *unrooted Strahler number* $\mathrm{usn}(G)$ like the Strahler number, except that the underlying graph G is unrooted and we can select an *arbitrary* path P to delete from G, instead of only a root path. Clearly, the minimum height of a PDT on G is precisely $\mathrm{usn}(G)$.

unrooted Strahler number

However, it is easy to see that $\mathrm{usn}(\cdot)$ and $\mathrm{sn}(\cdot)$ are asymptotically equal. Let S be a rooting of a tree G. Clearly, we have $\mathrm{usn}(G) \leq \mathrm{sn}(S)$. On the other hand, since each path in G can be covered by at most two root paths in S, we also have $\mathrm{sn}(S) \leq 2\,\mathrm{usn}(G)$. Thus, the PDTs used in our approach above are nearly-optimal, and there is no significant advantage in trying to compute $\mathrm{usn}(G)$ and proper minimum-height PDTs.

12.6. The interleave lower bound for dynamic search trees

In this section, we show how our generalization of the interleave lower bound applies to dynamic STGs. Recall that this was shown before for the special case when the underlying graph is a tree [BCI+20].

Theorem 12.36. *Let G be a connected graph, and let $\sigma \in V(G)^m$ be a sequence of nodes in G. Let $\mathcal{S}$ be the set of search trees on G. Then*

$$\mathrm{DynOPT}(G, \sigma) \geq \max_{S \in \mathcal{S}} \tfrac{1}{2}\Lambda(S, \sigma) - 2\,\mathrm{diam}(\mathcal{R}(G)).$$

Proof. Consider an optimal serve sequence Q of σ in the dynamic search tree model. Let S^0 be the initial search tree and S^* be the final search tree. By definition, the cost of the serve sequence is $\mathrm{DynOPT}(G, \sigma)$.

We make a few changes to the dynamic search tree model to suit our needs. Let each serve consists of three phases. In the first phase, arbitrary rotations can be executed, at unit cost per rotation. In the second phase, the served node v is *accessed*, which simply adds a cost of $\mathrm{depth}_S(v)$, where S is the current search tree. In the third phase, again arbitrary rotations can be executed.

It is easy to see that each serve sequence can be transferred into the new model, with no increase in cost. Let Q' be the equivalent of Q in the new model. Formally, we let Q' be a sequence of triples (R_1, x, R_2), where R_1, R_2 are the sequence of rotations in the first, resp. second phase, and x is the accessed node.

We now define two search trees T and T' on a different graph H. We then first lower bound $d(T, T')$ by $\max_{S \in \mathcal{S}} \frac{1}{2}\Lambda(S, \sigma)$, and then upper bound $d(T, T')$ by $\mathrm{DynOPT}(G, \sigma) + 2\,\mathrm{diam}(\mathcal{R}(G))$. This implies the theorem.

First of all, we define H. Write $\sigma = s_1, s_2, \dots, s_m$ and let $\sigma^{\mathrm{R}} = s_m, s_{m-1}, \dots, s_1$ be the reverse of σ. Define a graph H as follows. For each element s_i of σ, attach a leaf ℓ_i to the vertex s_i (vertices may end up with no or multiple leaves). Observe that G is a spine of H.

Recall the definitions of $A(\cdot,\cdot,\cdot)$ and $B(\cdot,\cdot)$ from section 12.2.4. Let S be a search tree on G that maximizes $\Lambda(S,\sigma)$, and let $T = A(S, \sigma^{\mathrm{R}}, (T^i)_{i\in[m]})$ and $T' = B(S, (T^i)_{i\in[m]})$, where T^i is the search tree consisting only of the node ℓ_i. By lemma 12.9, we have

$$d(T,T') \geq \tfrac{1}{2}\Lambda(S,\sigma^{\mathrm{R}}) = \tfrac{1}{2}\Lambda(S,\sigma).$$

We now describe a sequence of $\mathrm{DynOPT}(G,\sigma) + 2\,\mathrm{diam}(\mathcal{R}(G))$ rotations that transform T into T', thus providing the upper bound on $d(T,T')$ and finishing the proof. We start by transforming the spine search tree S of T into S^0, using no more than $\mathrm{diam}(\mathcal{R}(G))$ rotations. This is possible since S is a suffix of T.

Now, for the i-th serve (R_1, s_i, R_2) in Q', we do the following. Let T^1 be the current search tree. First, apply the rotations in R_1 to $T^1|_H$, obtaining a search tree T^2. Then, use the `serve`(T^2, ℓ_i) operation defined in section 12.4.2. This requires $\mathrm{depth}_{T^2}(s_i)$ rotations by lemma 12.30. Let T^3 be the resulting search tree. Then, apply R_2 to $T^3|_H$.

Finally, we transform the final spine search tree S^* into S, using at most $\mathrm{diam}(\mathcal{R}(G))$ rotations. Clearly, the total number of rotations is at most $2\,\mathrm{diam}(\mathcal{R}(G))$ plus the cost of Q', as desired.

We now argue that the sequence of rotation is valid. Observe that for every intermediate search tree T'' between two calls of `serve`, each edge of the spine search tree is *light* in the sense defined in section 12.4.1. Thus, all non-`serve` rotations are valid.

For the rotations in `serve`, recall that the i-th call to `serve` is `serve`(T'', ℓ_i), for some current search tree T''. This means that the limb nodes at the top of the tree are transported to the bottom one-by-one, as outlined in section 12.4.2. (Recall that the order of the limb nodes at the top of T, directed away from the root of the spine search tree, is $\ell_1, \ell_2, \dots, \ell_m$.) Thus, we indeed transform T into T' with at most $\mathrm{DynOPT}(G,\sigma) + 2\,\mathrm{diam}(\mathcal{R}(G))$ rotations. □

Observe that this bound may be very weak, in particular on dense graphs. For example, if G is a clique, then $\Lambda(S,\sigma)$ is always zero, but $\mathrm{DynOPT}(G,\sigma)$ can be in $\Omega(|\sigma| \cdot |V(G)|)$ (see section 8.3).

Open question 12.2. Is there a dynamic search tree lower bound that is non-trivial even for dense graphs?

Back in the tree case, recall that Wilber [Wil89] also found a different lower bound, usually called the *funnel bound*, which is known to be stronger than the interleave bound [LW20].

Open question 12.3. Does some generalization of the *funnel* lower bound hold for STTs?

12.6.1. The static finger bound for general STTs

Lemma 12.15, together with theorem 12.36, has the following interesting consequence for the relation between the static and dynamic search tree models.

Proposition 12.37. *Let (G, w) be a weighted tree on n vertices. Then there exists a sequence $\sigma \in V(G)^m$ where each $v \in V(G)$ occurs precisely $w(v)$ times, such that*

$$\mathrm{DynOPT}(G, \sigma) \geq \tfrac{1}{2}\,\mathrm{StOPT}(G, w) - \tfrac{1}{2}n - 2\,\mathrm{diam}(\mathcal{R}(G)).$$

In, particular, we can show the following, which implies that the *static finger bound* (see section 8.1) cannot be achieved for STTs.

Proposition 12.38. *Let G be the unrooting of a complete binary tree on $n \geq 10$ vertices. Then, for each $m \geq 9n$, there exists a sequence $\sigma \in V(G)^m$ such that* $\mathrm{DynOPT}(G, \sigma) \in \Omega(m \log n)$.

Proof. We first bound $\mathrm{StOPT}(G, \mathbb{1})$. Since the maximum degree of G is three, every node in a search tree T on G has degree at most three. Hence, the number of nodes at depth at most k is at most $\sum_{i=1}^{k} 3^k \leq 3^{k+1}$. If $k = \lfloor \log_3(n) - 2 \rfloor$, at least $n - 3^{k+1} \geq \frac{2}{3}n$ vertices have depth at least $k+1$. This implies that $\mathrm{StOPT}(G, \mathbb{1}) \geq \frac{2}{3}n(\log_3(n) - 2)$.

Now let $\ell = \lfloor m/n \rfloor$, and let σ the sequence obtained with proposition 12.37 for the weight function $\ell \cdot \mathbb{1}$. Observe that $m - n < |\sigma| \leq m$. Using theorem 11.5 to bound $\mathrm{diam}(\mathcal{R}(G))$, we have

$$\begin{aligned}\mathrm{DynOPT}(G, \sigma) &\geq \tfrac{1}{2}\,\mathrm{StOPT}(G, \ell \cdot \mathbb{1}) - \tfrac{1}{2}n - 2\,\mathrm{diam}(\mathcal{R}(G)) \\ &\geq \tfrac{1}{2}(\ell - 8) \cdot \mathrm{StOPT}(G, \mathbb{1}) \\ &\geq \tfrac{1}{3}(\ell - 8) \cdot n(\log_3(n) - 2).\end{aligned}$$

By assumption, we have $\ell \geq 9$ and $\log_3(n) > 2$, so the required bound holds. With padding, we can easily extend σ to have length exactly m. □

As we discussed in section 8.1, proposition 12.38 shows that the static finger bound cannot be achieved for general trees. Indeed, in a binary tree with n vertices, the distance between any two vertices is $\mathcal{O}(\log n)$. This means that the static finger bound is $\mathcal{O}(m \log\log n)$ for sequences of length m, which is obviously incompatible with the $\Omega(m \log n)$ lower bound we just proved.

13. Rotation distance between k-cut STTs

In this chapter, we show that the rotation distance between k-cut search trees on trees is always linear. Recall that, in contrast, the rotation distance between *general* search trees on an n-vertex tree can be $\Omega(n \log n)$, by theorem 11.11.

We proceed with the formal statement of the result. For a connected graph G and $k \geq 1$, let $\mathcal{R}_k(G)$ be the subgraph of $\mathcal{R}(G)$ induced by k-cut trees. Observe that $\mathcal{R}_k(G)$ is non-empty if $\operatorname{tw}(G) \geq k$, and empty otherwise. Our main result is the following, which immediately implies theorem 11.15 in chapter 11.

$\mathcal{R}_k(G)$

Theorem 13.1. *For each tree G with n vertices and each $k \geq 1$, we have*

$$\operatorname{diam}(\mathcal{R}_k(G)) \leq (2k-1)n - (k+1)k + 1 \leq (2k-1)n.$$

Theorem 13.1 in particular implies that the graph $\mathcal{R}_k(G)$ is connected if G is a tree. Observe, however, that $\mathcal{R}_k(G)$ is *not* necessarily the graph of a convex polytope. For example, $\mathcal{R}_1(G)$ is isomorphic to G itself, since the vertices of $\mathcal{R}_1(G)$ are rootings of G, and two rootings are adjacent in $\mathcal{R}_1(G)$ if and only if their roots are adjacent in G. A tree is clearly not the graph of any convex polytope, if the tree has more than two vertices.

Tightness. Theorem 13.1 is not asymptotically tight for all k and n; already for $k \in \omega(\log n)$, the bound is strictly above the general $\mathcal{O}(n \log n)$ bound for STT rotation distance [CLPL18]. It is possible that the true bound is $\mathcal{O}(n \log k)$. Or, perhaps, our bound is correct for $k \in \mathcal{O}(\log n)$. This would mean that there are $\mathcal{O}(\log n)$-cut search trees on trees with maximal rotation distance, which would be interesting in itself.

Open question 13.1. For every n, is there a tree G on n vertices that admits two $\mathcal{O}(\log n)$-cut search trees with rotation distance $\Omega(n \log n)$?

We remark that the $\Omega(n \log n)$ lower bound of Cardinal, Langerman, and Pérez-Lantero [CLPL18] for the maximum diameter of tree associahedra heavily relies on using $\Omega(n)$-cut search trees, and the same is true for our caterpillar lower bound (theorem 11.10).

Another open question is whether the leading constant $(2k-1)$ in theorem 13.1 is tight for small k. Observe that if G is a path, then $\mathcal{R}_2(G) = \mathcal{R}(G)$ (every binary search tree is 2-cut), so $\operatorname{diam}(\mathcal{R}_2(G)) \leq 2n$, which is stronger than the $3n-3$ bound we get from theorem 13.1.

Open question 13.2. Is $\operatorname{diam}(\mathcal{R}_2(G)) \leq 2n$ for every tree G?

While we cannot rule out that $\operatorname{diam}(\mathcal{R}_2(G)) \leq 2n$ holds for all trees, we can show that a certain strategy to prove it cannot work. The $2n$ bound for paths is easily proved by showing that there exists a "canonical" 1-cut search tree S (a rooting of the path at one of its endpoints), such that $d(S,T) \leq n$ for all search trees S'. We show that such a search tree S cannot exist in general.

Proposition 13.2. *For each $k \in \mathbb{N}_+$, there is a tree G on $n = 3k+1$ vertices such that for each 1-cut search tree S on G, there is a 2-cut search tree T on G with $d(S,T) \geq \frac{4}{3}n - 6$.*

Of course, proposition 13.2 does not rule out a proof of $\operatorname{diam}(\mathcal{R}_2(G)) \leq \frac{8}{3}n$ with this strategy.

Finally, whether theorem 13.1 can be generalized to bounded-tree-width graphs is another interesting open question.

Open question 13.3. Is $\operatorname{diam}(\mathcal{R}_k(G)) \in \mathcal{O}(kn)$ for each connected graph G with $\operatorname{tw}(G) \leq k$?

$d_k(T,T')$

Notation. For two k-cut search trees T, T' on a graph G, let $d_k(T,T')$ denote the rotation distance of T and T' in $\mathcal{R}_k(G)$.

13.1. Upper bound

We prove theorem 13.1 by induction. For $k = 1$, we directly transform T into T'. For $k > 1$, we reduce k by transforming both T and T' into $(k-1)$-cut trees, and then proceed by induction.

The following lemma concerns the case $k = 1$. Recall that 1-cut trees are precisely *rootings* of G.

Lemma 13.3. *Let G be a tree on n vertices, let T be a rooting of G at some node x, and let T' be a rooting of G at some node x'. Then $d_1(T,T') \leq \operatorname{depth}_T(x') - 1 \leq n - 1$.*

Proof. Let c be the child of x in T such that $x' \in V(T_c)$, and let T'' be the tree obtained by rotating c with x in T. We claim that T'' is precisely the tree rooted at c, and thus $\operatorname{depth}_{T''}(x') = \operatorname{depth}_T(x') - 1$. By induction, after $\operatorname{depth}_T(x') - 1$ such rotations, we obtain T'.

Indeed, by lemma 2.10, the only nodes whose set of descendants is changed by the rotation are x and c. Since both have depth at most one in T'', both x and c are 1-cut in T'' (by observation 2.7). For all other nodes $y \in V(G) \setminus \{x, c\}$, we have $V(T''_y) = V(T_y)$, so y is 1-cut in T''. Clearly, the root of T'' is c, so T'' is a rooting of G at c, as desired. □

We continue with the transformation of k-cut STTs into $(k-1)$-cut STTs. The following lemma shows that certain rotations strictly "improve" the tree, by reducing the boundary size of some rooted subtree and not affecting the boundary sizes of other rooted subtrees.

Lemma 13.4. *Let T be a k-cut search tree on a tree S, where $k \geq 2$. Let v be a node of T with parent p such that $|\partial(T_v)| = k$ and $|\partial(T_p)| = k-1$. Then, rotating v with p produces a k-cut search tree T', where $|\partial(T'_v)| = k-1$ and for each node $x \in V(T) \setminus \{v\}$, we have $|\partial(T'_x)| \leq |\partial(T_x)|$.*

Proof. By lemma 2.10, we have $V(T'_x) = V(T_x)$ for all $x \in V(G) \setminus \{v, p\}$, which implies $|\partial(T'_x)| = |\partial(T_x)| \leq k$. Further, we have $V(T'_v) = V(T_p)$, so $|\partial(T'_v)| = |\partial(T_p)| = k-1$. It remains to show $|\partial(T'_p)| \leq k-1$.

Assume on the contrary that $|\partial(T'_p)| \geq k$. Let $B = \partial(T_p)$. By observation 2.7, and since $|\partial(T_v)| = k$ and $|\partial(T_p)| = |\partial(T'_v)| = k - 1$, we have

$$\begin{aligned}\partial(T_v) &= \partial(T_p) \cup \{p\} = B \cup \{p\}, \text{ and}\\ \partial(T'_p) &= \partial(T'_v) \cup \{v\} = B \cup \{v\}\end{aligned}$$

Let U_v, U', U_p be the (v, p)-partition of G, as defined in chapter 9. By the above observations, we know that each $b \in B$ is adjacent to $V(T_v) = U_v \cup U'$, but also to $V(T'_p) = U' \cup U_p$, and thus specifically to U'. Hence, we have $|\partial(U')| \geq |B \cup \{v, p\}| = k + 1$. But there is a child subtree T_c of v in T such that $V(T_c) = U'$, which contradicts that T is k-cut. □

It is easy to see that in a k-cut search tree that is not a $(k-1)$-cut search tree, we always have a node v that satisfies the requirements of lemma 13.4. Thus, we can repeatedly apply lemma 13.4 and finish after at most n steps. In fact, we can do slightly better:

Lemma 13.5. *Let T be a k-cut search tree on a tree G, with $k \geq 2$. Then T can be transformed into a $(k-1)$-cut search tree with at most $n - k$ rotations, so that each intermediate tree is a k-cut tree.*

Proof. Suppose T is not a $(k-1)$-cut search tree. Since $|\partial(T)| = 0$, there is a node v with parent p such that $|\partial(T_v)| = k$ and $|\partial(T_p)| < k$. Since $\partial(T_v) \subseteq \partial(T_p) \cup \{p\}$ by observation 2.7, we have $|\partial(T_p)| = k - 1$. We can thus apply lemma 13.4, increasing the number of nodes x with $|\partial(T_x)| \leq k - 1$ by at least one. Repeat this step until we have a $(k-1)$-cut search tree. In the original tree T, every node x of depth at most k already satisfies $|\partial(T_x)| \leq \mathrm{depth}(x) - 1 \leq k - 1$. As such, we can ignore these nodes and need at most $n - k$ steps. □

It is easy to see that lemma 13.5 can be implemented in the *pointer model* (see chapter 8) with a linear number of operations. We omit the formal proof. Theorem 13.1 now follows by induction as outlined above:

Theorem 13.1. *For each tree G with n vertices and each $k \geq 1$, we have*

$$\mathrm{diam}(\mathcal{R}_k(G)) \leq (2k-1)n - (k+1)k + 1 \leq (2k-1)n.$$

Proof. Let T and T' be k-cut search trees on G. We show that $d_k(T, T') \leq (2k-1)n - (k+1)k+1$. If $k = 1$, we need $n-1$ rotations by lemma 13.3, as desired. Otherwise, we transform the two k-cut search trees T, T' into $(k-1)$-cut search trees S, S', using $2(n-k)$ rotations, by lemma 13.5. By induction, we have $d_k(S, S') \leq d_{k-1}(S, S') \leq (2(k-1)-1)n - k(k-1)+1$. Therefore, we have

$$\begin{aligned}d_k(T, T') &\leq 2(n-k) + (2(k-1)-1)n - k(k-1) + 1\\ &= (2k-1)n - 2k - k(k-1) + 1\\ &= (2k-1)n - (k+1)k + 1.\end{aligned}$$ □

13.2. Lower bound

We now prove proposition 13.2, starting with the following fact about search trees on *paths*.

Lemma 13.6. *Let G be the path on $n = 2k + 1$ vertices, and let S be the rooting of G at the centroid. Then there exists a search tree T on G such that $d(S,T) \geq \frac{3}{2}n - 4$.*

Proof. We name the vertices as follows, in the order they appear on the path:

$$x_1, x_2, \dots, x_{k-1}, x_k, c, y_k, y_{k-1}, \dots, y_2, y_1.$$

Observe that c is the centroid. Let T be the degenerate search tree with topological ordering $x_1, y_1, x_2, y_2, \dots, x_k, y_k, c$.

We use the alternation number, as defined in section 12.2. Let $X = \{x_i \mid i \in [k]\}$ and $Y = \{y_i \mid i \in [k]\}$. Let $\mathcal{X} = \{X, Y, \{c\}\}$. Clearly, the subgraphs $G[X], G[Y], G[\{c\}]$ are connected and thus convex, so we can apply lemma 12.8:

$$d(S,T) \geq \tfrac{1}{2}|\mathrm{alt}(S,\mathcal{X}) - \mathrm{alt}(T,\mathcal{X})| + d(S|_X, T|_X) + d(S|_Y, T|_Y).$$

Clearly $\mathrm{alt}(S,\mathcal{X}) = 1$; the only alternation is between c and its children x_k, y_k. On the other hand, we have $\mathrm{alt}(T,\mathcal{X}) = n - 1$, since every edge is an alternation.

The two search trees $S|_X$ and $T|_X$ are both rootings of the path $G[X]$, but their roots are different ends of the path. It is easy to see that their distance is precisely $k - 1$. The same is true for $S|_Y$ and $T|_Y$. Overall, we have

$$d(S,T) \geq \tfrac{1}{2}(n-2) + 2(k-1) = \tfrac{3}{2}n - 4. \qquad \square$$

Observe that lemma 13.6 is tight up to an additive constant. Indeed, if T^* is the rooting of G at one of its ends (sometimes called the *left* or *right spine*[1] in the BST setting), then clearly $d(S,T^*) \leq k$, and it is not hard to see that $d(T,T^*) \leq n-1$ for all search trees T on G. Hence, $d(S,T) \leq \frac{3}{2}n - \frac{3}{2}$.

We are now ready to prove proposition 13.2.

Proposition 13.2. *For each $k \in \mathbb{N}_+$, there is a tree G on $n = 3k+1$ vertices such that for each 1-cut search tree S on G, there is a 2-cut search tree T on G with $d(S,T) \geq \frac{4}{3}n - 6$.*

Proof. Let G be the *spider graph* with three legs of length k. Recall that G consists of one vertex c, the *central vertex*, and three paths of length k, the *legs*. One leaf of each leg is adjacent to c. Let X, Y, Z be sets of the vertices on the three legs, respectively.

Take an arbitrary rooting (i.e., 1-cut tree) S of G. Without loss of generality, $\mathrm{root}(S) \in X \cup \{c\}$ (see figure 13.1). Write $R = V(G) \setminus X = Y \cup \{c\} \cup Z$. We construct T as follows. Let T^X be a search tree on $G[X]$ with maximum distance to $S|_X$, and let T^R be a search tree on $G[R]$ with maximum distance to $S|_R$. Let T be a search tree with prefix T^R and suffix T^X.

First observe that T indeed exists; just concatenate topological orderings of T^R and T^X to obtain a topological ordering of T. Moreover, observe that $S|_R$ is a rooting of $G[R]$ at

[1] No relation to our definition of *spines* in chapter 12.

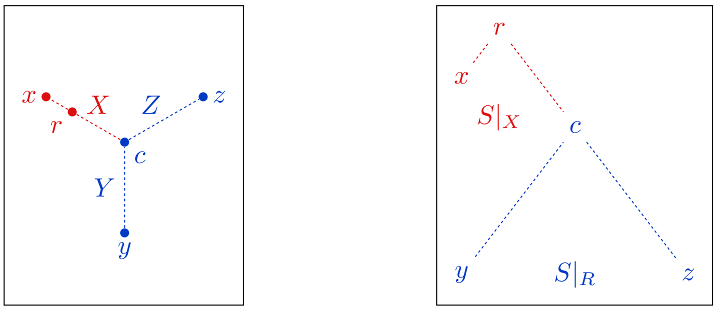


Figure 13.1.: The spider graph G (left) and its rooting S (left) in proposition 13.2.

its centroid c, so $d(S|_R, T^R) \geq \frac{3}{2}|R| - 4$ by lemma 13.6 and choice of T^R. Further observe that $d(S|_X, T^X) \geq \frac{1}{2}\operatorname{diam}(\mathcal{R}(G[X])) = \frac{1}{2}(2k-6)$ by choice of T^X, since $G[X]$ is a path (theorem 11.3).

Since R and X are disjoint and induced convex subgraphs of G, the following holds by lemma 2.20.

$$\begin{aligned} d(S,T) &\geq d(S|_R, T|_R) + d(S|_X, T|_X) \\ &= d(S|_R, T^R) + d(S|_X, T^X) \\ &\geq \tfrac{3}{2}(2k+1) - 4 + k - 3 = 4k - \tfrac{11}{2} = \tfrac{4}{3}n - \tfrac{41}{6}. \end{aligned}$$

Since $d(S,T)$ is an integer, this implies the required bound.

It remains to show that T is a 2-cut search tree. Its prefix T^R clearly is, since it is a search tree on a path. Thus, all nodes $v \in R$ are 2-cut in T. On the other hand, observe that T^X must be a child subtree of c in T, since $\partial(T^X) = \{c\}$ and $G[X]$ is connected. This means that $\partial(T_x) \subseteq X \cup \{c\}$ for all $x \in X$. Since $X \cup \{c\}$ induces a path of G, we have $|\partial(T_x)| \leq 2$, as desired. $\square$

14. Conclusion

In this thesis, we studied search trees on graphs from multiple different angles. We identified (and partially overcame) some significant challenges in our attempts to generalize results from the theory of binary search trees.

The *static* search tree model generalizes from BSTs in a straight-forward way. We gave several approximation algorithms and hardness results (part I), but left one central question open: Is there a polynomial-time algorithm that computes an optimal static search tree on a given weighted tree?

The *dynamic* search tree model and questions related to it turned out to be more subtle to generalize (part II). For example, some adaptivity bounds for BSTs do not have an obvious and sensible generalization to STGs (chapter 8). Still, we were able to generalize the Splay BST algorithm to the tree case and showed two non-trivial adaptivity properties (chapter 9). Generalizing other algorithms (like Greedy [Luc88, Mun00]), and further generalizing Splay to graphs with *bounded tree-width* are interesting topics for future research.

We also gave a practical application of our dynamic search tree algorithms: A data structure for *dynamic forests*. We sketched a generalization to graphs with bounded tree-width, which requires a generalization of Splay to such graphs.

One particular tool that was useful throughout the first two parts of the thesis is the restricted class of *k-cut search trees*, which retain some useful properties of BSTs and have a strong connection to *tree-width*. Our framework of k-cut search trees has been used already in a different setting [LK23].

Finally, we studied the diameter of *search tree rotation graphs* (part III). While this is a seemingly purely combinatorial question, we were able to utilize results from the theories of both static and dynamic search trees. Conversely, we were able to prove a dynamic search tree lower bound using tools developed for bounding rotation distance. Studying these connections further could lead to new discoveries in any of the three settings.

List of conjectures and open questions

Open question 3.1. Is there an algorithm for $\mathrm{StOPT}(G, w)$ with approximation factor less than 2 that runs in time $\mathcal{O}(n \operatorname{polylog} n)$?

Open question 3.2. Can $\mathrm{StOPT}(G, w)$ be computed exactly in polynomial time when G is a tree?

Conjecture 3.3. Theorem 3.8 can be generalized to graphs with bounded tree-width.

Open question 3.4. Are there other natural graph classes that admit constant-factor approximations for the optimal static search tree problem?

Open question 3.5. Can $\mathrm{StOPT}(G, \mathbb{1})$, i.e., the minimal trivially perfect completion, be computed in linear time if G is a tree?

Open question 4.1. What is the complexity of computing a centroid tree on a tree with n vertices, when all weights are in $[1, W]$ for $W \in \omega(1)$ and $W \leq 2^{\log^{o(1)} n}$?

Open question 5.1. Can theorem 5.8 be improved or extended to cases $t \leq k \leq 3t$?

Open question 8.1. Are the pointer model and the prefix model equivalent when restricted to k-cut search trees on trees?

Conjecture 8.2. The offline optimum achieves the torso-tree-depth working-set bound.

Conjecture 8.3. For each graph G, input sequence X, and finger vertex v, there exists a static search tree T on G that achieves the tree-depth fixed-finger bound w.r.t. v.

Open question 8.4. Is there a generalization of Greedy or GreedyFuture to the STT or STG setting? Which bounds can be achieved by such a generalization?

Open question 9.1. Let G be a tree, and let $X \in V(G)^m$ be a sequence of $m \geq |V(G)|$ searches that are sampled independently from a distribution p on $V(G)$. Does MoveToRootTT serve X in expected running time $\mathcal{O}(\mathrm{StOPT}(G, p) + f(G))$, for some function f?

Open question 9.2. Does SplayTT achieve the working-set bound with additive error $\mathcal{O}(\mathrm{StOPT}(G, \mathbb{1})$?

Open question 10.1. Is there a generalization of SplayTT that maintains $\mathcal{O}(k)$-cut search trees on (static) graphs of tree-width k?

Open question 11.1. What are asymptotically tight bounds for $\mathrm{Diam}(\mathcal{G}_t^{\mathrm{tw}}, n)$ and $\mathrm{Diam}(\mathcal{G}_t^{\mathrm{pw}}, n)$?

Conjecture 11.2. For all t and large enough n, we have

$$\mathrm{Diam}(\mathcal{G}_t^{\mathrm{td}}, n) = \mathrm{diam}(\mathcal{R}(\mathrm{SPK}_{t-1,n-t+1})).$$

Conjecture 12.1. Let H be a binary tree, and let G be a subdivision of H. Let $s_e \in \mathbb{N}_0$ be the number of times the edge $e \in E(H)$ was subdivided to obtain G, and let $w(v) = \sum_{e \in E_v} s_e$ for each $v \in V(H)$, where E_v is the set of edges incident to v. Then

$$\mathrm{diam}(\mathcal{R}(G)) \in \Theta(\mathrm{StOPT}(H, w + \mathbb{1}) + |V(G)|).$$

Open question 12.2. Is there a dynamic search tree lower bound that is non-trivial even for dense graphs?

Open question 12.3. Does some generalization of the *funnel* lower bound hold for STTs?

Open question 13.1. For every n, is there a tree G on n vertices that admits two $\mathcal{O}(\log n)$-cut search trees with rotation distance $\Omega(n \log n)$?

Open question 13.2. Is $\mathrm{diam}(\mathcal{R}_2(G)) \leq 2n$ for every tree G?

Open question 13.3. Is $\mathrm{diam}(\mathcal{R}_k(G)) \in \mathcal{O}(kn)$ for each connected graph G with $\mathrm{tw}(G) \leq k$?

Index

Symbol reference

Bibliography

[AAK$^+$12] Pankaj K. Agarwal, Lars Arge, Haim Kaplan, Eyal Molad, Robert Endre Tarjan, and Ke Yi. An Optimal Dynamic Data Structure for Stabbing-Semigroup Queries. *SIAM J. Comput.*, 41:104–127, 2012. doi:10.1137/10078791X. 137

[ABH$^+$04] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick Woo. Dynamizing Static Algorithms, with Applications to Dynamic Trees and History Independence. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 531–540, USA, 2004. Society for Industrial and Applied Mathematics. doi:10.1145/990000/982871. 137

[AHLT05] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining Information in Fully Dynamic Trees with Top Trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966. 47, 49, 60, 65, 137

[AM78] Brian Allen and Ian Munro. Self-Organizing Binary Search Trees. *J. ACM*, 25(4):526–535, October 1978. doi:10.1145/322092.322094. 6, 105, 106, 113

[BCI$^+$20] Prosenjit Bose, Jean Cardinal, John Iacono, Grigorios Koumoutsos, and Stefan Langerman. Competitive Online Search Trees on Trees. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1878–1891, 2020. doi:10.1137/1.9781611975994.115. 3, 7, 18, 42, 99, 105, 153, 180, 199

[BDJ$^+$98] Hans L. Bodlaender, Jitender S. Deogun, Klaus Jansen, Ton Kloks, Dieter Kratsch, Haiko Müller, and Zsolt Tuza. Rankings of Graphs. *SIAM J. Discret. Math.*, 11(1):168–181, 1998. doi:10.1137/S0895480195282550. 5

[Ber22] Benjamin Aram Berendsohn. The Diameter of Caterpillar Associahedra. In Artur Czumaj and Qin Xin, editors, *18th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2022, June 27-29, 2022, Tórshavn, Faroe Islands*, volume 227 of *LIPIcs*, pages 14:1–14:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SWAT.2022.14. vii, 177

[Ber24] Benjamin Aram Berendsohn. Fast and simple unrooted dynamic forests. In Rezaul Chowdhury and Solon P. Pissis, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2024, Alexandria, VA, USA, January 7-8, 2024*, pages 46–58. SIAM, 2024. doi:10.1137/1.9781611977929.4. vii

[BFCK06] Michael A Bender, Martin Farach-Colton, and Bradley C Kuszmaul. Cache-oblivious string B-trees. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–242, 2006. doi:10.1145/1142351. 47

[BFN99] Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. Optimal Search in Trees. *SIAM J. Comput.*, 28(6):2090–2102, 1999. doi:10.1137/S009753979731858X. 9, 11

[BFPÖ01] Gerth Stølting Brodal, Rolf Fagerberg, Christian N. S. Pedersen, and Anna Östlin. The Complexity of Constructing Evolutionary Trees Using Experiments. *BRICS Report Series*, 8(1), January 2001. doi:10.7146/brics.v8i1.20220. 49, 50

[BGHK95] Hans L. Bodlaender, John R. Gilbert, Hjálmtyr Hafsteinsson, and Ton Kloks. Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree. *J. Algorithms*, 18(2):238–255, 1995. doi:10.1006/JAGM.1995.1009. 3, 29, 32, 79, 169

[BGKK23] Benjamin Aram Berendsohn, Ishay Golinsky, Haim Kaplan, and László Kozma. Fast Approximation of Search Trees on Trees with Centroid Trees. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*, volume 261 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2023.19. vii, 24, 48, 51

[BIP23] Djordje Baralic, Jelena S. Ivanovic, and Zoran Petric. Chromatic numbers for facet colouring of some generalized associahedra, 2023, 2311.05901. 170

[BK22] Benjamin Aram Berendsohn and László Kozma. Splay trees on trees. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1875–1900, 2022. doi:10.1137/1.9781611977073.75. vii, 18, 117

[Bod94] Hans L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In Jan van Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science*, pages 112–124, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. doi:10.1007/3-540-57899-4_45. 161

[Bod96] Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput*, 25(6):1305–1317, 1996. doi:10.1145/167088.167161. 32

[Bod98] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1):1–45, 1998. doi:10.1016/S0304-3975(97)00228-4. 79, 198

[BRSW18] Louigi Addario Berry, Bruce Reed, Alex Scott, and David R. Wood. A logarithmic bound for the chromatic number of the associahedron, 2018, 1811.08972v1. 170

[CD04] Michael Carr and Satyan L. Devadoss. Coxeter Complexes and Graph-Associahedra. *Topology and its Applications*, 153:2155–2168, 2004. doi:10.1016/j.topol.2005.08.010. 165

[CDKL04] Renato Carmo, Jair Donadelli, Yoshiharu Kohayakawa, and Eduardo Sany Laber. Searching in random partially ordered sets. *Theor. Comput. Sci.*, 321(1):41–57, 2004. doi:10.1016/J.TCS.2003.06.001. 10

[CDL24] Julien Courtiel, Paul Dorbec, and Romain Lecoq. Theoretical Analysis of Git Bisect. *Algorithmica*, 86(5):1365–1399, 2024. doi:10.1007/S00453-023-01194-0. 10

[CDV24] Agustín Caracci, Christoph Dürr, and José Verschae. Randomized Binary and Tree Search under Pressure, 2024, 2406.06468. 11

[CFPI10] G. Cattaneo, P. Faruolo, U. Ferraro Petrillo, and G.F. Italiano. Maintaining dynamic minimum spanning trees: An experimental study. *Discrete Appl. Math.*, 158(5):404–425, 2010. doi:10.1016/j.dam.2009.10.005. 137

[CGK$^+$15] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Self-adjusting binary search trees: What makes them tick? In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, volume 9294 of *Lecture Notes in Computer Science*, pages 300–312. Springer, 2015. doi:10.1007/978-3-662-48350-3_26. 114

[CGK$^+$16] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees, 2016, 1603.04892v1. 6, 102, 103, 120

[CJLM10] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. On Greedy Algorithms for Decision Trees. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part II*, volume 6507 of *Lecture Notes in Computer Science*, pages 206–217. Springer, 2010. doi:10.1007/978-3-642-17514-5_18. 9, 10, 53

[CJLM11] Ferdinando Cicalese, Tobias Jacobs, Eduardo Laber, and Marco Molinaro. On the complexity of searching in trees and partially ordered structures. *Theoretical Computer Science*, 412(50):6879–6896, 2011. doi:10.1016/j.tcs.2011.08.042. 9, 10, 40, 93

[CJLM14] Ferdinando Cicalese, Tobias Jacobs, Eduardo Laber, and Marco Molinaro. Improved approximation algorithms for the average-case tree searching problem. *Algorithmica*, 68(4):1045–1074, 2014. doi:10.1007/s00453-012-9715-6. 9, 22, 40, 53, 87, 92

[CJLV12] Ferdinando Cicalese, Tobias Jacobs, Eduardo Laber, and Caio Valentim. The binary identification problem for weighted trees. *Theoretical Computer Science*, 459:100–112, 2012. doi:10.1016/j.tcs.2012.06.023. 9

[CKL$^+$16] Ferdinando Cicalese, Balázs Keszegh, Bernard Lidický, Dömötör Pálvölgyi, and Tomáš Valla. On the tree search problem with non-uniform costs. *Theoretical Computer Science*, 647:22–32, 2016. doi:10.1016/j.tcs.2016.07.019. 9

[Cle02] S. Cleary. Restricted rotation distance between binary trees. *Inf. Process. Lett.*, 84:333–338, 2002. doi:10.1016/S0020-0190(02)00315-0. 192

[CLPL18] Jean Cardinal, Stefan Langerman, and Pablo Pérez-Lantero. On the Diameter of Tree Associahedra. *The Electronic Journal of Combinatorics*, 2018. doi:10.37236/7762. 18, 22, 101, 165, 167, 168, 180, 203

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, Cambridge, Mass., 2nd ed edition, 2001. 6, 15, 20, 100

[CMM21] Jean Cardinal, Arturo Merino, and Torsten Mütze. Combinatorial generation via permutation languages. IV. Elimination trees, 2021, 2106.16204v5. 170

[CMSS00] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting $\log n$-Block Sequences. *SIAM J. Comput.*, 30(1):1–43, 2000. doi:10.1137/S0097539797326988. 105

[Col00] Richard Cole. On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof. *SIAM J. Comput.*, 30(1):44–85, 2000. doi:10.1137/S009753979732699X. 105

[CPVP22] Jean Cardinal, Lionel Pournin, and Mario Valencia-Pabon. Diameter Estimates for Graph Associahedra. *Annals of Combinatorics*, August 2022. doi:10.1007/s00026-022-00598-z. 22, 24, 166, 167, 168, 169, 171, 180

[CPVP24] Jean Cardinal, Lionel Pournin, and Mario Valencia-Pabon. The rotation distance of brooms. *European Journal of Combinatorics*, 118:103877, 2024. doi:10.1016/j.ejc.2023.103877. 9, 170

[CS10] Sean Cleary and Katherine St. John. A Linear-Time Approximation Algorithm for Rotation Distance. *Journal of Graph Algorithms and Applications*, 14(2):385–390, 2010. doi:10.7155/jgaa.00212. 170

[CSZ15] Cesar Ceballos, Francisco Santos, and Günter M. Ziegler. Many non-equivalent realizations of the associahedron. *Comb.*, 35(5):513–551, 2015. doi:10.1007/S00493-014-2959-9. 8

[CW82] Karel Culík II and Derick Wood. A note on some tree similarity measures. *Inf. Process. Lett.*, 15(1):39–42, 1982. doi:10.1016/0020-0190(82)90083-7. 165

[Das16] Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 118–127. ACM, 2016. doi:10.1145/2897518.2897527. 9

[DBT96] Giuseppe Di Battista and Roberto Tamassia. On-Line Planarity Testing. *SIAM J. Comput.*, 25(5):956–997, 1996. doi:10.1137/S0097539794280736. 137

[Der06] Dariusz Dereniowski. Edge ranking of weighted trees. *Discrete Appl. Math.*, 154(8):1198–1209, 2006. doi:10.1016/j.dam.2005.11.005. 9, 10

[Der08] Dariusz Dereniowski. Edge ranking and searching in partial orders. *Discret. Appl. Math.*, 156(13):2493–2500, 2008. doi:10.1016/J.DAM.2008.03.007. 9, 10

[Dev09] Satyan L. Devadoss. A realization of graph associahedra. *Discrete Mathematics*, 309(1):271–276, 2009. doi:10.1016/j.disc.2007.12.092. 165

[DGPV19] Davide Della Giustina, Nicola Prezza, and Rossano Venturini. A New Linear-Time Algorithm for Centroid Decomposition. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 274–282, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-32686-9_20. 49, 50, 61

[DHI$^+$09] Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Pătrașcu. The Geometry of Binary Search Trees. In *Proceedings of the 2009 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 496–505, 2009. doi:10.1137/1.9781611973068.55. 6, 102, 105

[DHIP07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătrașcu. Dynamic Optimality—Almost. *SIAM J. Comput.*, 37(1):240–251, 2007. doi:10.1137/S0097539705447347. 6, 99, 105

[DKKM94] J. S. Deogun, T. Kloks, D. Kratsch, and H. Müller. On vertex ranking for permutation and other graphs. In Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, editors, *STACS 94*, pages 747–758, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. doi:10.1007/3-540-57785-8_187. 29

[DKUZ17] Dariusz Dereniowski, Adrian Kosowski, Przemyslaw Uznanski, and Mengchuan Zou. Approximation Strategies for Generalized Binary Search in Weighted Trees. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 84:1–84:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.84. 8

[dlTGS95] Pilar de la Torre, Raymond Greenlaw, and Alejandro A. Schäffer. Optimal edge ranking of trees in polynomial time. *Algorithmica*, 13(6):592–618, 1995. doi:10.1007/BF01189071. 9

[DN06] Dariusz Dereniowski and Adam Nadolski. Vertex rankings of chordal graphs and weighted trees. *Inf. Process. Lett.*, 98(3):96–100, 2006. doi:10.1016/j.ipl.2005.12.006. 8, 33

[EGIN97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification—a Technique for Speeding up Dynamic Graph Algorithms. *J. ACM*, 44(5):669–696, 1997. doi:10.1145/265910.265914. 137

[EIT+92] David Eppstein, Giuseppe F Italiano, Roberto Tamassia, Robert E Tarjan, Jeffery Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *Journal of Algorithms*, 13(1):33–54, 1992. doi:10.1016/0196-6774(92)90004-V. 137

[EKS16] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. Deterministic and probabilistic binary search in graphs. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 519–532. ACM, 2016. doi:10.1145/2897518.2897656. 10

[ELS14] Javier Esparza, Michael Luttenberger, and Maximilian Schlund. A Brief History of Strahler Numbers. In Adrian-Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2014. doi:10.1007/978-3-319-04921-2_1. 198

[Fer13] Paolo Ferragina. On the weak prefix-search problem. *Theoretical Computer Science*, 483:75–84, 2013. doi:10.1016/j.tcs.2012.06.011. Special Issue Combinatorial Pattern Matching 2011. 47

[FJ83] Greg N Frederickson and Donald B Johnson. Finding kth paths and p-centers by generating and searching good data structures. *Journal of Algorithms*, 4(1):61–80, 1983. doi:10.1016/0196-6774(83)90035-4. 47

[FJ86] Martin Farber and Robert E. Jamison. Convexity in Graphs and Hypergraphs. *SIAM Journal on Algebraic Discrete Methods*, 7(3):433–444, 1986. doi:10.1137/0607049. 23

[FMFPH+09] Ruy Fabila-Monroy, David Flores-Peñaloza, Clemens Huemer, Ferran Hurtado, Jorge Urrutia, and David R. Wood. On the chromatic number of some flip graphs. *Discrete Mathematics & Theoretical Computer Science*, 11(2), January 2009. doi:10.46298/dmtcs.460. 170

[Fre75] Michael L. Fredman. Two Applications of a Probabilistic Search Technique: Sorting X+Y and Building Balanced Search Trees. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, pages 240–244, New York, NY, USA, 1975. Association for Computing Machinery. doi:10.1145/800116.803774. 5, 37, 38, 49, 50

[Fre85] Greg N. Frederickson. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM J. Comput.*, 14(4):781–798, 1985. doi:10.1137/0214055. 137

[FV16] Paolo Ferragina and Rossano Venturini. Compressed Cache-Oblivious String B-Tree. *ACM Trans. Algorithms*, 12(4):52:1–52:17, 2016. doi:10.1145/2903141. 47

[GCC] GCC team. stl_tree.h – libstdc++ source code. URL `https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a01067_source.html`. Accessed 2024/09/29. 1

[GHL$^+$87] Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987. doi:10.1007/BF01840360. 47

[GHLW15] Travis Gagie, Danny Hermelin, Gad M Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015. doi:10.1007/s00453-014-9957-6. 47

[Gol71] A. J. Goldman. Optimal Center Location in Simple Network. *Transportation Science*, 5(2):212–22, 1971. doi:10.1287/trsc.5.2.212. 47, 59, 62

[Gol78] Martin Charles Golumbic. Trivially perfect graphs. *Discret. Math.*, 24(1):105–107, 1978. doi:10.1016/0012-365X(78)90178-4. 28

[GPT24] Ana Gargantini, Adrián Pastine, and Pablo Torres. On the structure and diameter of graph associahedra of graphs with a set of true twins, 2024, 2406.06317. 166

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A New Approach to the Maximum-Flow Problem. *J. ACM*, 35(4):921–940, October 1988. doi:10.1145/48014.61051. 137

[GT98] Michael T. Goodrich and Roberto Tamassia. Dynamic Trees and Dynamic Point Location. *SIAM J. Comput.*, 28(2):612–636, 1998. doi:10.1137/S0097539793254376. 47

[Har69] Frank Harary. *Graph theory*. Addison-Wesley, 1969. 14

[HBB$^+$21] Svein Høgemo, Benjamin Bergougnoux, Ulrik Brandes, Christophe Paul, and Jan Arne Telle. On Dasgupta's Hierarchical Clustering Objective and Its Relation to Other Graph Parameters. In Evripidis Bampis and Aris Pagourtzis, editors, *Fundamentals of Computation Theory - 23rd International Symposium, FCT 2021, Athens, Greece, September 12-15, 2021, Proceedings*, volume 12867 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 2021. doi:10.1007/978-3-030-86593-1_20. 5, 9, 28, 29, 41, 72, 95

[HFZ$^+$20] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs, 2020, 2005.00687. 157

[HK99] Monika R. Henzinger and Valerie King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM*, 46(4):502–516, July 1999. doi:10.1145/320211.320215. 137

[HLM86] D. S. Hirschberg, L. L. Larmore, and M. Molodowitch. Subtree weight ratios for optimal binary search trees. Technical report, UC Irvine: Donald Bren School of Information and Computer Sciences, 1986. 51

[Hor45] Robert E. Horton. Erosional development of streams and their drainage basins; hydrophysical approach to quantitative morphology. *Geological society of America bulletin*, 56(3):275–370, 1945. doi:10.1130/0016-7606(1945)56[275:EDOSAT]2.0.CO;2. 198

[HP11] Sariel Har-Peled. *Geometric approximation algorithms / Sariel Har-Peled.* American Math. Soc., Providence, RI, 2011. URL `https://sarielhp.org/book/`. 11

[HRR23] Jacob Holm, Eva Rotenberg, and Alice Ryhl. Splay Top Trees. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 305–331. Society for Industrial and Applied Mathematics, January 2023. doi:10.1137/1.9781611977585.ch28. 116, 137

[HT84] Dov Harel and Robert Endre Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024. 137

[IK75] Oscar H. Ibarra and Chul E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *J. ACM*, 22(4):463–468, October 1975. doi:10.1145/321906.321909. 87, 92

[IKK$^+$23] Takehiro Ito, Naonori Kakimura, Naoyuki Kamiyama, Yusuke Kobayashi, Shun-ichi Maezawa, Yuta Nozaki, and Yoshio Okamoto. Hardness of Finding Combinatorial Shortest Paths on Graph Associahedra. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*, volume 261 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 82:1–82:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2023.82. 9, 170

[IRV88] Ananth V. Iyer, H.Donald Ratliff, and G. Vijayan. Optimal node ranking of trees. *Inf. Process. Lett.*, 28(5):225–229, 1988. doi:10.1016/0020-0190(88)90194-9. 9

[ISO23] ISO/IEC. Working Draft, Standard for Programming Language C++, May 2023. URL `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf`. 1

[JCLM10] Tobias Jacobs, Ferdinando Cicalese, Eduardo Sany Laber, and Marco Molinaro. On the complexity of searching in trees: Average-case minimization. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I*, volume 6198 of *Lecture Notes in Computer Science*, pages 527–539. Springer, 2010. doi:10.1007/978-3-642-14165-2_45. 9

[Jor69] Camille Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik*, 70:185–190, 1869. 4, 47

[Kar00] David R. Karger. Minimum Cuts in Near-Linear Time. *J. ACM*, 47(1):46–76, January 2000. doi:10.1145/331605.331608. 137

[KH79] Oded Kariv and S. Louis Hakimi. An Algorithmic Approach to Network Location Problems. II: The p-Medians. *SIAM J. Appl. Math.*, 37:539–560, 1979. doi:10.1137/0137041. 47, 56, 59, 60, 62

[Kin92] Nancy G. Kinnersley. The vertex separation number of a graph equals its path-width. *Inf. Process. Lett.*, 42(6):345–350, 1992. doi:10.1016/0020-0190(92)90234-M. 198

[KL23] Tuukka Korhonen and Daniel Lokshtanov. An Improved Parameterized Algorithm for Treewidth. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 528–541. ACM, 2023. doi:10.1145/3564246.3585245. 27

[KMN$^+$23] Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michal Pilipczuk, and Marek Sokołowski. Dynamic treewidth. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1734–1744. IEEE, 2023. doi:10.1109/FOCS57990.2023.00105. 161

[KMT03] Haim Kaplan, Eyal Molad, and Robert E Tarjan. Dynamic rectangular intersection with priorities. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 639–648, 2003. doi:10.1145/780542.780635. 137

[KN13] Haim Kaplan and Yahav Nussbaum. Min-Cost Flow Duality in Planar Networks, 2013, 1306.6728. 137

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1968. 4

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, March 1971. doi:10.1007/BF00264289. 5, 37, 38, 39, 58, 71, 72, 198

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973. 1, 2

[Kőn90] Denes Kőnig. *Theory of finite and infinite graphs*. Birkhäuser, 1990. doi:10.1007/978-1-4684-8971-2. 47, 51

[Koz16] László Kozma. *Binary search trees, rectangles and patterns*. PhD thesis, Saarland University, 2016. URL `https://scidok.sulb.uni-saarland.de/frontdoor.php?source_opus=6646`. 101, 105, 114

[KPR$^+$14] Tomasz Kociumaka, Jakub Pachocki, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient counting of square substrings in a tree. *Theoretical Computer Science*, 544:60–73, 2014. doi:10.1016/j.tcs.2014.04.015. 47

[KS21] Christian Komusiewicz and Frank Sommer. Enumerating connected induced subgraphs: Improved delay and experimental comparison. *Discret. Appl. Math.*, 303:262–282, 2021. doi:10.1016/J.DAM.2020.04.036. 72

[Liu89] Joseph W.H Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11(1):73–91, 1989. doi:10.1016/0167-8191(89)90064-1. 33

[LK23] Calvin Leng and David Kempe. Binary Search with Distance-Dependent Costs, 2023, 2303.06488v1. 74, 209

[LM11] Eduardo Sany Laber and Marco Molinaro. An Approximation Algorithm for Binary Searching in Trees. *Algorithmica*, 59(4):601–620, 2011. doi:10.1007/S00453-009-9325-0. 9

[LS85] Nathan Linial and Michael E. Saks. Searching ordered structures. *J. Algorithms*, 6(1):86–103, 1985. doi:10.1016/0196-6774(85)90020-3. 10

[LT19] Caleb C. Levy and Robert E. Tarjan. A Foundation for Proving Splay is Dynamically Optimal, 2019, 1907.06310. 114

[Luc88] J.M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical report DCS-TR-250, Department of Computer Science, Hill Center for the Mathematical Sciences, Busch Campus, Rutgers University, New Brunswick, New Jersey 08903, 1988. 102, 209

[LW20] Victor Lecomte and Omri Weinstein. Settling the relationship between Wilber's bounds for dynamic optimality. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 68:1–68:21, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.68. 200

[LY98] Tak Wah Lam and Fung Ling Yue. Edge ranking of graphs is hard. *Discret. Appl. Math.*, 85(1):71–86, 1998. doi:10.1016/S0166-218X(98)00029-8. 9

[LY01] Tak Wah Lam and Fung Ling Yue. Optimal Edge Ranking of Trees in Linear Time. *Algorithmica*, 30(1):12–33, 2001. doi:10.1007/S004530010076. 9, 10

[Meh75] Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5(4):287–295, December 1975. doi:10.1007/BF00264563. 5, 37, 38, 48, 168

[Meh77] Kurt Mehlhorn. A Best Possible Bound for The Weighted Path Length of Binary Search Trees. *SIAM J. Comput.*, 6(2):235–239, 1977. doi:10.1137/0206017. 5, 37, 38, 49

[MOW08] Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 1096–1105, USA, 2008. Society for Industrial and Applied Mathematics. doi:10.5555/1347082.1347202. 9, 10, 11

[MP15] Thibault Manneville and Vincent Pilaud. Graph Properties of Graph Associahedra. *Séminaire Lotharingien de Combinatoire*, 73, 2015. URL `https://www.mat.univie.ac.at/~slc/wpapers/s73mannpil.html`. 18, 165, 166, 170

[Mun00] J. Ian Munro. On the Competitiveness of Linear Search. In Mike S. Paterson, editor, *Algorithms - ESA 2000*, pages 338–345, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/3-540-45253-2_31. 102, 209

[NOdM06] Jaroslav Nešetřil and Patrice Ossona de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *European Journal of Combinatorics*, 27(6):1022–1041, 2006. doi:https://doi.org/10.1016/j.ejc.2005.01.010. 29

[NOdM12] Jaroslav Nešetřil and Patrice Ossona de Mendez. Bounded Height Trees and Tree-Depth. In *Sparsity: Graphs, Structures, and Algorithms*. Springer, 2012. 5

[OP06] Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 379–388, 2006. doi:10.1109/FOCS.2006.32. 3, 4

[Ora24] Oracle. Map (Java SE 22 & JDK 22), 2024. URL `https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/Map.html`. Accessed 2024/09/02. 1

[Pal87] Jean Pallo. On the rotation distance in the lattice of binary trees. *Inf. Process. Lett.*, 25(6):369–374, 1987. doi:10.1016/0020-0190(87)90214-6. 165

[Pot88] Alex Pothen. The Complexity of Optimal Elimination Trees. Technical Report CS-88-16, The Pennsylvania State University, University Park, PA 16802, April 1988. 5, 29

[Pou14a] Lionel Pournin. The asymptotic diameter of cyclohedra. *Israel Journal of Mathematics*, 219:609–635, 2014. doi:10.1007/s11856-017-1492-0. 166

[Pou14b] Lionel Pournin. The diameter of associahedra. *Advances in Mathematics*, 259:13–42, 2014. doi:10.1016/j.aim.2014.02.035. 8, 165, 166

[PRW08] Alex Postnikov, Victor Reiner, and Lauren Williams. Faces of generalized permutohedra. *Documenta Mathematica*, 13:207–273, 2008. doi:10.4171/dm/248. 166

[Pyt24] Python Software Foundation. Mapping Types – dict, 2024. URL `https://docs.python.org/3/library/stdtypes.html#mapping-types-dict`. Accessed 2024/09/02. 1

[Ros70] Donald J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970. doi:10.1016/0022-247X(70)90282-9. 30

[Ros72] Donald J. Rose. A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations. In Ronald C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972. doi:10.1016/B978-1-4832-3187-7.50018-0. 30

[Rou21] Tim Roughgarden, editor. *Beyond the Worst-Case Analysis of Algorithms.* Cambridge University Press, 2021. 5

[RP89] Arnon Rosenthal and José A. Pino. A generalized algorithm for centrality problems on trees. *J. ACM*, 36(2):349–361, 1989. doi:10.1145/62044.62051. 47

[RRVS14] Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, and Somnath Sikdar. A Faster Parameterized Algorithm for Treedepth. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming*, pages 931–942, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-662-43948-7_77. 5

[RS83] Neil Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983. doi:10.1016/0095-8956(83)90079-5. 32

[RS86] Neil Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4. 32

[Rus] Rust team. std::collections – Rust. URL `https://doc.rust-lang.org/std/collections/`. Accessed 2024/09/02. 1

[Sch89a] Petra Scheffler. *Die Baumweite von Graphen als ein Maß für die Kompliziertheit algorithmischer Probleme.* PhD thesis, Akademie der Wissenschaften der DDR, Karl-Weierstrass-Institut für Mathematik, 1989. 198

[Sch89b] Alejandro A. Schäffer. Optimal node ranking of trees in linear time. *Inf. Process. Lett.*, 33(2):91–96, 1989. doi:10.1016/0020-0190(89)90161-0. 4, 42

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5. 117, 137, 138, 152

[ST85a] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985. doi:10.1145/2786.2793. 7, 107

[ST85b] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32:652–686, 1985. doi:10.1145/3828.3835. 6, 102, 113, 114, 117, 120, 122, 124, 137, 138, 152, 154

[Sta15] Richard P Stanley. *Catalan numbers.* Cambridge University Press, 2015. 72

[Str57] Arthur N. Strahler. Quantitative analysis of watershed geomorphology. *Eos, Transactions American Geophysical Union*, 38(6):913–920, 1957. doi:10.1029/TR038i006p00913. 198

[STT88] Daniel D. Sleator, Robert E. Tarjan, and William P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *Journal of the American Mathematical Society*, 1(3):647–681, 1988. doi:10.1090/S0894-0347-1988-0928904-4. 165, 166

[SV17] Fernando Sánchez Villaamil. *About Treedepth and Related Notions.* PhD thesis, RWTH Aachen, 2017. 5, 10, 32

[Tam54] Dov Tamari. Monoïdes préordonnés et chaînes de malcev. *Bulletin de la S. M. F.*, 82:53–96, 1954. 8, 165

[Tar85] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, April 1985. doi:10.1137/0606031. 120

[Tar97] Robert E. Tarjan. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78(2):169–177, August 1997. doi:10.1007/bf02614369. 137

[TW05] Robert E. Tarjan and Renato F. Werneck. Self-Adjusting Top Trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 813–822, USA, 2005. Society for Industrial and Applied Mathematics. 137

[TW10] Robert E. Tarjan and Renato F. Werneck. Dynamic Trees in Practice. *ACM J. Exp. Algorithmics*, 14, January 2010. doi:10.1145/1498698.1594231. 137, 155, 157

[Vod23] Johannes Voderholzer. Algorithms for optimal search trees on trees. Bachelor's Thesis, April 2023. URL `https://www.mi.fu-berlin.de/inf/groups/ag-ti/theses/bachelor_finished/voderholzer_johannes/index.html`. 58, 72

[Wer06a] Renato F. Werneck. *Design and Analysis of Data Structures for Dynamic Trees.* PhD thesis, Princeton University, 2006. 154

[Wer06b] Sebastian Wernicke. Efficient detection of network motifs. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 3(4):347–359, 2006. doi:10.1109/TCBB.2006.51. 72

[Wil89] Robert Wilber. Lower Bounds for Accessing Binary Search Trees with Rotations. *SIAM J. Comput*, 18(1):56–67, 1989. doi:10.1137/0218004. 99, 102, 105, 177, 180, 200

[Wol62] E. S. Wolk. The comparability graph of a tree. *Proc. Amer. Math. Soc.*, 13:789–795, 1962. doi:10.1090/S0002-9939-1962-0172273-0. 28

[Wol65] E. S. Wolk. A note on "the comparability graph of a tree". *Proc. Amer. Math. Soc.*, 16:17–20, 1965. doi:10.1090/S0002-9939-1965-0172274-5. 28

[Yan81] Mihalis Yannakakis. Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981. doi:10.1137/0602010. 28, 41, 95

[Yao80] F. Frances Yao. Efficient Dynamic Programming Using Quadrangle Inequalities. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 429–435, New York, NY, USA, 1980. ACM. doi:10.1145/800141.804691. 39

Zusammenfassung

Suchbäume auf Graphen (SBGs) sind eine weitreichende Verallgemeinerung binärer Suchbäume. Bei SBGs ist der Suchraum ein *Graph* statt einer totalen Ordnung. Die Idee ist, *Knoten* statt Schlüssel zu "vergleichen", was uns Informationen über die relative Position der Knoten im Graphen gibt.

SBGs können einerseits als Datenstruktur für Knotensuche in bestimmten Graphen gesehen werden, wobei der Graph Knoten-"Vergleiche" erlauben muss (dies ist z.B. bei sogenannten *Quaternärbäumen* der Fall). Andererseits sind SBGs eine hierarchische Zerlegung dieser Graphen. Suchbäume sind im letzteren Sinne unter verschiedenen Namen in der Literatur zu finden (z.B. als *Eliminationsbäume*), und haben enge Verbindungen zu den bekannten Konzepten *Baumtiefe* und *Baumweite*.

Viele algorithmische und kombinatorische Probleme zu binären Suchbäumen erlauben eine natürliche Verallgemeinerung zu SBGs. In dieser Arbeit konzentrieren wir uns auf die folgenden drei.

- Nimm an, uns ist eine Eingabeverteilung bekannt und wir möchten einen SBG berechnen, der die erwartete Suchzeit minimiert. Solche SBGs heißen *optimale statische Suchbäume*. Optimale *binäre Suchbäume* können in quadratischer Zeit berechnet werden. Für SBGs ist nicht einmal ein Polynomialzeitalgorithmus bekannt, selbst wenn der zugrundeliegende Graph ein Baum ist.

  Wir besprechen mehrere Approximationsalgorithmen für Bäume und Graphen mit beschränkter Baumweite, sowie die NP-Schwere des Problems im Allgemeinen.

- *Dynamische Suchbäume* dürfen während der Benutzung mit sogenannten *Rotationen* verändert werden (anders als statische Suchbäume). In dieser Arbeit verallgemeinern wir die sogenannten *Spreizbäume* (engl. *splay trees*) von binären Suchbäumen zu Suchbäumen auf *Bäumen*. Weiterhin benutzen wir diesen Algorithmus, um eine Datenstruktur für *dynamische Wälder* zu implementieren, die wir experimentell auswerten.

- Der *Rotationsabstand* zwischen zwei SBGs ist die minimale Anzahl an Rotationen, die benötigt werden, um einen der SBGs in den anderen umzuformen. Uns interessiert der maximale Rotationsabstand zwischen Suchbäumen auf einem gegebenen Graphen, was der Durchmesser des sogenannten *Graphenassoziaeder* ist. Überraschenderweise ist dieses kombinatorische Problem eng mit den statischen und dynamischen Suchbaumproblemen verbunden. Wir besprechen mehrere neue Ergebnisse, unter anderem einen einfachen Algorithmus zur Berechnung des Durchmessers des Graphenassoziaeder falls der zugrundeliegende Graph ein Baum mit beschränkter *Wegbreite* ist.