Chapter 3

Point Labeling: Label-Number Maximization

Generally it is assumed that a point label can be seen as an axis-parallel rectangle; the bounding box of the text, see Figure 3.1. Many algorithms for point labeling have been described in the automated cartography literature and in computational geometry. For an extensive bibliography see [WS96].

Good point labeling has two basic requirements. A label should be placed close to the point to which it belongs, and two labels should not overlap each other. For high quality cartographic label placement, further requirements have been formulated [Imh75, Yoe72]. Given the basic requirements, an algorithm can try to either label as many points as possible, or find the largest possible font such that all points can be labeled. In general, both of these problems are NP-hard [FW91, MS91]. In this chapter, we focus on the former problem, i.e. label-number maximization. As in Chapter 2, we are given a set of features (here: points) and for each point a set of label candidates. A solution is a function that maps every point to 0 or to a label from its set of candidates such that no two labels intersect. The size of a solution is the number of points that receive a label. An optimal solution is a solution of maximum size.

Although finding an optimal solution is NP-hard, approximation algorithms have been suggested. For axis-parallel rectangular labels of arbitrary height and width, Agarwal et al. propose an algorithm with an approximation ratio of $1/O(\log n)$ [AvKS98]. Their algorithm is based on divide-and-conquer. If the label height (or width) is fixed, the same paper suggests a line-stabbing algorithm that labels in $O(n \log n)$ time at least half the number of points that are labeled in an optimal solution. For maximizing the *size* of uniform rectangular labels, this approximation factor is optimal, but for maximizing the *number* of fixed-height labels, Agarwal et al. also present a polynomial time approximation scheme (PTAS).

Nearly all of the existing algorithms for point labeling limit the placement of a label with respect to its point to a finite number of label positions. Most algorithms described before allow four label candidates, namely those where a

rectangular label touches its point in one of its four corners [FW91, WW97]. In the automated cartography literature eight candidates per point is also quite common, while the approximation algorithm of Agarwal et al. [AvKS98] allows any constant number as does a heuristic proposed by Kakoulis and Tollis [KT98]. Their algorithm is based on a heuristic for splitting connected components into cliques and uses maximum-cardinality bipartite matching.

We call restrictions of the allowed label positions a *labeling model*. Models that allow a finite number of positions per label are *fixed-position models*, those that allow an infinite number are *slider models*.

Figure 3.1: Rectangular labels of cities of the U.S.A.

This chapter is structured as follows. Section 3.1 introduces six point-labeling models; three fixed-position and three slider models. We analyze how many more labels can be placed in one model than another, in theory.

In Section 3.2 we specialize the general concept of Chapter 2 to the context of point labeling and give the details of three variants of an algorithm based on this concept. Our algorithm has a runtime of $O(k + n \log n)$, where n is the number of points and k the number of intersections among the label candidates. Other than all approximation algorithms suggested so far, our algorithm does not make any assumptions about label shapes and the position of a label relative to its point. Due to this generality we could not expect to prove any approximation factors. However, our algorithm works very well in practice. We compare it experimentally to a number of other methods.

In Section 3.3 we drop the restriction that a label can only be placed at a finite number of positions. Instead, we allow any position where an edge of the label is incident to the point, see Figure 3.1. We show that it is NP-

complete to decide whether a set of points can be labeled with unit squares in the four-slider model. However, each of our three slider models allows a simple factor-$\frac{1}{2}$ approximation algorithm that uses $O(n)$ space and $O(n \log n)$ time. We also give a polynomial-time approximation scheme for each of our slider models. Similar results were already known for fixed-position models [AvKS98]. In order to support the practical relevance of our approximation algorithms for the three slider models, we do a thorough experimental analysis on real-world data and randomly generated point sets.

3.1 Comparing Various Models

This section is joint work with Marc van Kreveld and Tycho Strijk, both Universiteit Utrecht [vKSW98, vKSW99].

In this section we introduce and then compare some common point-labeling models. All of the algorithms we present in the following sections aim to label as many points as possible according to the chosen model.

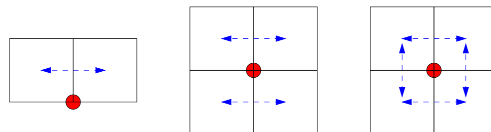


Figure 3.2: top-, two- and four-slider model

Definition 3.1 (point labeling, size of a labeling, optimum labeling) *Given a set P of n points in the plane, and for each point $p \in P$ a set of label candidates L_p, a* point labeling *is a subset $P' \subseteq P$ and a function λ which maps every point $p \in P'$ to a label $\lambda(p) \in L_p$ such that no two labels intersect. The number of labeled points, i.e. the cardinality of P', is the* size of the point labeling. *An* optimal labeling *is a point labeling which has maximum size among all point labelings.*

In this chapter, we restrict ourselves to axis-parallel rectangular label candidates. If we require additionally that a label must be placed such that one of its edges contains the point to be labeled, we get the following labeling models.

Definition 3.2 (slider models) *In the* four-slider model*, a point p must be labeled such that any edge of the label contains p. In the* two-slider model*, either the label's top or bottom edge has to contain p. In the* one-slider *or* top-slider *model, the bottom edge of a label must contain p.*

For an illustration of slider models, see Figure 3.2. Note that in all of our models we allow that a label contains other points which then of course cannot be labeled. Our labels are closed, i.e., we disallow touching. One alternative

would be "half-open" labels as in [WW97]. In that paper all edges of a label which are not adjacent to its point are allowed to touch other labels or points. This would make sure that if every label is scaled down by a small amount with its point as scaling center, then all labels are disjoint. When labels do not touch, a map user can more easily match a label and the point to which it belongs. The algorithms could be adjusted to this additional requirement, but intensive case study would be necessary to decide whether a label can be placed when it touches other labels. The bounds of the following comparison of models would still hold, but for the sake of simplicity we keep the number of requirements to a minimum.

One alternative would be to consider labels open and thus allow touching generally. In this case however, we were not able to keep the greedy algorithm's approximation guarantee of 50%, although the bounds of the comparison below would hold.

We will compare the slider models introduced above to the following fixed-position models.

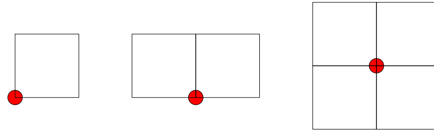


Figure 3.3: one-, two- and four-position model

Definition 3.3 (fixed-position models) *Labeling in the* four-position model *requires that the a point p is labeled such that one of the label's corners lies on p. In the* two-position model *one of the label's bottom corners must lie on p and in the* one-position model *the lower left corner of the label must coincide with p.*

For an illustration of fixed-position models, see Figure 3.3. Our measure for comparing the models above is based on optimal labelings of point sets. Some point sets allow a labeling of the whole set in all models. Such point sets are not very interesting for a comparison, so we are mainly interested in point sets where the size of an optimal labeling differs from model to model. We define the *ratio* of two models as follows.

Definition 3.4 (ratio of two models) *Given unit square label candidates and two label-placement models M_1 and M_2, the (asymptotic)* $(M_1 : M_2)$-ratio *is*

$$\lim_{n\to\infty} \max_{P, |P|=n} \frac{\text{size}(\textit{optimal } M_1\textit{-labeling for } P)}{\text{size}(\textit{optimal } M_2\textit{-labeling for } P)}.$$

This measure does *not* take into account aesthetic criteria as listed by Imhof [Imh75]. Since it is a purely quantitative measure and, moreover, only refers

to square labels, it does not apply directly to many practical label placement problems. However, it gives a fair indication of how many more points can be labeled in one model than in another in general.

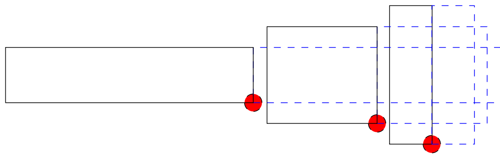


Figure 3.4: The ratio between the two- and the one-position model can become arbitrarily large for labels of different size.

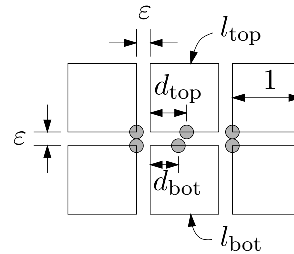


Figure 3.5: 3/2 is a lower bound on the ratio between the two- or four-slider and any fixed-position model

The reason why we only consider unit square labels in the definition above and in the remainder of this section, is that otherwise some of the ratios between two models would become arbitrarily bad, see Figure 3.4. All points depicted there can be labeled in the two-position model, but only one point can be labeled in the one-position model.

It is worth mentioning that the size of an optimal placement in a slider model cannot be approximated arbitrarily well by a fixed-position model, no matter at how many discrete positions a fixed-position label can be attached to its point. Given such a model, consider all positions in which a unit square label can be attached to its point. W.l.o.g. we may assume that the four corner positions are among them. For each position, mark the place on the edge of the label that the point touches. Choose some $\varepsilon > 0$ to be smaller than half the minimum distance between two markers that both lie either on the top or on the bottom edge of the label. Then there must be points p_{top} and p_{bot} on the top and the bottom label edge, respectively, that are further than ε away from any marker. Let d_{top} and d_{bot} be their respective distances to the label's left edge.

Now consider the six points marked by disks in Figure 3.5. The two leftmost points have vertical distance ε from each other and horizontal distance $1 + 2\varepsilon$ from the corresponding rightmost points. These four points can be labeled in all models that allow any corner of a label to lie on the point to be labeled. The other two points lie at a distance of $\varepsilon + d_{\text{top}}$ and $\varepsilon + d_{\text{bot}}$ to the right of the leftmost points. These two points can be labeled by labels l_{top} and l_{bot} in the two- or four-slider model such that l_{top} and l_{bot} keep a distance of ε to the labels of the left- and rightmost points. However these points cannot be labeled properly in the given fixed-position model since moving l_{top} and l_{bot} by up to ε to the right or left does not make the points coincide with any of the markers at the top or bottom edge of l_{top} and l_{bot}. This is due to our choice of d_{top} and d_{bot}.

In order to get larger point sets as required by Definition 3.4, we simple copy the group of six points depicted in Figure 3.5 at regular intervals along the x-axis. This yields a ratio of 3/2 between the two- (or four-) slider model and any given fixed-position model.

The labeling models used in this section will be the six introduced in Definition 3.2 and 3.3. All of our comparisons of two such models M_1 and M_2 are based on the following strategy. We want to bound the ratio Ψ by which more labels can be placed in the model with more freedom, say M_1. We assume an optimal label placement in M_1. Then we canonically relabel the labeled points by moving every label into a position that is valid in the more restrictive model M_2. This might cause some labels to intersect. We determine the maximum number δ_{left} of M_2-labels that intersect the leftmost M_2-label l. Then we put l into a set S of non-intersecting labels, remove l and all its conflicting labels from the instance and repeat until no labels remain. At the end of the process, S contains at least $k^1_{\text{opt}}/(\delta_{\text{left}}+1)$ non-intersecting M_2-labels, where k^1_{opt} is the size of the assumed optimal M_1-placement. The size of S is a lower bound for the size of an optimal M_2-placement, thus $\delta_{\text{left}}+1$ is an upper bound for the $(M_1 : M_2)$-ratio. Lower bounds for the ratio Ψ are obtained by giving examples of arbitrary size for which any M_2-placement is worse by a certain factor than some M_1-placement.

Since we do not want to compare every two models in isolation, we define three groups. They consist of pairs of models where points with labels in one model can be canonically relabeled such that a certain fraction of points gets labels in the other model.

Definition 3.5 (flipping) *Given two different label placement models M_1 and M_2, and an axis-parallel vector v of unit length, model M_1 can be* flipped into model M_2 by v *if any label position in M_1 that is not allowed in M_2 can be translated by v into a valid M_2-label position.*

Example 3.6 The two-slider model can be flipped into the top-slider model by $(0,1)$. Analogously, the four-position model can be flipped by $(0,1)$ into the two-position model, while the two-position model can be flipped by $(1,0)$ into the one-position model.

Lemma 3.7 *For any two labeling models M_1 and M_2 where M_1 can be flipped into M_2 the $(M_1 : M_2)$-ratio is 2.*

Proof. Consider an optimal M_1-labeling of an arbitrary instance of points. Let M_2 be a model into which M_1 can be flipped by a vector v. Then the canonical relabeling mentioned above means translating by v all M_1-labels that are not valid in M_2.

We can assume that the vector by which we flip is $(0,1)$; the case $(1,0)$ is symmetric. This means that an M_2-label is either identical to the corresponding M_1-label or lies one unit above it. Let l_1 be the M_1-label corresponding to the

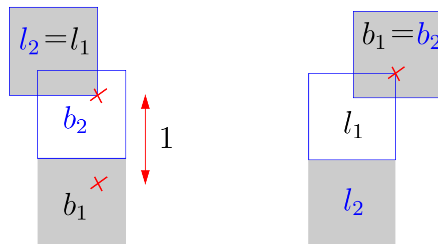


Figure 3.6: If M_1 can be flipped into M_2 then the leftmost M_2-label l_2 (solid edges) cannot intersect more than one M_2-label b_2. (The corresponding non-intersecting M_1-labels are shaded.)

leftmost M_2-label l_2. We show that l_2 can intersect at most one M_2-label whose M_1-counterpart is not in conflict with l_1. As indicated above, this gives us an upper bound of 2 for the $(M_1 : M_2)$-ratio Ψ.

Suppose that l_2 is identical to the corresponding M_1-label l_1; the other case is symmetric, see the left and right part of Figure 3.6, respectively. Let I_2 be the set of all M_2-labels intersecting l_2 and let I_1 be the set of their mutually non-intersecting M_1-counterparts. Then all labels in I_2 must contain the lower right corner of l_2; otherwise, either their M_1-counterparts intersect l_1, or l_2 is not leftmost. This however forces all labels in I_1 to contain a point at unit distance below that corner (marked by a cross in Figure 3.6) in order not to intersect l_1. Hence $|I_1| = |I_2| \leq 1$ and $\Psi \leq 2$.

In order to establish the lower bound of 2 for Ψ, just take the four corner points of an axis-parallel square of edge length less than one. For all models M_1 that we are considering and that can be flipped into a model M_2 (see Example 3.6), exactly twice as many of these points can be labeled as in the corresponding M_2-model. An instance can consist of arbitrarily many of such groups of four points, separated sufficiently. ∎

Definition 3.8 (sliding) *Given two different label placement models M_1 and M_2, and an axis-parallel vector v of unit length, model M_1 can be* slid into model M_2 along v *if every label position in M_1 can be translated by μv into a valid M_2-label position for some $\mu \in [0, 1]$*

Example 3.9 The four-slider model can be slid into both the two-slider and the top-slider model along $(0, 1)$. Along $(1, 0)$ we can slide the two-slider into the four-position model and the top-slider into both the two- and the one-position model. Note that the four-slider model cannot be slid into the four-position model.

Lemma 3.10 *Let M_1 and M_2 be two (different) labeling models where M_1 can be slid into M_2, and let Ψ be the $(M_1 : M_2)$-ratio. Then $2 \leq \Psi \leq 3$.*

Proof. Again we consider an optimal M_1-labeling of an arbitrary instance. We assume that we can slide M_1- into M_2-label positions along $(0,1)$; the case $(1,0)$ is symmetric. We canonically slide all M_1-labels upwards until we arrive in an M_2-label position. We show that the leftmost M_2-label l_2 can then intersect at most two other M_2-labels. This yields the upper bound of 3 for Ψ.

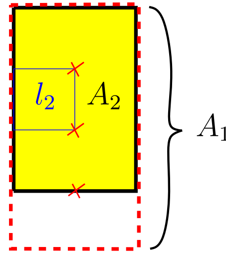


Figure 3.7: If M_1 can be slid into M_2 then the leftmost M_2-label l_2 cannot intersect more than two M_2-labels.

M_2-labels intersecting l_2 can only lie within area A_2, a rectangle of width two and height three that is placed such that its left edge is centered at the left edge of l_2, see Figure 3.7. This holds because l_2 is chosen to be leftmost. The corresponding M_1-labels are restricted to area A_1, a rectangle of width two and height four obtained by extending A_2 one unit downwards. Every label in A_1 must contain one of the three grid points in the interior of A_1 marked by crosses in Figure 3.7. Thus A_1 can contain only three non-intersecting M_1-labels including the M_1-counterpart of l_2. It follows that l_2 cannot intersect more than two M_2-labels, and hence that $\Psi \leq 3$.

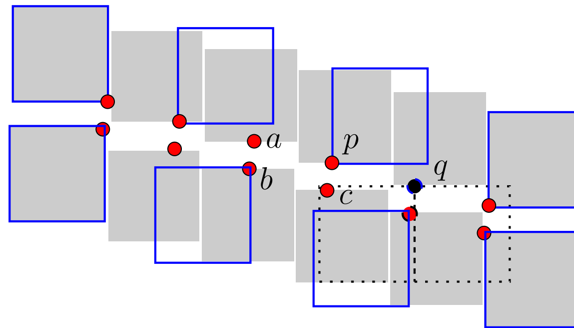


Figure 3.8: If M_1 can be slid into M_2 then the M_1-M_2-ratio approaches 2. Here we chose M_1 to be the two-slider model (shaded labels) and M_2 to be the four-position model (solid edges).

For a lower bound that approaches 2 refer to Figure 3.8. There are two rows of n points. Two neighboring points of one row have x-distance $1-\frac{1}{n-1}+\varepsilon$ and y-distance δ, where $\frac{1}{n-1} > \delta > \varepsilon > 0$. The upper row is a copy of the lower, shifted by the vector $(\delta/2, \delta)$.

Comparing the top-slider model to the one- or two-position model is easy;

just consider one row. In order to compare the four- to the two-slider model, the figure must be rotated by 90 degrees. So let us focus on comparing the two-slider to the four-position model here.

It is obvious that all points can be labeled in the two-slider model. For the four-position model we can argue as follows. Let p be a point of the upper row excluding the last or first two points and let q be the right neighbor of p in the upper row. If a four-position label is attached to p, then either it contains at least one extra point (a, b and c, or q in Figure 3.8), or it makes labeling q difficult. Either q is not labeled, or q's label is in a lower position and hence q will contain at least two extra points. Since p is the only point whose label can intersect the upper positions of q without intersecting q itself, a failure to label q can be uniquely charged to p. And if p and q are both labeled, we charge the failure to label the two points in q's label to p and q. In any case, we can charge one point that cannot be labeled to each point that is labeled. For the corresponding points p of the lower row, the same argument holds if we choose q to be the left neighbor of p. ◻

Note that the proof above can be simplified for closed labels. We chose to give a proof that carries over to the case of open labels.

The upper bounds for Ψ can be improved to 3 for the pairs of models satisfying the following definition.

Definition 3.11 (corner-sliding) *Given two different label placement models M_1 and M_2, model M_1 can be* corner-slid into model M_2 *if every label position in M_1 can be shifted both to the left* and *to the right such that the point coincides with a corner of the label, and these positions are valid in M_2. Vertical corner-sliding is defined with left and right replaced by top and bottom.*

Example 3.12 The top-slider model can be corner-slid into the two-position or the four-position model. We can corner-slide the two-slider model into the four-position model. The four-slider model can be vertically corner-slid into the two-slider model. Note that the four-slider model cannot be corner-slid into the four-position model since sliding would be necessary in two directions. The top-slider model cannot be corner-slid into the one-position model since top-slider labels cannot be slid to the left to reach a position in the one-position model.

Lemma 3.13 *Let M_1 and M_2 be two (different) labeling models where M_1 can be corner-slid into M_2. Then the $(M_1 : M_2)$-ratio is at most 2. The same holds if M_1 can be vertically corner-slid into M_2.*

Proof. Again we consider an optimal M_1-labeling of an arbitrary instance. We assume that we can corner-slide M_1- into M_2-label positions; the vertical corner-sliding case is symmetric. We draw a set of vertical lines with unit spacing over the M_1-labeling such that no line contains a point of the instance, nor a vertical

edge of a label. We count both the number of M_1-labels that intersect the odd lines and the number of M_1-labels that intersect the even lines. Assume that the even lines intersect the greater number of squares in the M_1-labeling; the other case is symmetric. Delete the squares and corresponding points intersecting the odd lines. The remaining squares are now corner-slid into a M_2-label position such that each label intersects an even line, see Figure 3.9.

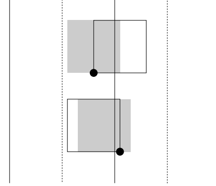


Figure 3.9: After removing M_1-labels intersecting the odd lines (dashed), the remaining M_1-labels (shaded) are corner-slid to intersect an even line (solid). This results in a valid M_2-labeling.

Note that if a given M_1-label intersects an even line, then the resulting label in the M_2-position intersects that same even line. Since the spacing between even lines is 2 and the lines are in non-degenerate position, two M_2-labels intersecting different even lines cannot intersect. Furthermore, two M_2-labels intersecting the same even line arose from two M_1-labels intersecting that same even line. Since corner-sliding is done horizontally, the M_2-labels do not intersect since the M_1-labels did not intersect.

Since we never remove more than half the M_1-labels, and the remaining M_1-labels are all corner-slid to non-intersecting M_2-labels, the $(M_1 : M_2)$-ratio is at most 2. ∎

Lemma 3.14 *Let $\Psi_{1S,1P}$ be the ratio between the top-slider and the one-position-model. Then $2\frac{1}{4} \leq \Psi_{1S,1P} \leq 3$*

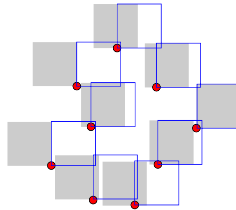


Figure 3.10: Sliding top-slider labels (shaded) to the right into one-position labels (solid edges) can create 9-cycles in the resulting conflict graph of one-position labels.

Proof. With Lemma 3.10 we get $2 \leq \Psi_{1S,1P} \leq 3$. The example in Figure 3.10 raises the lower bound to $2\frac{1}{4}$. ∎

Unfortunately we were not able to lower the upper bound of $\Psi_{1S,1P}$ although we can show the following. In order to get from an optimal top-slider labeling to a one-position labeling, we *must* shift all labels to the extreme right. Note that this is not the same as the idea of a canonical relabeling. If we consider the resulting conflict graph of the one-position labels, then this graph is planar, has maximum degree $\Delta = 4$ and all odd cycles have length at least 9. Grötsch's Theorem says that a planar and triangle-free graph is three-colorable, but this gives us only a more graph-theoretic proof of the upper bound of 3.

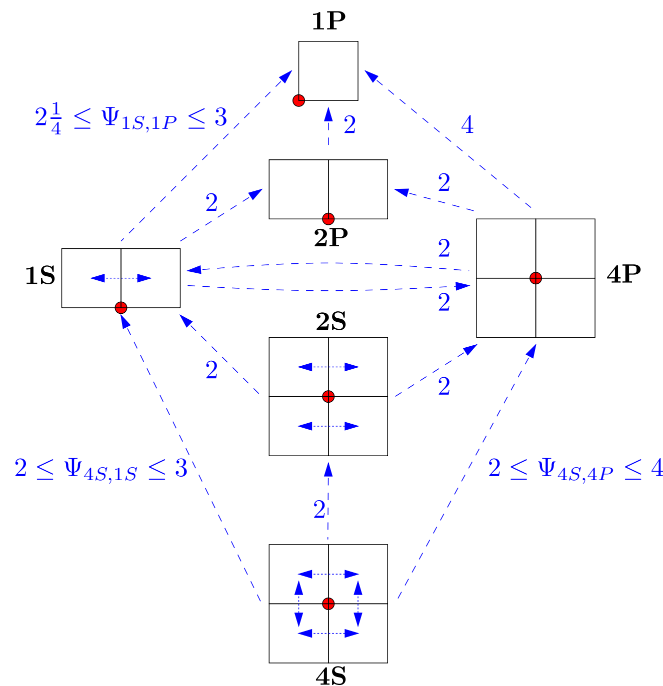


Figure 3.11: Ratios between some label placement models. From top to bottom: the one-position, two-position, top-slider (left), four-position (right), two- and four-slider model

Figure 3.11 summarizes our results. The reason for the upper bound 4 for the ratio $\Psi_{4S,4P}$ between the four-slider and the four-position model is the following. First we convert the four-slider labeling to a two-slider labeling as described in the proof of Lemma 3.13. After that we convert this two-slider labeling to a four-position labeling in the same way. Since both these conversions keep at least half the number of labels, we get $\Psi_{4S,4P} \leq 4$.

3.2 Fixed-Position Models

In this section we specialize the general concept of Chapter 2 to the context of point labeling. First, however, we review some of the previous approaches to maximizing the number of labeled points. For an extensive bibliography on label placement in general, refer to [WS96]. At the beginning of this chapter we have already mentioned a number of approximation algorithms. Let us add the most prominent heuristic methods.

In [CMS95] Christensen et al. compare simulated annealing to gradient descent, to an incremental force-driven algorithm by Hirsch [Hir82], and to 0-1 integer programming suggested by Zoraster [Zor86, Zor90]. The conclusion of their comparison is that simulated annealing is the method of choice for point labeling, see the results in Figure 3.72, page 79. Zoraster later also applied the simulated-annealing algorithm of Christensen et al. to labeling very dense point sets and reported good results [Zor97]. Edmondson et al. showed that the simulated annealing algorithm of Christensen et al. can also be used for general label placement, i.e. for arbitrary feature and label shapes [ECMS97].

Recently, Kakoulis and Tollis suggested another approach to label-number maximization [KT98] that is independent of the shape of the label candidates. The authors compute the *candidate conflict graph* G_{cand}. G_{cand} has a node for each candidate and an edge for each pair of intersecting candidates. In the next step they compute the connected components of G_{cand}. Then Kakoulis and Tollis use a heuristic similar to the greedy algorithm for maximum independent set to split these components into cliques. Finally they construct a bipartite "matching graph" whose nodes are the cliques of the previous step and the features of the instance. In this graph, a feature and a clique are joined by an edge if the clique contains a candidate of the feature. A maximum-cardinality matching yields the labeling. Given G_{cand}, the runtime of their algorithm depends on how the clique check and the matching algorithm are implemented. Practical matching algorithms take $O(k\sqrt{n})$ time; however, our implementation of their heuristic has an asymptotic runtime of $O(kn)$, where k refers to the number of edges in the candidate conflict graph. The authors do not give any time bounds.

The algorithm for label-number maximization we present here unites the following advantages. Our algorithmic approach

- does not depend on the shape of labels,
- can be applied to point, line, or area labeling (even simultaneously) if a finite set of label candidates has been precomputed for each feature,
- is easy to implement,
- runs fast (i.e. in $O(n + k)$ time given the candidate conflict graph), and
- returns good results in practice—at least for labeling points with rectangular labels.

To our knowledge, none of the algorithms described so far shares all of these

characteristics.

The input to our algorithm is the candidate conflict graph of the label candidates. The algorithm is divided into two phases similar to the first two phases of the algorithm for label size maximization described in [WW97]. In phase I, we apply a set of rules to all features in order to label as many of them as possible and to reduce the number of label candidates of the others. These rules do not destroy a possible optimal placement. Then, in phase II, we heuristically reduce the number of label candidates of each feature to at most one. Given the candidate conflict graph, our algorithm takes $O(k+n)$ time and $O(n)$ space.

In Section 3.2.1 we give the details of three variants of the algorithm based on this concept. In Section 3.2.2 we describe the set-up and in Section 3.2.3 the results of our experiments. We compare our algorithm to five other methods, namely simulated annealing, a greedy algorithm (see Section 3.3.2), two variants of the matching heuristic of Kakoulis and Tollis, and a hybrid algorithm that combines our rules with their clique matching.

Part of the examples on which we do the comparison are benchmarks that were already used in [WW97]. We added examples for placing rectangular labels of varying size, both randomly generated and from real world data. Our samples come from a variety of sources; they include the location of some 19,400 ground-water drill holes in Munich, 373 German railway stations, and 357 shops. The latter are marked on a tourist map of Berlin that is labeled on-line by our algorithm. The algorithm is also used by the city authorities of Munich to label drill-hole maps. All example generators, real world data and algorithms are available on the World Wide Web[1].

3.2.1 Algorithm

As an application of the ideas of the previous chapter, we describe three closely related variants of a label-placement algorithm. Although these algorithms do not make any assumptions about the features to be labeled or the label candidates, our description is based on the context of point labeling. This simplifies presentation and experimental evaluation. We used four rectangular, axis-parallel label candidates per point, namely those where one of the label's corners is identical to the point. Our objective is to label as many points as possible.

Each of our algorithms consists of two phases. In phase I, we try to remove as many label candidates as possible without destroying an optimal placement. Then, in phase II, we heuristically pick a candidate, remove it, and fall back on the methods of phase I to further reduce the number of label candidates. We repeat this process until each point has at most one label candidate left, and none of these intersects any of the other remaining candidates. These candidates form our solution.

[1] `http://www.math-inf.uni-greifswald.de/map-labeling/general`

While the heuristic in phase II is identical for all algorithms, the way they choose candidates for removal in phase I differs.

Rules applies a set of rules exhaustively to all points. Each rule tries to find a candidate that can be put into the solution and then removes all candidates of the same site or those that intersect the chosen candidate.

EI-1* establishes edge-irreducibility as described in Section 2.4.

EI+L3 establishes edge-irreducibility and additionally applies rule L3 that is described below.

Phase I of Algorithm Rules

This algorithm was joint work with Frank Wagner and Vikas Kapoor, both at Freie Universität Berlin.

In the first phase of algorithm Rules, we apply all of the following rules to each of the points. Let p_i be the label candidate of point p in position i. For each of the rules we supply a sketch of a typical situation in the context of point labeling. We shaded the candidates that are chosen to label their point, and we used dashed edges to mark candidates that are eliminated after a rule's application. We say that two label candidates (of distinct features) *are in conflict* if they intersect. The *conflict partners* of a candidate c are all those candidates that are in conflict with c. Finally let the *conflict number* of c be the number of conflict partners of c.

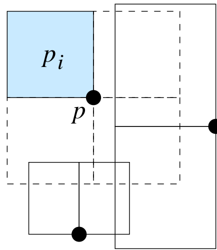


Figure 3.12: Rule **L1**

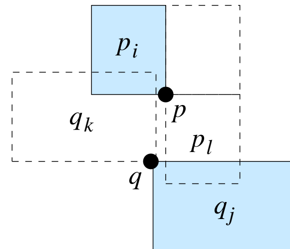


Figure 3.13: Rule **L2**

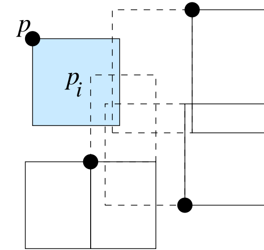


Figure 3.14: Rule **L3**

(L1) If p has a candidate p_i without any conflicts, declare p_i to be part of the solution, and eliminate all other candidates of p, see Figure 3.12.

(L2) If p has a candidate p_i that is only in conflict with some q_k, and q has a candidate q_j $(j \neq k)$ that is only overlapped by p_l $(l \neq i)$, then add p_i and q_j to the solution and eliminate all other candidates of p and q, see Figure 3.13.

(L3) If p has only one candidate p_i left, and the candidates overlapping p_i form a clique, then declare p_i to be part of the solution and eliminate all candidates that overlap p_i, see Figure 3.14.

We want to make sure that the rules are applied exhaustively. Therefore, after eliminating a candidate p_i, we check whether the rules can be applied in the *neighborhood* of p_i, i.e. to p or to the points of the conflict partners of p_i.

Similar to the rules A1 to A3 in Section 2.4 the rules L1 to L3 have the property that if there is a solution of size t (i.e. t points can be labeled) before applying any of the rules, then this is also the case after the rule's application. This is easy to show, see [WW98]. Note that L1 and L2 are special cases of A1 and A2.

Phase II

If we have not managed to reduce the number of candidates to at most one per point in the first phase, then we must do so in phase II. Since phase II is heuristical, we are no longer able to guarantee optimality. The heuristic RemoveLocalTroubleMakerMaxCandNumber described in Section 2.5 is conceptually simple and makes the algorithm work well in practice, see Section 3.2.2. The intuition is to start eliminating "troublemakers" where we still have a choice. Speaking more algorithmically, we go through all points p that have the maximum number of candidates, and delete the candidate with the maximum number m of conflicts among the candidates of p if $m \neq 0$. This process is repeated until each point has at most one candidate left and these candidates have no more conflicts. These candidates then form the solution.

As in phase I, after eliminating a candidate, we check whether our rules (i.e. L1 to L3 for Rules, A1 to A3 for EI-1*, and A1 to A3 plus L3 for EI+L3) can be applied in its neighborhood.

Analysis

Other than in Chapter 2 we assume here that the number of candidates per point is a small constant, typically four or eight for point labeling. Let n be the number of points, and k the number of pairs of intersecting candidates in the instance, i.e. the number of edges in the candidate conflict graph. Then EI-1* runs in $O(k)$ time according to Lemma 2.14. In order to bound the running time of Rules and EI+L3, we must analyze the time complexity of rule L3. Rules L1 and L2 are special cases of A1 and A2; thus they can also be checked in constant time per application, see Lemma 2.10.

We use a stack to make sure that our rules are applied exhaustively. After we have applied a rule successfully and eliminated a candidate, we put all points in its neighborhood on the stack and apply the rules to these points. A point is only put on the stack if one of its candidates was deleted or lost a conflict partner.

For rule L3, we have to check whether a candidate is intersected by a clique. In general, this takes time quadratic in the number of conflict partners. Falling back on geometry, however, can help to cut this down. In the case of axis-parallel rectangles, a clique can be detected in linear time by testing whether

the intersection of all conflicting rectangles is not empty. A simple charging argument then yields $O(k^2)$ total time for checking L3.

Note that for L3 other than for the heuristic RemoveLocalTroubleMaker-MaxCandNumber it is not enough to have access to the conflict number of a candidate c; we actually need access to each conflict partner of c. For implementing rules A1 to A3 one only needs to know whether c is in conflict with candidates of points other than a query point. Of course the candidate conflict graph gives access to all this information.

However, checking L3 can be done in constant time if we apply L3 only to candidates with less than a constant number of conflicts. This makes sense since it is not very likely that the neighborhood of a candidate with many conflicts is a clique. In this case, phase I can be done in $O(n + k)$ time.

In phase II, we can afford to simply go through all points sequentially and check whether they have the current maximum number of candidates. If so, we go through the candidates of the current point and determine the one with the maximum number of conflicts. The amount of time needed to delete this candidate and apply our rules has already been taken into account in phase I. Thus phase II needs only $O(dn)$ extra time.

Putting things together, Rules and EI+L3 take $O(n + k^2)$ time if rule L3 can be checked in linear time, and $O(n + k)$ time if we allow only constant effort for checking L3. In our experiments, we have not bounded this effort, yet this part of the algorithms showed a linear-time behavior. Finally, for axis-parallel rectangular labels, the candidate conflict graph can be determined in $O(k + n \log n)$ time and takes $O(k)$ space.

Thus our algorithms can be implemented to label n points in $O(k + n \log n)$ total time, given a constant number of axis-parallel rectangular label candidates per point and constant effort for checking L3. The algorithms require $O(n)$ storage apart from the candidate conflict graph.

A Hybrid Algorithm

This algorithm was joint work with Tycho Strijk, Universiteit Utrecht.

Since the decisions our algorithms make in phase II are only based on local properties of the candidate conflict graph, these decisions can be made very efficiently. Using more global information is time-costly, but might also improve the quality of the results. Therefore we thought that it would be interesting to combine our set of rules with the global aspect of the matching heuristic of Kakoulis and Tollis [KT98]. The resulting hybrid algorithm proceeds as follows.

As before, we compute the candidate conflict graph. In phase I, we apply our set of rules L1 to L3 exhaustively. In the new phase II, however, we use the heuristic proposed by Kakoulis and Tollis to break up the connected components of the candidate conflict graph into cliques. Recall that in every connected component, they determine the candidate with the highest degree, eliminate it, and repeat this process recursively until each component is a clique. The

choice of the candidate that is to be eliminated has the following exception. If the candidate belongs to a feature that has "few" candidates left, then the candidate with the second highest degree in the current connected component is eliminated, and so on.

Like in the old phase II, after each deletion, we apply our rules in the neighborhood of the eliminated candidate in order to propagate the effect of our heuristical decision.

As soon as all connected components are cliques, we use a maximum-cardinality bipartite-matching algorithm to match as many cliques with features as possible.

The new phase II can be implemented by an extended breadth-first search (BFS). First, we compute all connected components of the candidate conflict graph by common BFS. At the same time, we store the candidate with the highest, second-highest and the lowest degree for each component. To decide whether a component C is a clique, we simply check whether the vertex with minimum degree in C matches the number of vertices in C minus one. If this is not the case, we delete the vertex v_1 with the highest degree. There is one exception. Let a vertex be *important* if it corresponds to a candidate that is the last candidate of a feature. If v_1 is important and the vertex v_2 with the second highest degree in C is not important, then we delete v_2 instead of v_1.

Now let v be the vertex we deleted. We apply our rules in the neighborhood of v and then the extended BFS recursively to all vertices that were adjacent to v just before its deletion. In each level of the recursion at least one vertex is deleted, and each edge is considered at most twice by BFS. Thus, if each of the n features has a constant number of candidates, we have at most $O(n)$ recursion levels and each takes at most $O(k)$ time. This results in $O(nk)$ time for the new phase II compared to $O(n+k)$ for the previous version.

Computing a maximum-cardinality bipartite matching takes $O(k\sqrt{n})$ in practice.

3.2.2 Experiments

We compare our algorithms **Rules**, **EI-1***, and **EI+L3** to the following five other methods that we all implemented in C++.

Anneal is a simulated-annealing algorithm based on the experiments by Christensen et al.. We follow their suggestions for the initial configuration, the objective function, a method for generating configuration changes, and the annealing schedule [CMS95]. In order to save time, we allowed only 30 instead of the proposed 50 temperature stages in the annealing schedule. This did not seem to influence the quality of the results.

Greedy picks repeatedly the leftmost label (i.e. the label whose right edge is leftmost), and discards all candidates that intersect the chosen label. This simple algorithm has an approximation factor of $1/(H+1)$, where H is the ratio of the greatest and the smallest label height [vKSW98]. Greedy can be

implemented to run in $O(n \log n)$ time by using a priority-search tree and a heap, see Remark 3.18 in Section 3.3.2 (page 71). However, our $O(n^2)$-implementation is simply based on lists and uses brute force to find the next leftmost label candidate.

Match refers to the "pure" matching heuristic of Kakoulis and Tollis [KT98]. Our implementation uses the recursive extended BFS of Section 3.2.1 and the maximum-cardinality bipartite-matching algorithm supplied by LEDA [NM90]. It runs in $O(kn)$ time. We did not apply any of our rules and did not do any pre- or post-processing.

Match+L1 is a variant of their algorithm, also proposed in [KT98]. Here rule L1 is applied exhaustively before the heuristic that reduces all connected components to cliques. This does not change the asymptotic runtime behavior.

Hybrid refers to the algorithm sketched in Section 3.2.1. It combines the heuristic by Kakoulis and Tollis that reduces connected components to cliques with our rules L1 to L3. Our implementation uses LEDA's matching algorithm and requires $O(kn)$ time.

We run our algorithm and those described above on the following instance classes. Figures 3.31 to 3.38 depict an example of each of these classes. The first figure in each caption refers to the number of points in the depicted instance. In some figures, an additional number in parenthesis is given, namely the size of the solution of our algorithm Rules applied on the depicted instance. Where Rules found a complete labeling no extra number is given.

RandomRect. We choose n points uniformly distributed in a square of size $25n \times 25n$. To determine the label size for each point, we choose the length of both edges independently under normal distribution, take its absolute value and add 1 to avoid non-positive values. Finally we multiply both label dimensions by 10.

DenseRect. Here we try to place as many rectangles as possible on an area of size $\alpha_1\sqrt{n} \times \alpha_1\sqrt{n}$. α_1 is a factor chosen such that the number of successfully placed rectangles is approximately n, the number of points asked for. We do this by randomly selecting the label size as above and then trying to place the label 50 times. If we don't manage, we select a new label size and repeat the procedure. If none of 20 different sized labels could be placed, we assume that the area is well covered, and stop. For each rectangle we placed successfully, we return its height and width and a corner picked at random. It is clear that all points obtained this way can be labeled by a rectangle of the given size without overlap.

RandomMap and **DenseMap** try to imitate a real map using the same point placement methods as RandomRect and DenseRect, but more realistic label sizes. We assume a distribution of 1:5:25 of cities, towns and villages. After randomly choosing one of these three classes according to the assumed distribution, we set the label height to 12, 10 or 8 points accordingly. The length of the label text then follows the distribution of the German Railway station names (see below). We assume a typewriter font and set the label length to the

number of characters times the font size times 2/3. The multiplicative factor reflects the ratio of character width to height.

VariableDensity. This example class is used in the experimental paper by Christensen et al. [CMS95]. There the points are distributed uniformly on a rectangle of size 792×612. All labels are of equal size, namely 30×7. We included this benchmark for reasons of comparability.

HardGrid. In principle we use the same method as for DenseRect and DenseMap, that is, trying to place as many labels as possible into a given area. In order to do so, we use a grid of $\lfloor \alpha_2 \sqrt{n} \rfloor \times \lceil \alpha_2 \sqrt{n} \rceil$ cells with edge lengths n. Again, α_2 is a factor chosen such that the number of successfully placed squares is approximately n. In a random order, we try to place a square of edge length n into each of the cells. This is done by randomly choosing a point within the cell and putting the lower left corner of the square on it. If it overlaps any of the squares placed before, we repeat at most 10 times before we turn to the next cell.

RegularGrid. We use a grid of $\lfloor \sqrt{n} \rfloor \times \lceil \sqrt{n} \rceil$ squares. For each cell, we randomly choose a corner and place a point with a small constant offset near the chosen corner. Then we know that we can label all points with square labels of the size of a grid cell minus the offset.

MunichDrillholes. The municipal authorities of Munich provided us with the coordinates of roughly 19,400 ground-water drill holes within a 10 by 10 kilometer square centered approximately on the city center. From these points, we randomly pick a center point and then extract a given number of points closest to the center point according to the L_∞–norm. Thus we get a rectangular section of the map. Its size depends on the number of points asked for. The drill-hole labels are abbreviations of fixed length. By scaling the x-coordinates, we make the labels into squares and subsequently apply an exact solver for label-size maximization. This gives us an instance with a maximum number of conflicts which can just be labeled completely.

In addition to these example classes, we tested the algorithms on the point sets depicted in Figures 3.39 and 3.40, see page 58.

3.2.3 Results

We used examples of 250, 500, ..., 3000 points. For each of the example classes and each of the example sizes, we generated 30 files. Then we labeled the points in each file with axis-parallel rectangular labels. We used four label candidates per point, namely those where one of the label's corners is identical to the point. We allowed labels to touch each other but not to obstruct points.

Quality

The graphs in Figures 3.15 to 3.22 (see pages 53 and 54) show the performance of the eight algorithms. The average example size is shown on the x-axis,

the average percentage of labeled points is depicted on the y-axis. Note that we varied the scale on the y-axis from graph to graph in order to show more details. The worst and the best performance of the algorithms are indicated by the lower and upper endpoints of the vertical bars. To improve legibility, we give two graphs for each example class; on the left the results of Rules, EI-1*, EI+L3, Anneal, and Hybrid are depicted, while those of Greedy and the two variants of Match are shown in the graph on the right side of each figure.

The example classes are divided into two groups; those that have a complete labeling and those that have not. For the former group, the percentage of labeled points expresses directly the performance ratio of an algorithm. For examples of the latter group, which consists of RandomRect, RandomMap and VariableDensity, there is only a very weak upper bound for the size of an optimal solution, namely the number of labels needed to fill the area of the bounding box of the instance completely. Thus for VariableDensity at most 2539 points can possibly be labeled. Experiments we performed with an exact solver on examples of up to 200 points showed that on an average about 85% of the points in an instance of RandomRect and usually less than 80% in the case of RandomMap can be labeled. Other than VariableDensity, these classes are designed to keep their properties with increasing point number. This is reflected by the fact that the algorithms' performance was nearly constant on these examples. We used the same set of rules as in phase I of our algorithm to speed up the exact solver.

In terms of performance the algorithms can be divided into two groups. The first group consists of simulated annealing, our rule-based algorithms and the new hybrid algorithm; the second group is represented by the greedy method and the two variants of the matching heuristic. The first group outperforms the second group clearly in all but one example class. On RegularGrid data, the second group and Hybrid achieve 100%, followed very closely by the remaining algorithms; note the scale in Figure 3.20. For all example classes (except RegularGrid and MunichDrillholes, where all algorithms performed extremely well), there is a 5- to 10-percent gap between the results of the two groups.

For all examples that have a complete labeling, Rules, EI+L3, EI-1*, and Hybrid label between 95 and 100% of the points. Experiments on small examples hint that the same holds for larger RandomRect and RandomMap examples. For some of the example classes, simulated annealing outperforms our algorithms by one to two percent. However, in order to achieve similarly good results, simulated annealing needs much longer (see below), in spite of the fact that it uses the same fast $O(n \log n)$ algorithm for detecting rectangle intersections (based on an interval tree). It is not surprising that EI+L3 is better than Rules in most cases; recall that the rules L1 and L2 are special cases of A1 and A2. However, we were astonished to see that Hybrid and Rules yield practically identical results in spite of their different approaches. Only for HardGrid and RegularGrid Hybrid was better than Rules—by merely one percent. The similarity of their results suggests that it is the rules which do most of the work. Rules and EI-1* also yield very similar results; for DenseRect, DenseMap, and

RegularGrid EI-1* is slightly better, Rules on the other example classes. The graph for VariableDensity suggests that EI-1* becomes worse than Rules when the density of the candidate conflict graph increases.

In the second group, the greedy algorithm performed well given that it makes its decisions only based on local information. Surprisingly, its results were practically always better than that of the "pure" Kakoulis-Tollis heuristic that relies on a global matching step. Adding rule L1 as a pre-processing step improved the result of the matching heuristic by up to three percent. This variant performed better than the greedy algorithm in most example classes, but was still clearly worse than simulated annealing and our algorithms except on the rather degenerate RegularGrid data.

Time

In Figures 3.23 to 3.30 (see pages 55 and 56) we present the running times of our implementations in CPU seconds on a Sun UltraSparc. We used the SUN-Pro compiler with optimizer flag `-fast`.

Again, to improve legibility, we give two graphs for each example class; on the left the results of the faster algorithms Rules, EI+L3, Hybrid, and the two variants of Match are depicted, while those of Anneal and Greedy are shown in the graph on the right of each figure. Since EI-1* is only slightly faster than EI+L3, and a difference was only perceivable for RandomMap and VariableDensity, we dropped the graphs for EI-1*.

Given heaps and priority search trees, the greedy algorithm would definitively run faster. Our implementation of simulated annealing seems to be slower by a factor of 2 to 3 than that of Christiansen et al. [CMS95]. This difference in running time may be due to the machines on which the times were measured.

On large examples, Rules is faster by a factor of 2 to 10 as compared to the matching heuristic, and by a factor of 30 to 100 with respect to simulated annealing. Applying rule L1 as a pre-processing step speeds up the matching algorithm up to a factor of a third.

EI+L3 (and thus EI-1*) is slower by a factor of 2 as compared to Rules. This is due to the fact that we did not implement REVISE as in Lemma 2.10 but with the brute-force algorithm sketched at the beginning of Section 2.4.

The fact that some of the algorithms are faster on larger than on smaller point sets of VariableDensity, see Figure 3.30 on page 56, is due to the fact that with the increase in density, many label candidates contain points and are therefore eliminated during preprocessing, see also Figure 3.56 on page 60.

Phase I

Since the difference between the three variants of our algorithms (Rules, EI-1*, and EI+L3) does not show very clearly, we also investigated how efficient they were in phase I, i.e. before applying the heuristic RemoveLocalTroubleMaker-

MinCandNumber. Note that EI-1* is identical to EI-1 before phase II.

The graphs in Figures 3.41 to 3.48 (see page 59) show how many percent of the given points are already labeled at the end of phase I. The graphs in Figures 3.49 to 3.56 (see page 60) show how many percent of the label candidates are removed in phase I. The x-axis in these figures shows the initial number of candidates, which is four times the number of points. Recall that we eliminate all candidates that contain points before phase I. These removals are also counted here.

It is not surprising that EI+L3, whose rules are a superset of those of Rules and EI-1, is always better than both Rules and EI-1. However, it is interesting to see that in most of the graphs EI-1 dominates Rules. EI-1 is more effective in labeling points (in all classes except RandomMap and VariableDensity) and in removing candidates (in all but VariableDensity). In our opinion these graphs support the hypothesis that EI-1 represents a considerable progress compared to Rules in attacking label-placement-type problems. It opens the road for other efficient algorithms that achieve higher degrees of irreducibility and will yield even better results since the need for heuristical decisions will decrease with the gain in terms of irreducibility.

On the example class VariableDensity EI-1 is better for small examples, while Rules does better on the larger and thus denser point sets. This seems to be where rule L3 that is used by Rules but not by EI-1 becomes effective. At the same time, this does not influence the runtime behavior of Rules noticeably, see Figure 3.30, page 56.

Quality of Results I

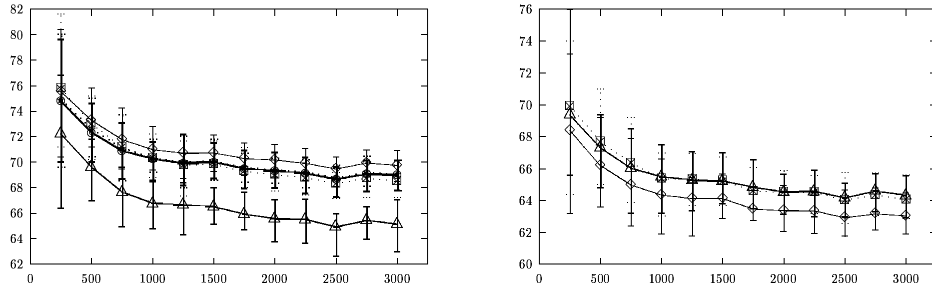


Figure 3.15: RandomMap

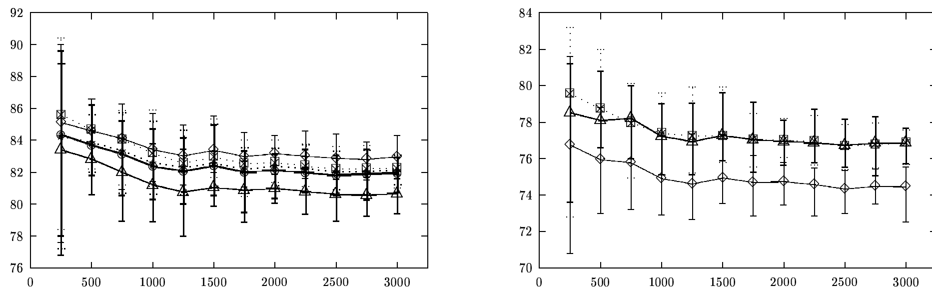


Figure 3.16: RandomRect

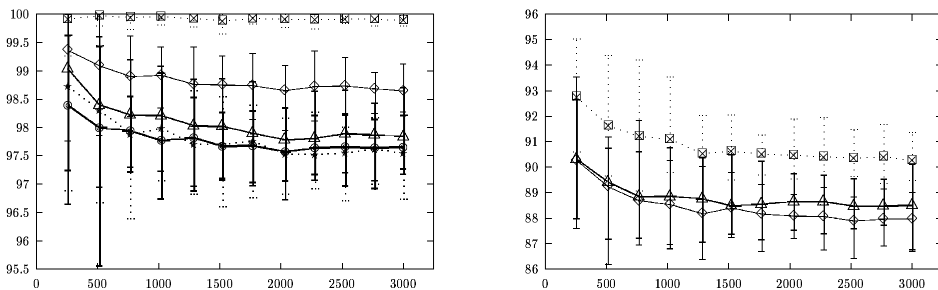


Figure 3.17: DenseMap

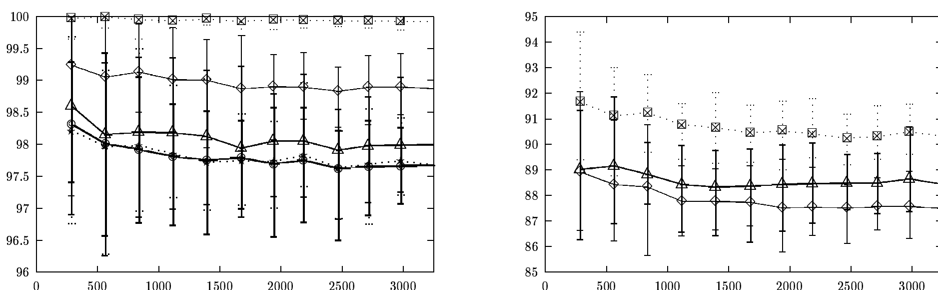


Figure 3.18: DenseRect

Quality of Results II

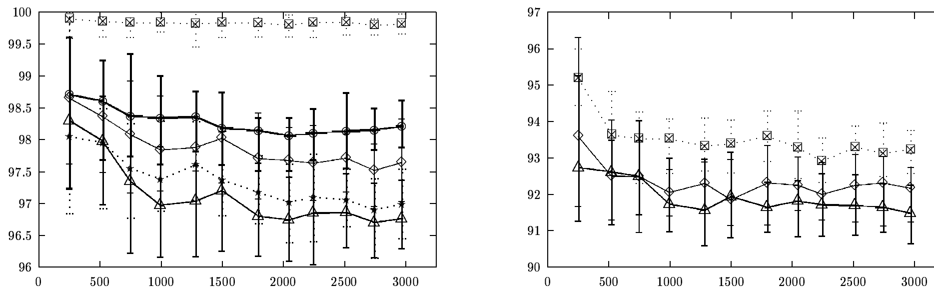


Figure 3.19: HardGrid

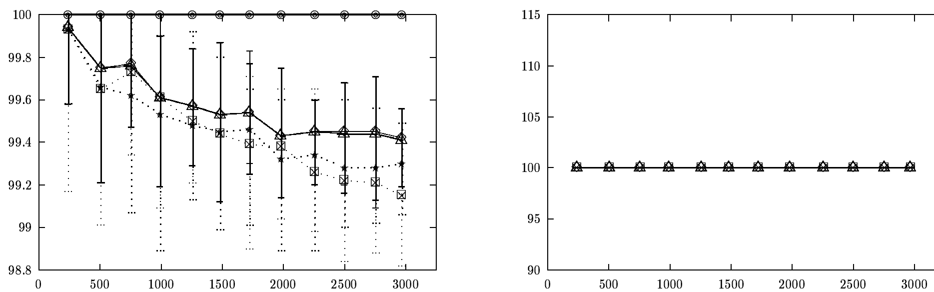


Figure 3.20: RegularGrid

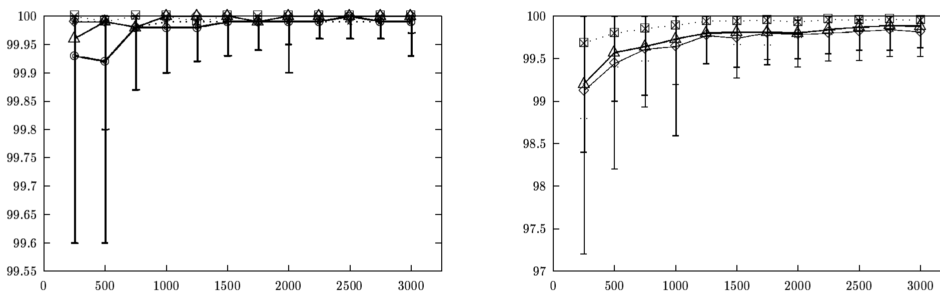


Figure 3.21: MunichDrillholes

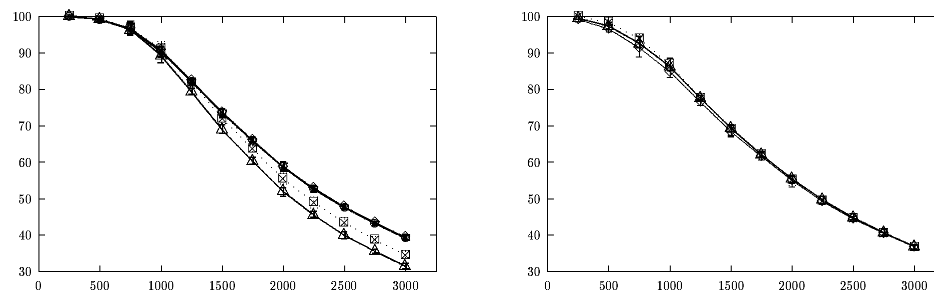


Figure 3.22: VariableDensity

Running Time I

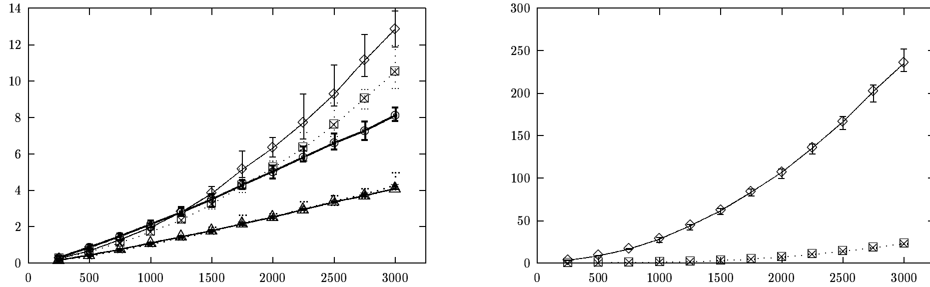


Figure 3.23: RandomMap

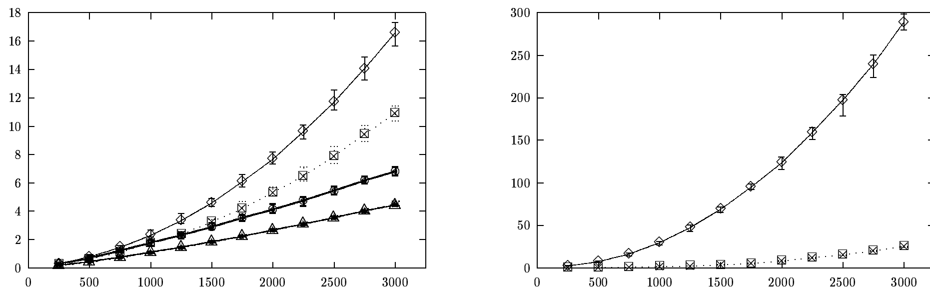


Figure 3.24: RandomRect

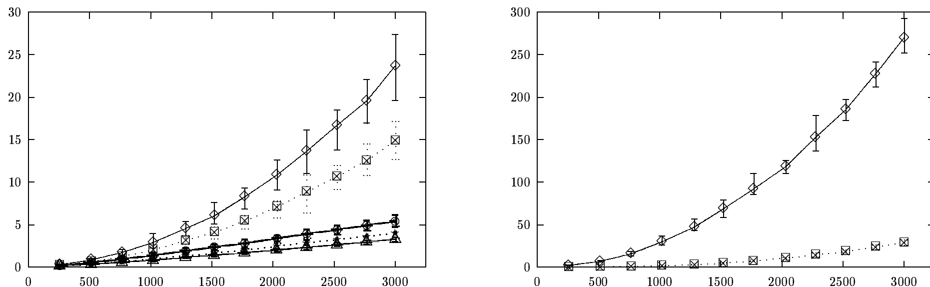


Figure 3.25: DenseMap

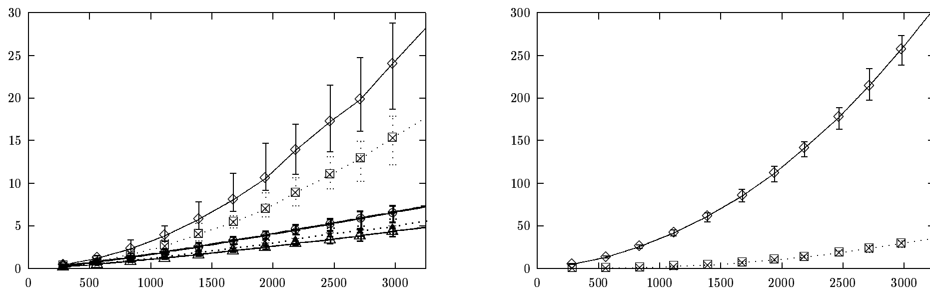
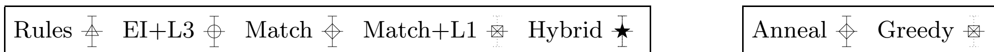


Figure 3.26: DenseRect



Running Time II

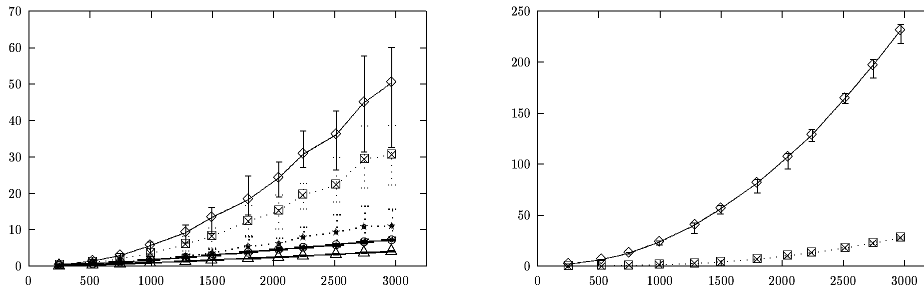


Figure 3.27: HardGrid

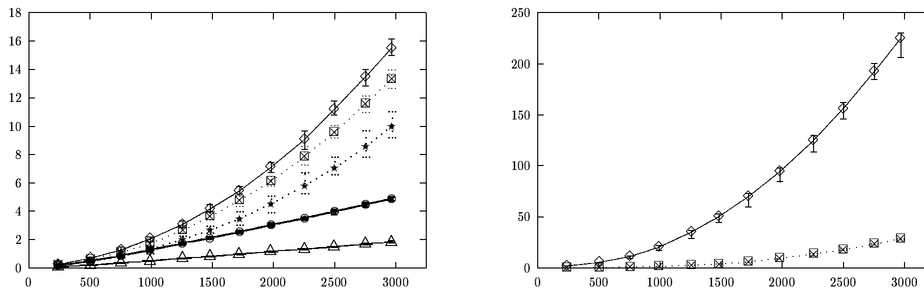


Figure 3.28: RegularGrid

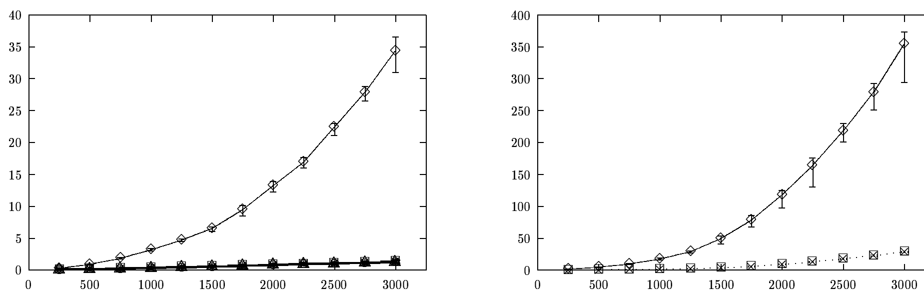


Figure 3.29: MunichDrillholes

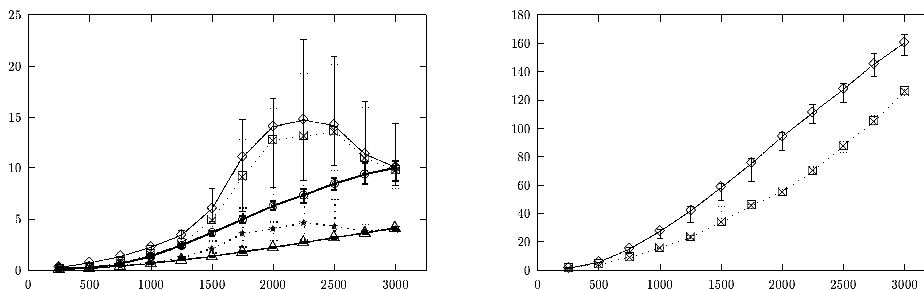
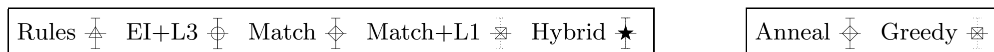


Figure 3.30: VariableDensity



Example Classes

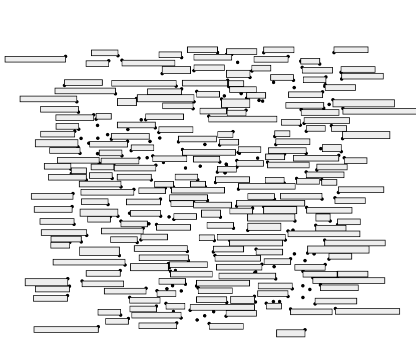


Figure 3.31: RandomMap 250 (193) points

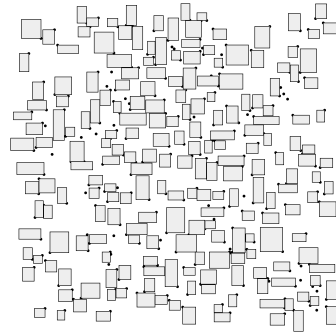


Figure 3.32: RandomRect 250 (212) points

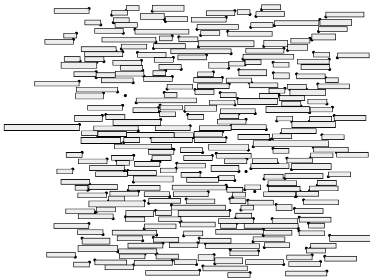


Figure 3.33: DenseMap 253 (249) points

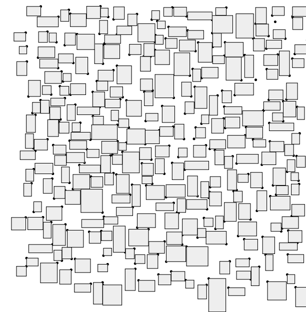


Figure 3.34: DenseRect 261 (258) points

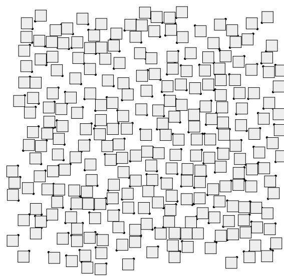


Figure 3.35: HardGrid 253 (252) points

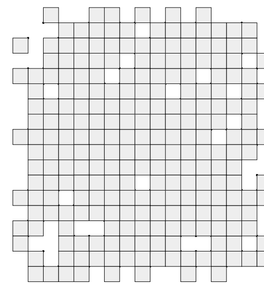


Figure 3.36: RegularGrid 240 points

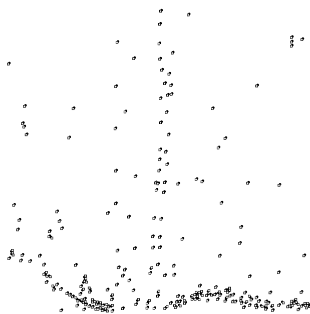


Figure 3.37: MunichDrillholes 250 points

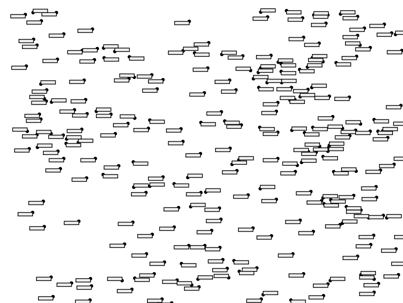


Figure 3.38: VariableDensity 250 points

Two Real-World Examples

Figure 3.39: left: 357 tourist shops in Berlin, right: 45 of 63 labeled.

Figure 3.40: 373 German railway stations, 270 labeled.

Phase I: Number of points labeled

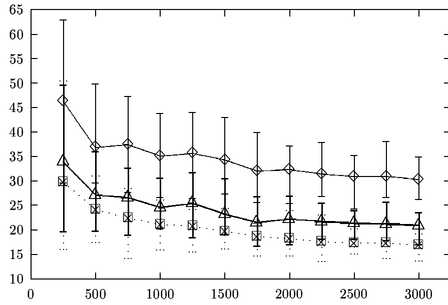


Figure 3.41: RandomMap

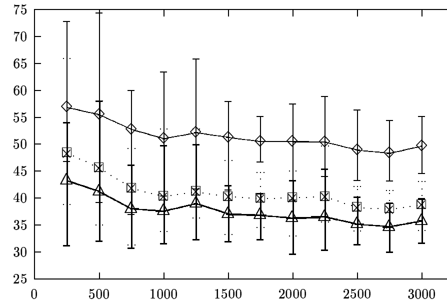


Figure 3.42: RandomRect

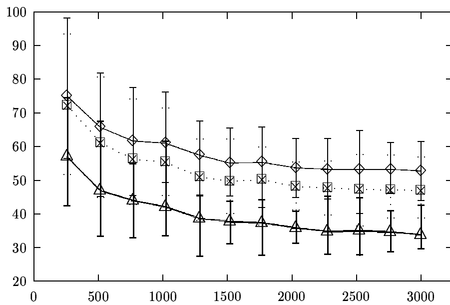


Figure 3.43: DenseMap

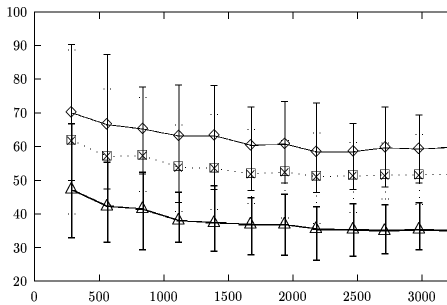


Figure 3.44: DenseRect

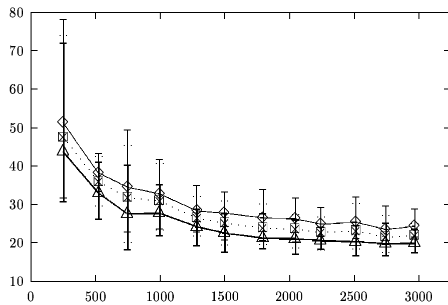


Figure 3.45: HardGrid

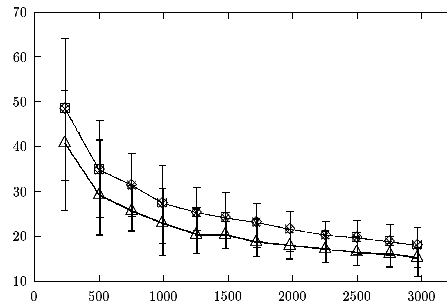


Figure 3.46: RegularGrid

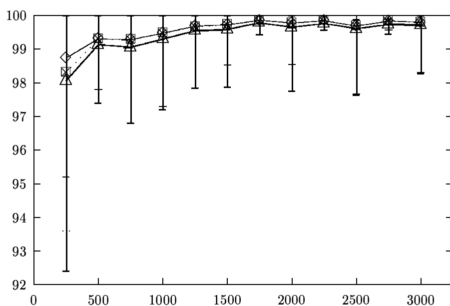


Figure 3.47: MunichDrillholes

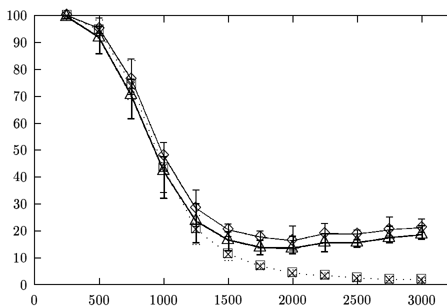


Figure 3.48: VariableDensity

Rules EI-1 EI+L3

Phase I: Number of removed label candidates

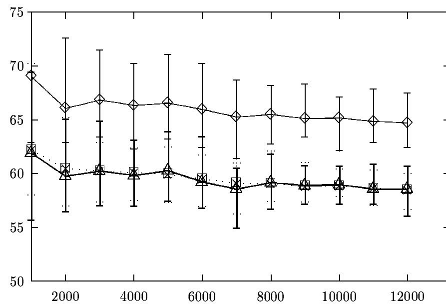


Figure 3.49: RandomMap

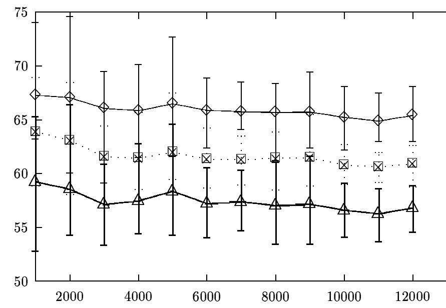


Figure 3.50: RandomRect

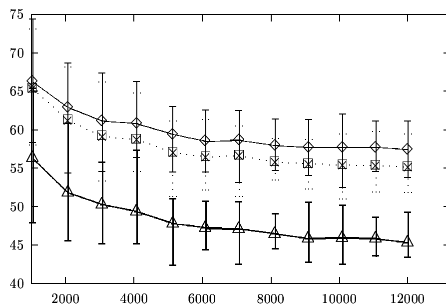


Figure 3.51: DenseMap

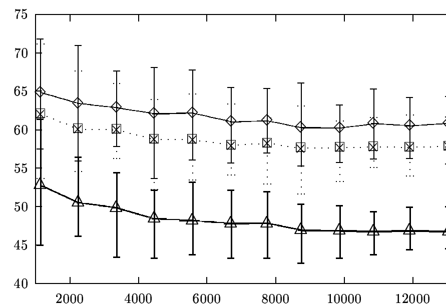


Figure 3.52: DenseRect

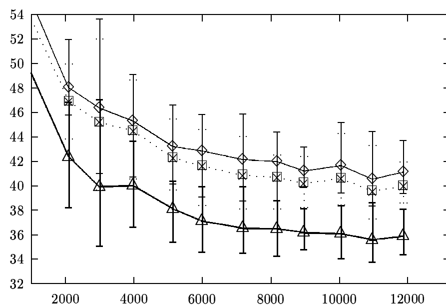


Figure 3.53: HardGrid

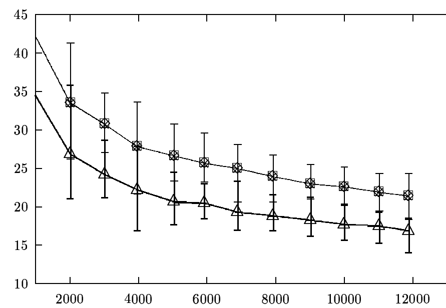


Figure 3.54: RegularGrid

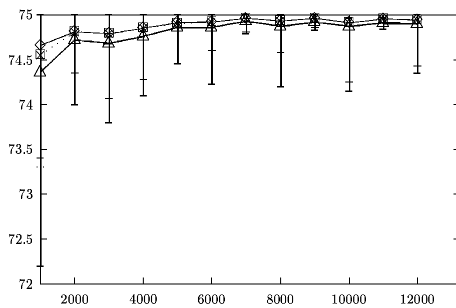


Figure 3.55: MunichDrillholes

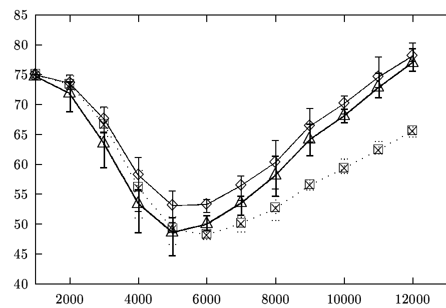


Figure 3.56: VariableDensity

Rules EI-1 EI+L3

3.3 Slider Models

This section is joint work with Marc van Kreveld and Tycho Strijk, both Universiteit Utrecht [vKSW98, vKSW99].

In this section we drop the restriction that a label can only be placed at a finite number of positions. Instead, we allow any position on the edges of the rectangle to coincide with the point, see Figure 3.1. Such a model is called a *slider model*. We will study how many more labels can be placed with slider models than with fixed-position models, and to what extent slider models require more difficult algorithms. We generally assume that labels have equal height but not necessarily the same width. This is a natural assumption if labels contain text or numbers of a fixed font size. We consider the rectangle that represents a label to be closed, which implies that labels are not allowed to touch.

Slider models have been used in two previous papers. Hirsch's paper [Hir82] defines repelling forces for overlapping labels and computes translation vectors for them. After translation, this process is repeated and hopefully, a labeling with few overlaps appears after a number of iterations. This is completely different from our approach, which is combinatorial. The paper by Doddi et al. [DMM+97] contains a number of labeling problems and algorithms, each using a different labeling model. One of the problems is solved in a slider model, where each label is allowed to rotate around the point to be labeled. The labels must be equal-size squares (or other regular polygons); the objective is to maximize the label *size*.

Point labeling has long been known to be NP-complete for fixed-position models [FW91, FPT81, KR92, MS91]. However, this does not imply that label placement is also hard for slider models. In Section 3.3.1 we show that this is the case; we prove that it is NP-complete to decide whether a set of points can be labeled in the four-slider model.

In Section 3.3.2, we show that the slider models allow a simple factor-$\frac{1}{2}$ approximation algorithm that uses $O(n)$ space and $O(n \log n)$ time. This was already known for the fixed-position models [AvKS98]. Our algorithm is greedy in that it always places the label whose right edge is leftmost among the right edges of all possible label placements. The algorithm uses a kind of generalized sweep-line in order to select the next label. We remark that our algorithm can be adapted to labels of varying height, but then the approximation factor depends on the ratio of maximum and minimum label height.

In Section 3.3.3, we give a polynomial time approximation scheme for each of our slider models, showing that for any constant $\epsilon > 0$, there is a polynomial time algorithm that labels a fraction of at least $1 - \epsilon$ of the optimal number of labels that can be placed. Again, this result was already known for fixed-position but not for slider models.

In order to support the practical relevance of the greedy algorithm, we do a thorough experimental analysis in Section 3.3.4. We have implemented our

greedy algorithm for the six models. We test it on three data sets from different application areas. One contains 1000 city names of the USA, another contains a data posting with 236 measurements, and the third contains 75 sequence numbers in a scatter plot near a regression line. We give tables showing how many points are labeled in each model for a range of font sizes. It appears that the greedy algorithm produces about 10–15% more labels for a slider model than in the corresponding fixed-position model. This improvement is significant, because more labels are placed in dense areas. We also compare our algorithm to a simulated annealing algorithm proposed by Christensen et al. [CMS95] on a sequence of randomly generated point sets.

3.3.1 NP-Hardness

The complexity of labeling points with axis-parallel rectangular labels from a *finite* set of label candidates is well established in the literature [FW91, FPT81, IL97, KR92, MS91]. Slider models are a generalization of those fixed-position models that force a label to touch the point to be labeled. However, this observation does not yield the NP-hardness of the slider models, since it is not clear how an instance for a fixed-position model can be reduced to an instance of a slider model. Recall for example that the NP-completeness of 0-1-integer programs does not apply to their relaxation. Therefore we show that placing unit square labels in the 4-slider model is NP-complete.

Theorem 3.15 *It is NP-complete to decide whether a set of points can be labeled with axis-parallel unit squares in the 4-slider model.*

Proof. The problem is in $\mathcal{NP}$ for the following reason. We can guess (i.e. compute non-deterministically) a permutation of the points and an integer between 1 and 4 for each point. This number indicates which edge of a label will be attached to the point. Then we go through the points according to the permutation and check for each point whether we can label it such that its label touches it on the chosen edge—given the labels we have already placed. If the new point can be labeled, we move its label into a *canonical* position: Depending on whether the pre-computed edge is horizontal or vertical, we slide the label along this edge as far left or down as possible. If all points can be labeled this way, we accept. Otherwise we discard the subset. The reason why we can reject in this case is the following. If all points could be labeled, we could push all labels in their canonical positions and name a permutations of the points, such that the procedure outlined above would produce the same canonical label placement.

The proof of the NP-hardness is by reduction from planar 3-SAT. Lichtenstein showed that this restriction of 3-SAT is NP-hard [Lic82]. Our proof follows Knuth and Raghunathan's proof of the NP-hardness of the Metafont-labeling problem [KR92]. We encode the variables and clauses of a Boolean formula ϕ of planar 3-SAT type by a set of points such that all points can only be labeled if ϕ is satisfiable, i.e. if there is a variable assignment such that all

clauses evaluate to true. The advantage of a planar 3-SAT formula is that the variables can always be arranged on a straight line such that they are connected by *non-intersecting* three-legged clauses, see [KR92, Figure 5].

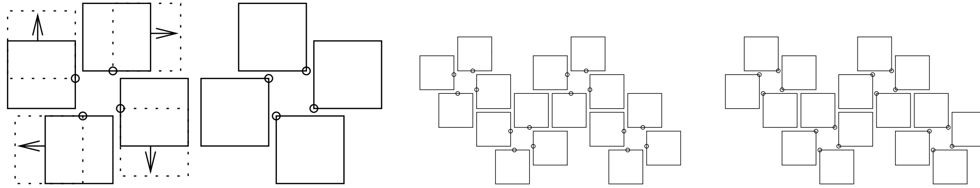


Figure 3.57: Label placements encoding *true* and *false*.

Figure 3.58: Zig-zagging cluster patterns model a variable; the labels its Boolean value.

The main observation leading to our proof is the following. Given a cluster of four points (the corner points of a square with edges slightly longer than 1/2 and rotated by a small angle against the axes), there are two fundamentally different ways to label these points, see Figure 3.57. Under the condition that all points have to be labeled, the points can only be labeled as on the left side (which allows some sliding) or on the right side (where the labels are nearly fixed) of Figure 3.57. Note that is impossible that some points are labeled as on the left and others as on the right side. This gives us a means to encode the Boolean values of a variable in the planar 3-SAT formula ϕ that we want to reduce to a set of points.

The building blocks (or "gadgets") of our reduction are the clusters for variables, three-legged "combs" for clauses, and adapters connecting variables to clauses. In order to be able to connect a variable to all clauses in which it occurs, we model it not by one but by several four-point clusters in a zig-zag pattern as shown in Figure 3.58. Then still all points have to be labeled according to one of the two schemes mentioned above.

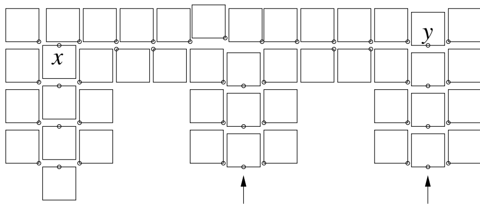


Figure 3.59: Clause with pressure from two variables.

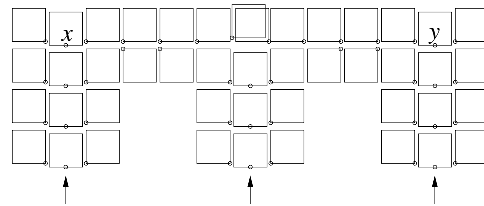


Figure 3.60: Clause with pressure from three variables.

We model the clauses by point sets which resemble large combs with three legs, see Figure 3.59. The fourth column of points from the right and the left can be repeated as often as needed to reach the three variables belonging to the clause. The legs can be extended by duplicating the bottom-most row of points. Each leg is connected to a variable by an adapter. An adapter consists of three points a, b, and c. There are two types of adapters, see Figure 3.61

and 3.62. Which type is chosen depends on whether the variable is negated in the clause. If the variable is set to a value which negates the corresponding literal in the clause, the lowest point b in the adapter must be labeled upwards, i.e. the label sticks into the pipe leading to the clause in question. This forces all other points above b to have their label above them as well. Graphically speaking, pressure is transmitted. This is indicated by arrows in Figure 3.59 to 3.62. When the pressure arrives in the top row of points in the representation of a clause, it is transmitted further horizontally, see the labels of the points x and y in Figure 3.59 and 3.60. Note that a variable assignment which fulfills the corresponding literal does not force anything; no pressure is exerted.

If all literals of a clause evaluate to false, then the points of type b in the adapters of the corresponding variables are labeled upwards and pressure is transmitted through all three vertical pipes into the clause. In this case there is a point which cannot be labeled, see Figure 3.60. If, however, there is at least one vertical pipe without pressure, all points belonging to a clause can be labeled, see Figure 3.59.

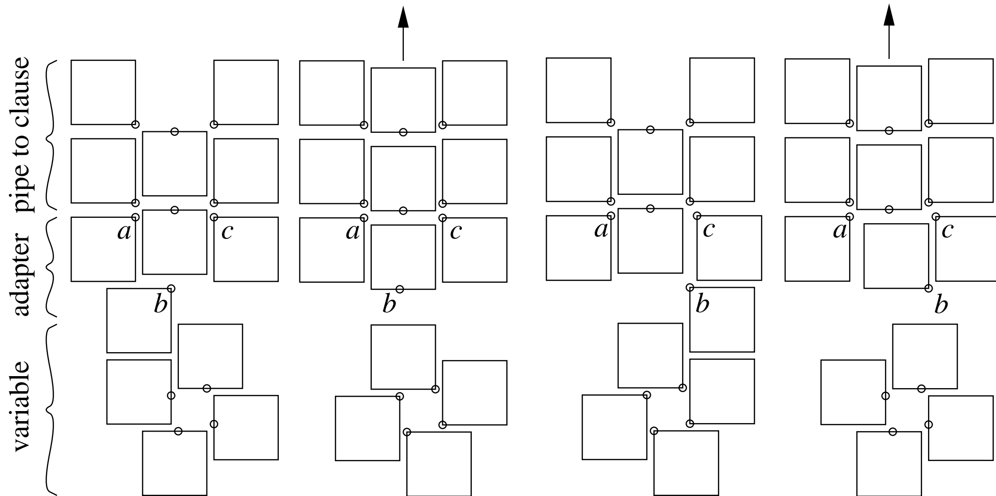


Figure 3.61: Adapters for unnegated literals exert pressure when a variable is set to *false*.

Figure 3.62: Adapters for negated literals exert pressure when a variable is set to *true*.

Hence the question whether ϕ is satisfiable is equivalent to asking whether all points can be labeled in the 4-slider instance to which ϕ is reduced. It is easy to see that the reduction is polynomial: if ϕ consists of m clauses, the instance has $O(m^2)$ points. Their position can certainly be computed in polynomial time. □

3.3.2 A Greedy Approximation Algorithm

In this section we describe algorithms for point feature labeling in the slider models. They apply to labels of fixed height but arbitrary width. We describe an $O(n \log n)$ time algorithm for the slider models that approximates an optimal solution in the following sense. If the maximum number of labels that can be placed is k_{opt}, then our algorithm places at least $k_{\text{opt}}/2$ labels: a factor-$\frac{1}{2}$ approximation algorithm. In most data sets, however, we expect to come much closer to the optimum.

For the fixed position models, a simple $O(n \log n)$ time, factor-$\frac{1}{2}$ approximation algorithm was described recently by Agarwal et al. [AvKS98]. We obtain the same result for the slider models. We'll only describe the most general four-slider algorithm; it is an extension of the top-slider and two-slider algorithms. It is based on a greedy strategy. For convenience we'll first describe the algorithm with labels allowed to touch, unlike in the previous sections where labels were considered to be closed. Later we show that simple adaptations can be made to obtain non-touching labels.

Given a set of points with labels that have already been placed, and a set of points that don't have a label yet, define the *leftmost label* to be the label whose right edge is leftmost among all label candidates of unlabeled points and that does not intersect previously placed labels.

Lemma 3.16 *Given labels of fixed height and any of the slider models, the greedy strategy of repeatedly choosing the leftmost label finds a labeling of at least half the number of points labeled in an optimal solution.*

Proof. Given a set P of points and a sliding model M, let L_{opt} be an optimum M-labeling. Let L_{left} be the set of labels computed by the greedy strategy. The set L_{left} is maximal in the sense that no label can be added to it without intersecting another label in L_{left}. So any label in L_{opt} must either be in L_{left} as well, or intersect some label in L_{left}, whose right edge is at least as much to the left. Charge each label in $L_{\text{opt}} \setminus L_{\text{left}}$ to a label in L_{left} that lies as least as much to the left and intersects it. For any label in $L_{\text{opt}} \cap L_{\text{left}}$, charge it to itself.

We claim that any label in L_{left} is charged at most twice, from which the lemma follows. For labels in $L_{\text{opt}} \cap L_{\text{left}}$ the claim is obviously true. For any other label $l \in L_{\text{left}}$, observe that a label of L_{opt} that charges l must intersect the closed right edge of l. Since all labels have unit height, and the labels in L_{opt} don't intersect each other, there can only be two labels of L_{opt} that intersect the closed right edge, and hence, charge l. ◻

A brute-force algorithm for this simple strategy would need $O(n^3)$ steps. In order to achieve an $O(n \log n)$ time bound, we must use some common geometric data structures.

Let $\{p_1, \ldots, p_n\}$ be the set of points that has to be labeled. The label of

p_i is denoted l_i, and the reference point of a label is its lower left vertex. The possible positions of the reference point of a point p_i are represented by four line segments. Two are horizontal, h_{2i-1} and h_{2i}, and two are vertical, v_{2i-1} and v_{2i}. Their position is exactly the position of the edges of the label l_i if it were placed left and below p_i. The width of l_i is denoted w_i, and the height is 1. We can always scale the y-coordinates to this situation.

If a label l_i has been placed, then no reference point position inside l_i is possible. The same holds for reference points inside the rectangle l'_i precisely one unit below l_i (since any label extends one unit above its reference point). The open rectangle that exactly covers l_i and l'_i and their mutual bounding edge is the *extended rectangle* $\tilde{l}_i$. Since labels are placed from left to right, no reference point positions in nor to the left of $\tilde{l}_i$ will still be accepted by the algorithm. Suppose a subset of the points has already received labels by the algorithm.

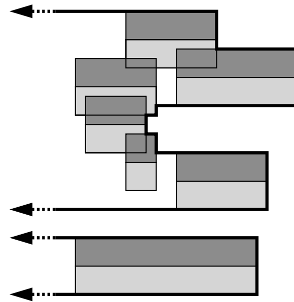


Figure 3.63: Frontier of the placed labels (dark grey) and their lowered copies (light grey).

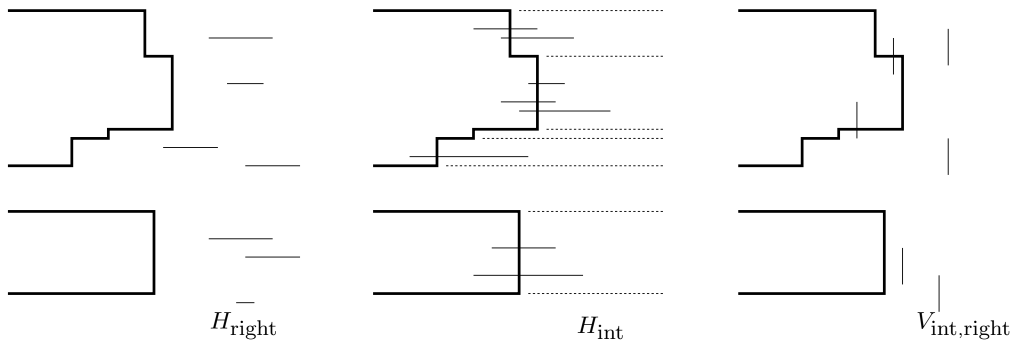


Figure 3.64: The sets H_{right}, H_{int}, and $V_{\text{int,right}}$. The dashed lines in the middle picture separate the segments of H_{int} that are in different red-black trees $\mathcal{T}_i$.

The *right envelope* of all extended rectangles $\tilde{l}$ for all labels l outlines all reference point positions that are impossible, or cannot occur any more, see the bold line in Figure 3.63. We call this right envelope the *frontier* and denote it by F.

To determine the next leftmost label, we only have to consider the frontier F and the segments $h_{2i-1}, h_{2i}, v_{2i-1}$, and v_{2i} of the points p_i to the right of F that don't have a label yet. Given a horizontal segment h and the frontier F, there are three possibilities: (i) h lies completely left of F. Then h can be discarded; a point on it cannot be a reference point for a label that doesn't overlap another label. (ii) h lies completely right of F. Then the leftmost point on h is a candidate for the next leftmost label. (iii) h intersects F. Then a point just right of the intersection point is the candidate. For a vertical segment v, a similar situation occurs. If v lies left of F, it can be discarded; if v lies right of F, any point on v can be chosen; and if v and F intersect, then any point on v right of F can be chosen as a candidate.

Let H be the set of all horizontal segments that represent reference points of the labels. Similarly, let V be the set of the corresponding vertical segments. Let $H_{\text{right}} \subseteq H$ be the subset of all horizontal segments that lie completely right of F, see Figure 3.64. Let $H_{\text{int}} \subseteq H$ be the subset of all horizontal segments that intersect F. Let $H_{\text{left}} \subseteq H$ be the subset of all horizontal segments that lie completely left of F (these cannot give a valid label any more). Let $V_{\text{int,right}} \subseteq V$ be the subset of all vertical segments that contain at least some point right of F.

To maintain the frontier and the candidates for the best reference point efficiently, we need some data structures. Some of the data structures are used to find the next leftmost label; other data structures are only used to update the former ones efficiently. The data structures are red-black trees $\mathcal{T}$, heaps $\mathcal{H}$, and priority search trees $\mathcal{P}$ [McC85]. These are also described in standard textbooks on algorithms [CLR90] and computational geometry [dBvKOS97].

Finding the Leftmost Label

We use three data structures to find the leftmost label position among the ones represented by H_{int}, H_{right}, and $V_{\text{int,right}}$. They are:

1. For each segment in H_{right} we store the x-coordinate of its right endpoint. This corresponds to the right edge of a label whose reference point is the left endpoint of the segment. These values are stored in a heap $\mathcal{H}_{\text{right}}$, where the root stores the minimum.

2. The subset H_{int} is stored as follows. For each vertical segment f_i of F, we maintain a red-black tree $\mathcal{T}_i$ with the segments in H_{int} that intersect f_i (see the middle picture of Figure 3.64). These are stored in the leaves sorted on y-coordinate. With each leaf we also store the width of the corresponding label. We augment each red-black tree by storing at each internal node the minimum width label in the subtree of that node [CLR90]. We use a heap $\mathcal{H}_{\text{int}}$ to have fast access to the segment in H_{int} that allows the leftmost label placement. $\mathcal{H}_{\text{int}}$ stores for each $\mathcal{T}_i$ the sum of the x-coordinate of f_i and the minimum width of the segments in $\mathcal{T}_i$. Thus the root of $\mathcal{H}_{\text{int}}$ corresponds to the leftmost label among the labels represented by H_{int}.

3. For the vertical segments in V, we don't maintain the set $V_{\text{int,right}}$ but some set V' for which $V_{\text{int,right}} \subseteq V' \subseteq V$. The set V' may contain vertical segments that lie completely left of the frontier; these are removed later. The x-coordinate of each segment of V' is stored in a heap $\mathcal{H}_V$. After extracting the minimum from $\mathcal{H}_V$, we test whether it is in $V_{\text{int,right}}$, as described later in **3a**. If not, we discard it and extract the next minimum from the heap, until we find one in $V_{\text{int,right}}$.

We query the three heaps described above. Among their answers, one corresponds to the leftmost label. This is the label we place.

Update assistance structures

After the leftmost label has been determined, we must update the frontier F and several of the data structures described above. This is not so easy. We'll use some more data structures that help to do the updating after the frontier has changed. Let f_{new} be the right edge of the extended rectangle $\tilde{l}$ of the newly placed label l. The new frontier F is the right envelope of the old frontier and f_{new}, see Figure 3.65.

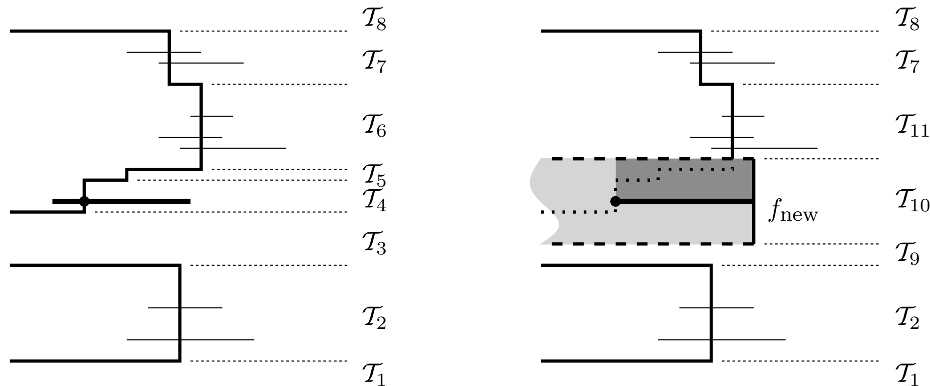


Figure 3.65: When the fat horizontal segment s from H_{int} is chosen, the frontier becomes the right envelope of f_{new} and the old frontier. The new label is dark grey. The grey range (light and dark) is the one with which queries in the priority search trees are done.

1a. To determine which segments move from H_{right} to H_{int} or H_{left} when the frontier changes, we use a priority search tree $\mathcal{P}_{\text{left}}$ on the left endpoints of segments in H_{right}. After placing a label, we query $\mathcal{P}_{\text{left}}$ with the region left of f_{new} (grey in Figure 3.65) to locate the left endpoints of all segments that are no longer in H_{right}. We delete these endpoints from $\mathcal{P}_{\text{left}}$, and we delete the corresponding segments from the heap $\mathcal{H}_{\text{right}}$. For each deleted segment we test whether its right endpoint is right of the frontier. If so, that segment is in H_{int}, and we insert it in the data structures for H_{int}. If not, the segment is in H_{left} and can be discarded.

2a. To determine which segments move from H_{int} to H_{left} when the frontier changes, we use a priority search tree $\mathcal{P}_{\text{right}}$ on the right endpoints of segments in H_{int}. After placing a label, we query $\mathcal{P}_{\text{right}}$ with the region left of f_{new} (grey in Figure 3.65) to locate all right endpoints of segments that have moved from H_{int} to H_{left}. Then we delete the entries corresponding to these segments from the trees $\mathcal{T}_i$, from the heap $\mathcal{H}_{\text{int}}$ and from $\mathcal{P}_{\text{right}}$ itself.

When the frontier changes, we must also reorganize the red-black trees and $\mathcal{H}_{\text{int}}$ as a whole. Recall that we use a red-black tree $\mathcal{T}_i$ for each vertical segment of F. At most three new vertical segments can arise when the frontier changes, but many more vertical segments may cease to exist. We use the trees of the destroyed vertical segments of F to assemble the at most three new red-black trees. This is done by the operations SPLIT and CONCATENATE, which are standard for red-black trees. In Figure 3.65 the trees $\mathcal{T}_3$, $\mathcal{T}_4$, $\mathcal{T}_5$, and $\mathcal{T}_6$ are reorganized to the new trees $\mathcal{T}_9$, $\mathcal{T}_{10}$, and $\mathcal{T}_{11}$. The heap $\mathcal{H}_{\text{int}}$ is updated by removing the value of each destroyed tree, and by inserting the value of each new tree.

3a. Due to the lazy deletion of segments from $\mathcal{H}_V$, we don't need any additional data structures to update the heap on the vertical segments. However, we need to decide whether an extracted minimum from the heap really is in $V_{\text{int,right}}$. We use an augmented red-black tree $\mathcal{T}_V$ for this test. The leaves of this tree store the vertical segments of the frontier sorted from bottom to top. Each leaf also stores the x-coordinate of its segment. Each internal node is augmented with a value that represents the minimum x-coordinate in its subtree. For any query y-interval, a search in $\mathcal{T}_V$ reports the minimum x-coordinate of the frontier in this y-interval.

Algorithm

While there are still segments in any of the heaps $\mathcal{H}_{\text{int}}$, $\mathcal{H}_{\text{right}}$, or $\mathcal{H}_V$, do the following steps:

1. Let v be the vertical segment that corresponds to the minimum of $\mathcal{H}_V$. Search with v in the augmented red-black tree $\mathcal{T}_V$ to see if v has some point right of F. If not, remove v from $\mathcal{H}_V$ and repeat this step.

2. Determine the smallest among the minima of the three heaps $\mathcal{H}_{\text{int}}$, $\mathcal{H}_{\text{right}}$, and $\mathcal{H}_V$. Remove this minimum from its heap. Let l_i be the label position of point p_i corresponding to this minimum. Choose l_i as the next label to be placed.

3. Determine f_{new}, the right edge of the extended rectangle $\tilde{l}_i$. Update the frontier F with f_{new}. Update the augmented red-black tree $\mathcal{T}_V$ (from **3a.**) with f_{new}.

   Search with the region horizontally left of f_{new} (grey in Figure 3.65) in the priority search trees $\mathcal{P}_{\text{left}}$ and $\mathcal{P}_{\text{right}}$ (from **1a** and **2a**) and update the

structures $\mathcal{H}_{\text{right}}$ (from **1**), $\mathcal{P}_{\text{left}}$(from **1a**), $\mathcal{H}_{\text{int}}$ and the $\mathcal{T}_i$'s (from **2**), and $\mathcal{P}_{\text{right}}$ (from **2a**) as explained in the description of these structures.

4. Remove all other reference segments corresponding to p_i from the data structures, in which they occur.

Analysis

The basic structures used by the algorithm are heaps, red-black trees, augmented red-black trees, and priority search trees. All of these structures require $O(n)$ space for a set of size n. Also, these structures can be updated in $O(\log n)$ time per insertion or deletion, or extract-min for heaps. Red-black trees allow Split and Concatenate in $O(\log n)$ time. The queries on the red-black trees take $O(\log n)$ time, and the queries on the priority search trees take $O(k+\log n)$ time, where k is the number of points found in the query range.

The algorithm's runtime of $O(n \log n)$ follows from the following observations. Any vertical segment f_{new} creates one vertical edge in the frontier F, and shortens at most two of them. It follows that throughout the whole algorithm, at most $3n-2$ different vertical edges appear in F. Therefore, at most $3n-2$ vertical edges can be destroyed in the whole algorithm (although many can be destroyed when one vertical segment f_{new} is added to the frontier). This bounds the total number of red-black trees $\mathcal{T}_i$ (from **2**) that can appear, the total number of Split operations, and the total number of Concatenate operations by $O(n)$. Since Split and Concatenate operations take $O(\log n)$ time each, at most $O(n \log n)$ time is spent on splitting and concatenating. The augmented red-black tree $\mathcal{T}_V$ (from **3a**) can also be maintained in $O(n \log n)$ time for the same reasons.

For each new label placed, one query is done on each of the two priority search trees $\mathcal{P}_{\text{left}}$ and $\mathcal{P}_{\text{right}}$. Such a query takes $O(k + \log n)$ time, where k is the number of points in the range. These points are always deleted from the priority search tree, so the algorithm cannot spend time on reporting these points again later in the algorithm. The priority search trees are initialized with one point for each horizontal segment, and we never add more points to them. So in total, at most $O(n \log n)$ time is spent for initializing, querying and updating the priority search trees.

Closed labels

So far we have only discussed the placement of labels that were allowed to touch at the boundaries, that is, the disjoint placement of open rectangles. How can the ideas be adapted to incorporate closed rectangles as labels? Firstly, we let the frontier represent a closed region where reference points of labels cannot lie any more. But the real problem is that we cannot choose and place the *leftmost* label, because this is not well-defined in the slider model with closed rectangles. The solution is to make a distinction between a placement of a rectangle at

some position with x-coordinate $\bar{x}$ and a placement at some position with x-coordinate arbitrarily close to $\bar{x}$, but still strictly to the right of it. Such a distinction can be made by using a symbolic value $\epsilon > 0$ that is arbitrarily close to 0. In case of ties in x-coordinates of labels in the heap, one of them may have been moved symbolically to the right, which resolves the tie. If neither or both labels have been moved symbolically, there is a real tie and we can choose either label as the leftmost. When the algorithm finishes and a set of labels has been selected, then the actual positions of these label can be computed.

We conclude:

Theorem 3.17 *Given n points in the plane, and for each point a rectangular label with fixed height and some given width, then for each of the fixed-position and slider models, there is an $O(n \log n)$ time and linear space algorithm which places at least half the optimal number of labels.*

Remark 3.18 For fixed position models, the algorithm can be implemented using only one priority search tree and one heap. We initialize the priority search tree with the reference points of all label positions. In the heap, we store the sum of x-coordinate and label width for each reference point. When the label corresponding to the heap's minimum is chosen, we query in the priority search tree with the appropriate range to find the reference points that are no longer valid. We remove the entries of these reference points from heap and priority search tree, and repeat by selecting the minimum from the heap.

3.3.3 A Polynomial Time Approximation Scheme

In this section we present schemes for approximating the number of points we can label with unit height labels in all slider models. First we will only consider the top-slider model and then show how these results can be generalized to polynomial time approximation schemes for the two- and four-slider model.

Top-slider model

Given a constant $\varepsilon \in (0, 1)$ we show that there is an algorithm that finds a top-slider labeling of at least $(1 - \varepsilon) \cdot k_{\text{opt}}^{1\text{S}}$ points, where $k_{\text{opt}}^{1\text{S}}$ is the number of labeled points in an optimal top-slider solution. The algorithm has running time $O(n^{4/\varepsilon^2})$.

We use line stabbing to split the problem into smaller units as suggested in [AvKS98]. We stab the unit height labels with horizontal lines of spacing strictly greater than 1 such that each label is stabbed by exactly one line. This can be done in $O(n \log n)$ time [AvKS98] and gives us a partition of the set of input points P into disjoint sets $P_1, \ldots, P_m$, where P_i contains all points whose label intersects the i-th line, and m is the number of stabbing lines.

If we want to obtain an approximation ratio better than 1/2, we cannot afford to discard every second subset P_i of input points. Instead, we have to

look at groups of t consecutive subsets. For $1 \le i \le t+1$, let

$$P^i = P - \bigcup_{j=0}^{\lfloor \frac{m-i}{t+1} \rfloor} P_{i+j\cdot(t+1)}$$

be the set of points that we get from P if we discard every $(t+1)$-st subset starting with P_i. This makes sure that if we compute the optimal solution for t consecutive lines, then we get an approximation for P^i since solutions for its blocks of t lines do not interfere with each other. The pigeon hole principle guarantees that one of the $t+1$ sets of type P^i has an optimal solution of size at least $\frac{t}{t+1} \cdot k_{\rm opt}^{\rm 1S}$. In [AvKS98] this approach was taken, where the optimal solution for the t-lines problem was solved by dynamic programming. In the case of sliding labels one cannot take this approach because the number of candidate label positions in the discretization is superpolynomial. We will still arrive at a polynomial time approximation scheme for the original problem by *approximating* the t-lines subproblem.

Suppose we find a $\frac{k}{k+t-1}$-approximation for the t-line problem, then we can approximate the original problem by a factor of $\gamma = \frac{k}{k+t-1} \cdot \frac{t}{t+1}$, which depends on the two parameters t and k. Setting $k = (t+1)(t-1)$ and $t = \lceil 2/\varepsilon \rceil - 2$ then yields $\gamma = t/(t+2) \ge 1 - \varepsilon$, the desired approximation factor. If the instance needs less than $\lceil 2/\varepsilon \rceil - 2$ stabbing lines, the solution of the problem becomes easier. In this case we set $k = (m-1)(\lceil 1/\varepsilon \rceil - 1)$ and approximate the m-line problem directly with a factor of $\gamma = \frac{k}{k+m-1} \ge 1 - \varepsilon$. The running time would then slightly improve to n^{k+1}. So we can assume $t \le m$ from now on.

It remains to show how we can approximate an optimal solution for t lines by a factor of $\frac{k}{k+t-1}$. The idea is simple and uses the geometrical flavor of the problem. We call a labeling of a set of points *canonical* if all points are labeled and, going through the points from left to right, all labels have been pushed as far left as possible, that is, until they nearly hit another label or have arrived in their leftmost position. (Recall that labels are not allowed to touch each other. As in Section 3.3.2 we treat the distinction between an x-coordinate and a position slightly more to the right symbolically.) Now we just look at *all* canonical label placements of k points. For each such placement we consider the vertical line that goes through the right edge of its rightmost label. We search for the canonical labeling of k points with the leftmost such line $\ell_{\rm left}$, see Figure 3.66. (We have *not* visibly drawn the infinitesimally small spaces between the labels.) We call this placement *leftmost* and compare it to the leftmost k labels of the optimum. Let $\ell_{\rm opt}$ be the vertical line that goes through the k-th leftmost right label edge of the optimal solution, see Figure 3.67. Then we know that $\ell_{\rm left}$ is at least as far to the left as $\ell_{\rm opt}$. We would like to repeat this process with all sets of k points to the right of $\ell_{\rm left}$. We must label them under the restriction that their labels can only be placed to the right of $\ell_{\rm left}$. If we do so, by how much do we get worse than the optimal solution?

By definition $\ell_{\rm opt}$ touches one label of the optimal solution and intersects up to $t-1$ labels on the other $t-1$ lines. Since $\ell_{\rm left}$ is not to the right of $\ell_{\rm opt}$,

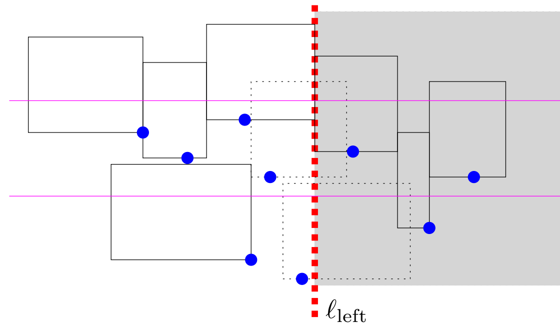


Figure 3.66: Leftmost label placement for a subset of $k = 4$ labels and $t = 2$ lines.

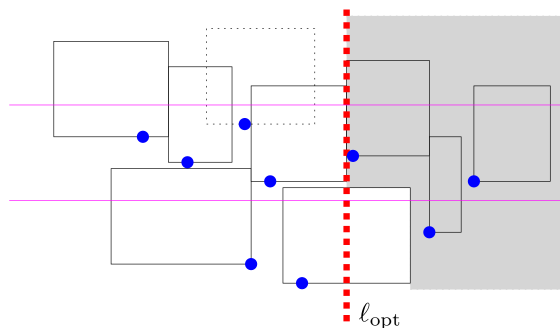


Figure 3.67: An optimal solution for the same points as in the figure above.

the constraint that our leftmost labeling exerts on the next group of k labels is no stronger than the constraint defined by the labels of the optimal solution touching or intersecting ℓ_{opt}, see the gray zones in Figure 3.66 and 3.67. Thus we have placed our first k labels in at most as much 'space' as the first $k+t-1$ labels of the optimal solution. This makes sure that the next line like ℓ_{left}, defined by the next (restricted) leftmost labeling of k points, will again be at most as far to the right as the vertical line through the $(2k+t-1)$-st leftmost right label edge of the optimal solution. By repeating this process until all points are used up, we get a $k/(k+t-1)$-approximation for the number of labeled points in an optimal solution since we always fit k labels in at most as much space as $k+t-1$ labels of the optimal solution. This shows that for the appropriate choice of t and k, we obtain a $(1-\varepsilon)$-approximation for the whole problem.

Let n' be the number of points whose labels intersect a fixed set of t consecutive lines. What is the time we need to compute the first leftmost placement of k out of these n' points? We enumerate all $\binom{n'}{k}$ choices of these k points. For each choice we have to find its canonical labeling—if there is any. Observe that labeling a point p_1 can constrain the labeling of a point p_2 to its left only by not at all allowing to label it. Since we are only interested in subsets of k points that can be labeled completely, it is enough to go through the points

in lexicographical order and try to place each of them leftmost. We can find a label's leftmost position by going through the list of its predecessors once, so finding a canonical labeling can certainly be done in $O(k^2)$ steps. This means that it takes us $O((n')^k)$ steps to compute the first leftmost labeling. Thus we need $T_{t-\text{line}}(n') = \sum_{j=0}^{\lceil n'/k \rceil} O((n'-jk)^k) = O((n')^{k+1})$ time for an approximate solution of the t-line problem. In order to get the total running time T_{total}, we must sum up $T_{t-\text{line}}$ over all possible groups of t consecutive lines. In every group there are at most n points and m, the number of stabbing lines, is at most n as well. Hence $T_{\text{total}}(n) = O(n^{k+2})$. Using $k = (t+1)(t-1)$ and $t = 2/\varepsilon - 2$ as above yields $T_{\text{total}}(n) = O(n^{4/\varepsilon^2})$.

Two and four sliders

The scheme for the top-slider model immediately translates into a polynomial time approximation scheme for the two-slider model. For each point of the input set, we simply place a copy at unit distance below it. (To avoid trouble with an original point at the same place, we can move all copies upwards by an infinitesimal amount.) Then only one point of every such pair is labeled in a top-slider solution. Optimal top-slider solutions for this instance correspond one-to-one to optimal two-slider solutions for the original instance. The running time increases only by a constant factor.

In order to use the ideas given above for the four-slider model, we have to do a little more work. Since labels can now move up and down, the use of stabbing lines is not appropriate any more. Instead, we partition the set of input points into m *strips* of unit height. A strip contains all points between its two bounding horizontal lines and all points that lie on the upper boundary. Similar to our approach above, we will approximate the solution of t consecutive strips. This time, however, we have to drop the points of *two* strips between two blocks to guarantee that solutions of one block do not interfere with solutions of an adjacent block. The pigeon hole principle makes sure that one of the $t+2$ different sets we get by gluing blocks together has at least cardinality $n \cdot \frac{t}{t+2}$. Suppose we have a $\frac{k}{k+t}$-approximation for the t-strip problem, then we could approximate an optimal solution of the whole instance by a factor of $\gamma = \frac{k}{k+t} \cdot \frac{t}{t+2}$. Setting $k = t(t+2)$ and $t = \lceil 3/\varepsilon \rceil - 3$ would then result in $\gamma = t/(t+3) \geq 1 - \varepsilon$, the desired approximation factor.

The additional difficulty in designing an approximation for the t-strip problem is that we do not know on which of its four sides a label in the optimal solution is attached to its point. We can handle this by considering all four possibilities for each of the k points we have chosen. Now we define a canonical labeling as follows. If a label is to be attached to its point on the top or bottom edge, we again push it as far left as possible. If however its point is going to lie on the right or left edge, we push the label as far down as possible. The idea with considering a special order of the points does not work in this setting, so we try to label the k points in every of the $k!$ possible orders, and for every order we check each of the 4^k possible kinds of placement: left, right, bottom,

or top. In this way we can again find a leftmost labeling and a line ℓ_{left}. The constraint that the leftmost labeling exerts on the next group of k labels is at most as strong as the corresponding constraint of the assumed optimal solution. As above, the constraint of the optimal solution is defined by ℓ_{opt} and the labels of the optimal placement intersected by ℓ_{opt}. Apart from the label whose right edge defines ℓ_{opt}, at most t labels can intersect ℓ_{opt} without intersecting each other since their points have to lie within a vertical strip of height strictly less than t (the bottom borderline is excluded). Hence we have a $\frac{k}{k+t}$-approximation for the t-strip problem.

In the approximation algorithm for the four-slider model, we need $\binom{n'}{k}k!4^k k^2$ steps to compute the first leftmost labeling. This still yields an overall running time of $O(n^{4/\varepsilon^2})$.

Theorem 3.19 *For each of the slider models and for any constant $\varepsilon > 0$, there is a polynomial time algorithm which labels at least $(1 - \epsilon)$ times the maximum number of input points that can be labeled.*

3.3.4 Implementation and Experimental Results

The greedy algorithm of Section 3.3.2 has been implemented for the fixed-position and slider models and tested on three real world data sets from different application areas and on a large sequence of randomly generated point sets. In this section we compare experimentally how many labels are placed in each of the six models.

The algorithms were implemented by Tycho Strijk, Universiteit Utrecht, in C++. For the data structures he made use of the LEDA library [NM90]. He simplified the implementation described in Section 3.3.2 in three respects. Firstly, the red-black trees $\mathcal{T}_k$ can be expected to contain only a few horizontal segments at any moment. So he used simple lists for them. Secondly, LEDA does not have an implementation for priority search trees; he used orthogonal range trees instead. Thirdly, the augmented red-black tree $\mathcal{H}_V$ does not profit much from the augmentation in practice. When searching for the minimum x-coordinate of the frontier F in a y-interval, he simply scans all leaves of the red-black tree in that interval. One can expect to visit only a few leaves, since the y-interval is only twice the unit height.

The first of the three data sets contains 1000 cities of the USA that must be labeled with their name. We used several different font sizes, and determined the bounding boxes of the label text. The tables of Figure 3.68 show the results. The codes 1P, 2P, and 4P are shorthand for the 1-, 2-, and 4-position models. The codes 1S, 2S, and 4S are shorthand for the corresponding slider models. The values in the second table show the results in percentages with respect to the 4-position labeling.

The second data set contains the 236 points of a data posting. The labels are measurement values and come from a book on geostatistics [IS89]. Figure 3.69 shows the labeled data set and the number of labels placed in each model.

Number of labels placed

font	model					
	1P	2P	4P	1S	2S	4S
5	851	950	971	990	993	999
6	777	910	952	967	982	986
7	705	852	901	932	964	972
8	686	845	896	918	952	958
9	607	758	817	836	890	902
10	554	704	769	787	853	872
11	520	657	721	735	805	831
12	500	637	709	719	796	813
13	448	570	638	649	716	734
14	433	557	624	637	695	712
15	382	494	550	556	627	645

Percentage w.r.t. 4-position model

font	model					
	1P	2P	4P	1S	2S	4S
5	87	97	100	101	102	102
6	81	95	100	101	103	103
7	78	94	100	103	106	107
8	76	94	100	102	106	106
9	74	92	100	102	108	110
10	72	91	100	102	110	113
11	72	91	100	101	111	115
12	70	89	100	101	112	114
13	70	89	100	101	112	115
14	69	89	100	102	111	114
15	69	89	100	101	114	117

Figure 3.68: One thousand cities on a large map.

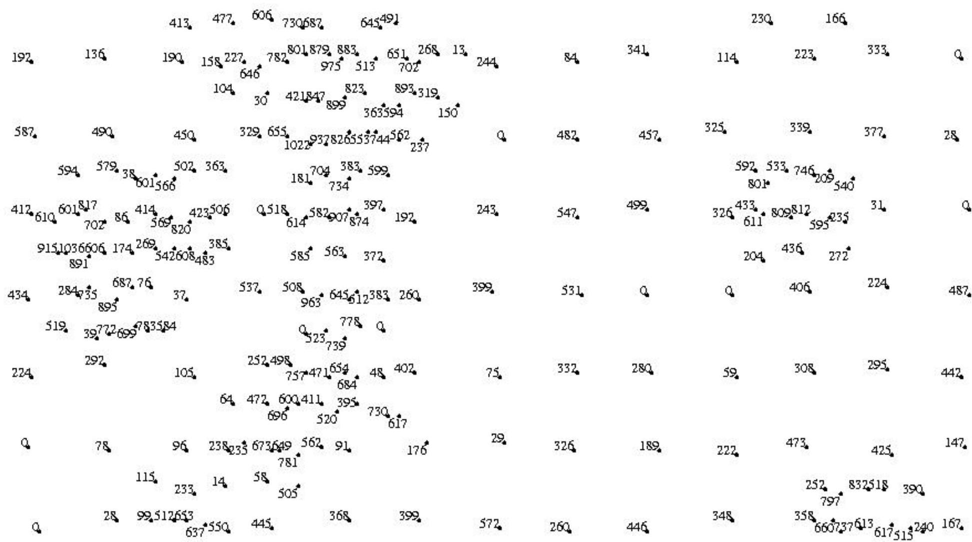
The third data set contains 75 points of a regression analysis. Here the points are clustered near a regression line, and the labels are simply identification numbers. Figure 3.70 shows the labeling.

The bottom tables of Figures 3.69 and 3.70 show that the 4-slider model sometimes places 10–15% more labels than the 4-position model. This improvement is significant, since it is always caused by a better labeling of the areas that are difficult to label. We also created artificial, pseudo-random data sets where all areas are hard to label. These sets were constructed by first placing all points on a grid and after that they were moved randomly a slight distance away from the grid point. Here we indeed found higher improvements: up to 92%.

Efficiency was not the main motivation for these experiments. Still it appeared that the label placement was computed in a few seconds for all data sets we tried, up to 2500 points. A plot shown on a computer screen seldom contains more than 1000 labeled points.

Christensen, Marks and Shieber compared different algorithms using random point sets [CMS95]. Their standard data sets were generated as follows. Inside a grid of size 792 by 612 units, n points were randomly placed and had to be labeled with labels of 30 by 7 units. We considered examples with $n = 100, 250, 500, 750, 1000$, and 1500 points. For each example size, we generated 25 files. We ran the greedy algorithm for each of our six models on all of the generated files. The average percentages of placed labels over the 25 trials are shown in Figure 3.71. Clearly the labeling model has a big influence on the results.

In Figure 3.72 we extend the comparison of Christensen et al. by the results of our algorithm for the four-position and the four-slider model. Our four-position algorithm is always better than gradient descent, and the denser the map the better it gets in relation to gradient descent. For 1500 points it is almost as good as simulated annealing. The four-slider algorithm yields almost equal results as simulated annealing for less than 750 points and is always better beyond 750 points. The running time of our algorithm is generally only a few



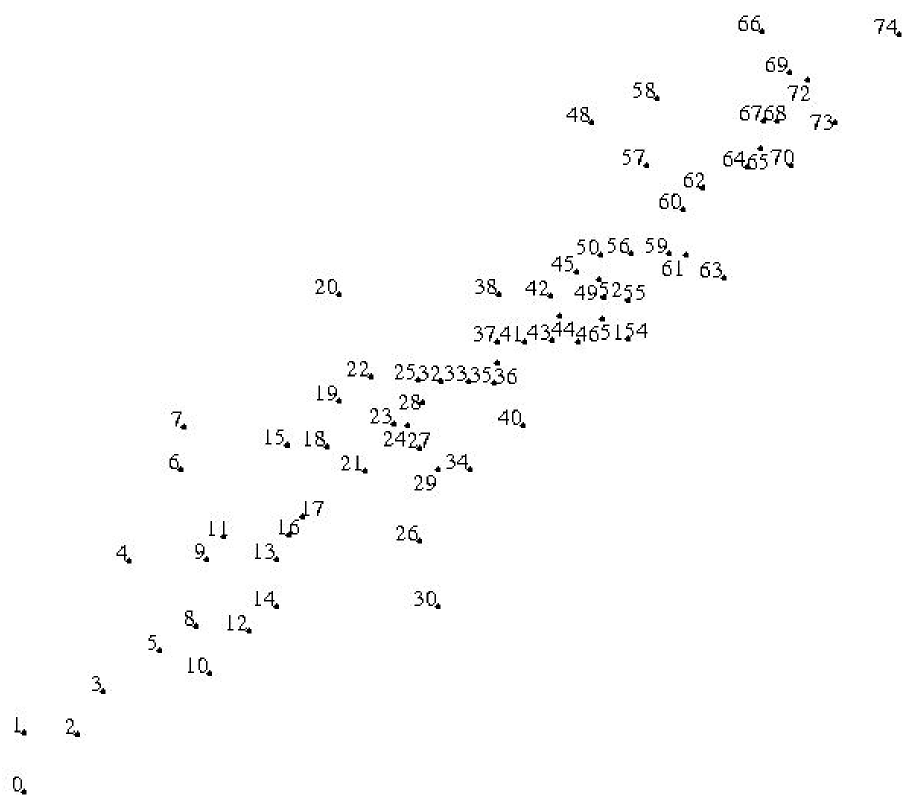
Number of labels placed

font	model 1P	2P	4P	1S	2S	4S
5	229	236	236	236	236	236
6	216	235	235	236	236	236
7	197	219	230	236	236	236
8	197	219	230	236	236	236
9	185	205	218	235	236	236
10	175	193	207	223	231	230
11	174	189	200	213	221	224
12	174	189	200	213	221	224
13	169	180	188	203	212	212
14	169	180	188	203	212	212
15	157	170	176	192	200	203

Percentage w.r.t. 4-position model

font	model 1P	2P	4P	1S	2S	4S
5	97	100	100	100	100	100
6	91	100	100	100	100	100
7	85	95	100	102	102	102
8	85	95	100	102	102	102
9	84	94	100	107	108	108
10	84	93	100	107	111	111
11	87	94	100	106	110	112
12	87	94	100	106	110	112
13	89	95	100	107	112	112
14	89	95	100	107	112	112
15	89	96	100	109	113	115

Figure 3.69: Labeling of the data posting in 9pt font using the 4-slider model (scaled to fit), and tables with the performance.



Number of labels placed

font	model 1P	2P	4P	1S	2S	4S
5	75	75	75	75	75	75
6	75	75	75	75	75	75
7	70	74	74	75	75	75
8	70	74	74	75	75	75
9	60	69	70	73	74	74
10	58	65	68	72	72	72
11	55	61	66	66	70	70
12	55	61	66	66	70	70
13	51	58	64	63	68	71
14	51	58	64	63	68	71
15	50	56	61	62	67	68

Percentage w.r.t. 4-position model

font	model 1P	2P	4P	1S	2S	4S
5	100	100	100	100	100	100
6	100	100	100	100	100	100
7	94	100	100	101	101	101
8	94	100	100	101	101	101
9	85	98	100	104	105	105
10	85	95	100	105	105	105
11	83	92	100	100	106	106
12	83	92	100	100	106	106
13	79	90	100	98	106	110
14	79	90	100	98	106	110
15	81	91	100	101	109	111

Figure 3.70: Labeling of the scatter plot in 11pt font using the 4-slider model (scaled to fit), and tables with the performance.

seconds; even the four-slider algorithm needed just 12 seconds for the largest data sets with 1500 points on a SUN Ultra Sparc. Simulated annealing takes several minutes to label these point sets on the same machine.

Percentage of labels placed

	number of points					
model	100	250	500	750	1000	1500
1P	92.60	84.30	73.16	64.56	57.96	48.58
2P	99.56	97.39	90.24	82.22	74.73	62.75
4P	99.84	99.07	95.45	90.47	83.99	71.74
1S	99.72	98.42	93.80	87.80	81.92	71.04
2S	99.92	99.55	97.83	94.85	90.71	80.75
4S	99.96	99.58	98.02	95.37	91.68	82.68

Figure 3.71: Random data sets (results are averaged over twenty-five trials).

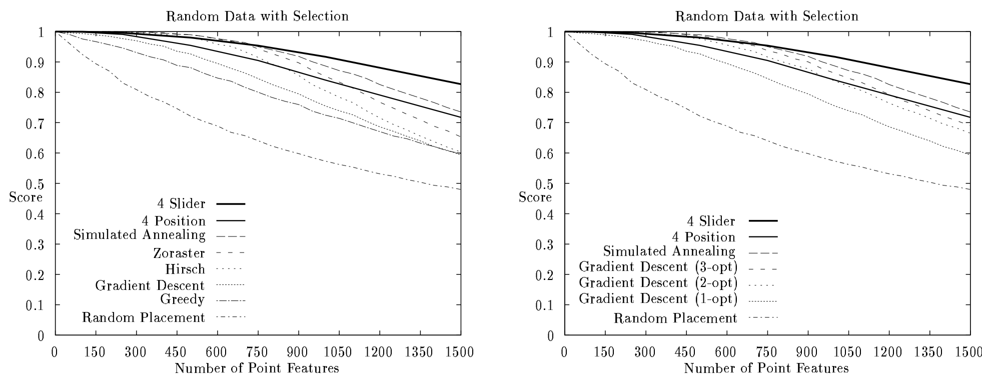


Figure 3.72: Comparison of the four-position and four-slider algorithm to other labeling algorithms.