Conflict Resolving - A Local Search Algorithm for Solving Large Scale Conflict Graphs in Freight Railway Timetabling

Julian Reisch
julian.reisch@synoptics.de
Synoptics GmbH

Peter Großmann
peter.grossmann@synoptics.de
Synoptics GmbH

Daniel Pöhle
daniel.poehle@deutschebahn.com
DB Netz AG

Natalia Kliewer
natalia.kliewer@fu-berlin.de
FU Berlin

July 20, 2020

Abstract

We consider the problem of planning the annual timetable for all freight trains in Germany simultaneously. That is, for each train, construct a slot through the network such that no two slots of different trains have a conflict. We denote this task by the Train Path Assignment Problem (TPAP) and consider a column generation approach where iteratively, we are given for each train request a growing set of possible slots. In each iteration, we look for a maximum subset without any conflicts. We model this problem as the Maximum Independent Set problem (MIS). Due to the many slots that are constructed, hence variables that are generated, we deal with large scale MIS instances. Therefore, we solve the MIS heuristically and develop a local search algorithm called Conflict Resolving (CR) that is tailored to the specially structured instances from the application. To solve the MIS, CR iteratively perturbs the current solution in order to leave local optima and then repeatedly improves the solution by replacing $k-1$ solution vertices by k non-solution vertices. These steps are embedded in a simulated annealing framework. In this paper, we present the column generation approach that is solved as an MIS. Furthermore, we introduce the CR algorithm and numerically compare it to both, a MIP solver and Iterated Local Search (ILS), a state-of-the-art heuristics. It turns out that CR performs best for the instances from real-world timetabling, and is also comparable to the ILS on selected MIS benchmark instances.

1 Introduction

1.1 Motivation

One of the most important planning phases for European railway infrastructure managers is the annual timetabling. All train path applications need to be coordinated within 50 days to yield a conflict-free timetable for the next year. In general, the planning procedure is hierarchical. That is, firstly all passenger trains are planned and secondly, the remaining capacity is used to plan the freight trains. In this paper, we study the problem of assigning a slot in the remaining capacity to as many freight trains as possible.

Speaking of freight trains, a train path application is the request of a railway undertaker who wants to operate a train with specific characteristics from an origin to a destination for a

specific time period within the year. Each train path application has a time flexibility within the day to make the coordination process easier. For example, an undertaker might wish to operate a particular train on every Sunday from Hamburg to Munich between 8 and 9 a.m. The German infrastructure manager DB Netz AG has to coordinate tens of thousands of such freight train path applications in the annual timetable. Today's iterative process of timetabling the trains individually and then resolving the resulting conflicts shall be enhanced by the support of automatic timetabling methods and a better capacity utilization as proposed by Pöhle (2016). In this paper, we present a solution approach that schedules all freight trains in the whole of Germany simultaneously which extends the scales of other approaches significantly.

1.2 The Train Path Assignment Problem for Railway Timetabling

The concrete task is to schedule as many trains as possible within their given time flexibility without violating capacity constraints. A slot is a technically valid path in the space-time graph from the origin to the destination. A slot has a validity of possibly multiple days. A schedule is a set of slots whose validities cover the requested planning period of the train. We denote the task of assigning one schedule to each of as many trains as possible such that no two schedules have a conflict by the Train Path Assignment Problem (TPAP). We give a formal definition in Section 2. Considering the example above, a schedule can consist of two slots on different spatial ways through the network, one for the Sundays in the first half of the year, starting at 8.05 and operating via Cologne, the other one for the second half of the year starting at 8.44 and via Berlin. Together, the validities of the two slots cover the whole planning period of the train so that they form a schedule.

Two schedules are then said to have a conflict if they consist of slots that run on the same day and whose block segments intersect in time. In other words, if they want to use the same infrastructure at the same time, as shown in an example in Figure 1.

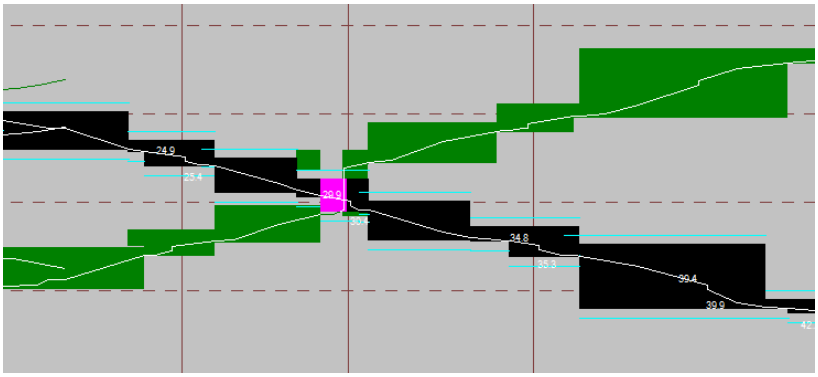

Figure 1: A time-track diagram for a particular day showing two trains (green and black) heading in opposite directions and having a conflict (pink). The diagram is a screenshot taken from the timetabling tool *Rut-K* of the German infrastructure manager DB Netz AG.

In total, there is a huge number of possible combinations that arise when constructing a schedule for a train, namely the number of possible partitions of the requested time period within the year times the number of possible spatial ways through the network times the number of possible departure times during the day for that spatial way. In addition, slots can depart at the same time but run with different speed dynamics or halts. For simplicity, we will focus on the single day planning of slots but we emphasize that all concepts can be adapted to schedules for multiple days.

1.3 Contribution and Outline

The contribution of this paper is twofold. The first contribution is to embed the MIS modelling of the TPAP, introduced by Zwaneveld et al. (1996), in a novel column generation setting that is different to previous column generation approaches for the TPAP in two ways. Firstly, it solves the MIS and not a flow variant of the TPAP and second, it does not work with LP relaxation but a heuristic pricing. This approach will be explained in Section 2. In Section 3, we will explain that the MIP that is solved in the TPAP can be modelled as a Maximum Independent Set Problem (MIS). Then, the second contribution of this paper is to present the MIS heuristic CR in Section 4. CR has been introduced and successfully applied to graphs that encode Maximum Satisfiability Problem instances in Reisch et al. (2019). Here, we study CR with its application in solving the TPAP in detail. The algorithm extends the state-of-the-art heuristic algorithm Iterated Local Search (ILS) by Andrade et al. (2012) by two adaptions. Firstly, instead of simple replacement of a solution vertex by two non-solution vertices, we apply the more complex tight improvements. That is, for each vertex that is not in the current independent set, we compute a tree around this vertex that contains more non-solution vertices than solution vertices. Then, we try to replace all solution vertices in this tree by the same number of formerly non-solution vertices plus the root node. If this step is possible, it improves the solution by one vertex. In order to avoid that we head into a local maximum that we cannot leave, we apply randomized perturbation steps that possibly make the solution worse for some time. The second difference to ILS is that we allow a worsening of the current solution in a simulated annealing fashion. We are provided test data for more than 5000 trains in the whole of Germany for planning a single day and give computational results in Section 5. We will show that CR performs slightly better than ILS and much better than a commercial MIP solver on the larger timetabling instances. This will allow for the conclusion that applying a heuristic for solving the MIS on conflict graphs in the TPAP yields much better results than solving the MIP. An additional improvement can be achieved by applying CR compared to the state-of-the-art heuristic ILS, though this improvement is only slight. In particular, we can find a valid timetable for all freight trains in Germany for one day in reasonable time with our approach which extends the scales of other approaches significantly. Finally, we give a conclusion and outlook in Section 6.

2 Solving the Train Path Assignment Problem

In our application, we want to simultaneously schedule multiple trains on an infrastructure graph by assigning slots on the infrastructure to train path requests. The infrastructure graph is a directed graph where each arc represents a track together with a direction. In this chapter, we state the problem definition and give a literature overview of existing approaches.

2.1 Contents and Notation

Assume that we are given a set of train path requests or trains for short, $\mathcal{T}$. Each train $T \in \mathcal{T}$ consists of an origin, a destination, possibly some intermediate service points, the train characteristics, a desired starting time interval.

Example 2.1. A train path request T wishes to operate from Hamburg to Munich starting between 8 and 9 a.m. with certain characteristics including a locomotive with maximum speed 100 km/h.

A slot for this train is a tuple $y = (\mathbf{x}, \mathbf{t})$ where $\mathbf{x}$ is a vector of tracks the train uses, meeting the demands of service points, and $\mathbf{t}$ is a vector of times the train uses the tracks. A slot can

be found by computing a (shortest) path through the time-expanded infrastructure graph G where each arc (x, y) in G represents a track x (with direction) at a time t. Note that $\mathbf{x}$ and $\mathbf{t}$ denote the vectors whereas x and t the single points. Note further that a path in G can consist only of arcs that are both, successors in the infrastructure graph, and meet the time demands that the succeeding arc can start at a time at least the starting time of the predecessor plus a minimum traveling or halting time. Moreover, a train is specified by its characteristics such as maximum speed, and can therefore take only a subset of tracks. Also, the traveling times on tracks depend on the characteristics. We therefore assume that each train can operate on a subgraph G_T of G. In fact, all information of a train T is contained in G_T so we identify a train with its subgraph.

Let S_T denote the set of all possible slots a train T can operate. Note that the set S_T is merely theoretic since it contains a number of elements that exceeds all computational possibilities. More concretely, there exists one slot for each starting time, for each spatial way from its origin to its destination, for each speed profile on this way, including all possible halts.

Moreover, two slots y_1 and y_2 are said to have a conflict if they occupy the same infrastructure x at the same time t. More precisely, let (x, t) be a pair of a track and a time of a slot y. Then, this track x is blocked for some time $B = [t, t + \delta]$ and all slots containing a pair (x, t') where $t' \in B$ are in conflict to y. Moreover, there might be other arcs (x', t') such as the same track in opposite direction or crossing tracks, that also are in conflict with (x, t). Again, for multiple day planning, two schedules are said to have a conflict of they contain one slot each that together have a conflict. Then, in a generic way, the TPAP reads as follows.

Definition 2.2. Given a set of train path requests $\mathcal{T}$ and a time-expanded graph G, find one slot for each train such that no two slots have a conflict.

2.2 MIP formulation with Slot Variables

For a mathematical formulation of the TPAP, we define decision variables $z_{T,y}$ whether or not a train T uses a slot $y \in S_T$. The set of pairwise conflicts between slots y and y' where $y \in S_T$ and $y' \in S_{T'}$ for all distinct trains T, T' is denoted by $\mathcal{C}$. Then, the TPAP is defined by the following Mixed Integer Program (MIP).

$$\textbf{MIP-}\textit{slot}\ \text{Maximize} \qquad \sum_{T \in \mathcal{T}} \sum_{y \in S_T} z_{T,y} \tag{1a}$$

$$\text{s.t.} \qquad \sum_{y \in S_T} z_{T,y} \leq 1 \qquad \forall T \in \mathcal{T} \tag{1b}$$

$$z_{T,y} + z_{T',y'} \leq 1 \qquad \forall (y, y') \in \mathcal{C} \tag{1c}$$

$$z_{T,y} \in \{0, 1\} \qquad \forall z_{T,y} \tag{1d}$$

The question remains how S_T or at least a suitable subset can be computed. Therefore, we consider the problem of constructing slots in the following chapter. Also, even if S_T was known, this MIP formulation is rather naive and can be improved for computational efficiency, as proposed e.g. by Cacchiani et al. (2010) who separate the constraints by computing maximum independent sets in the so-called comparability graph.

2.3 Slot Construction

Let us now consider the problem of constructing a single slot $s \in S_T$ for a train T. Recall that a slot can be constructed as a path in G_T which already ensures that all train characteristics

are satisfied. However, the time horizon of a slot is a whole day consisting of 86400 seconds which multiplies with the size of the underlying (not time-expanded) graph to the size of G_T making the computation of a (shortest) path intractable. Therefore, Dahms et al. (2019) propose a slot construction heuristic. First, for each track in the infrastructure, the different travel times of T are restricted to a few speeds and possibly halts. These aggregated arcs are called snippets. Unlike arcs in G_T, snippets do not carry a specific point of time but a time interval in which they can start. This yields a much smaller snippet graph SG_T. Then, the authors describe a shortest path heuristic in SG_T that eventually gives rise to a slot y. What is more, this slot construction heuristic can ensure that conflicts to other slots are avoided by cutting (or penalizing) intervals in the snippets wherever another slot is planned already. In particular, two slots that are constructed in this way differ merely in the set of other slots they avoid. This is how different slots can be constructed for S_T. In this paper, we will just speak of this slot construction as an oracle that can be called to find a slot for a train which, in addition, avoid conflicts to other slots in a current solution. Note however, that a slot can still have conflicts to non-solution slots.

2.4 MIP formulation as Multi-Commodity-Flow

We have seen that a slot for T can be computed as a path in G_T. One way to derive a column generation approach for the **MIP-***slot* is to incorporate the slot construction in the MIP formulation. Following Borndörfer and Schlechte (2007) and Caprara (2015), we define the TPAP as a variant of the multi-commodity-flow problem with the adaption that the objective is to satisfy as many commodities as possible. Note that, in addition, one could include a secondary objective to chose shortest paths, but for the sake of simplicity we omit this here.

$$\textbf{MIP-}flow \text{ Maximize} \quad \sum_{T \in \mathcal{T}} q_T \tag{2a}$$

$$\text{s.t.} \quad \mathbf{F}_T y_T = -q_T I_T \qquad \forall T \in \mathcal{T} \tag{2b}$$

$$\mathbf{C}\mathbf{y} \leq \mathbf{1} \tag{2c}$$

$$q_T \in \{0,1\} \qquad \forall T \in \mathcal{T} \tag{2d}$$

$$y_T = (\mathbf{x}_t, \mathbf{t}_T) \qquad \forall T \in \mathcal{T} \tag{2e}$$

$$x \in \{0,1\} \qquad \text{for each track and each } y_T \tag{2f}$$

Let us explain the **MIP-***flow* in detail. Each q_T indicates whether or not a train T is assigned a slot. In (2b), we ensure flow conservation of T if $q_T = 1$. More precisely, F_T is the incidence matrix of the time-expanded graph G_T where a column has a -1 in the row corresponding to the pair (x,t) and $+1$ in the row (x',t') if and only if (x',t') succeeds (x,t) in G_T. Flow conservation is fulfilled if all entries in $\mathbf{F}_T y_T$ are zero with the exception that at one source pair (x,t) it is -1 and at one target pair $+1$. With I_T being the indicator vector that is 1 at all source pairs (x,t) we have flow conservation for T if $q_T = 1$.

Furthermore, the conflict matrix $\mathbf{C}$ is defined to have one row with two entries being 1 for each conflict of possible slots y and y' where $\mathbf{y}$ is the vector of all slots y and $\mathbf{1}$ the vector having all entries 1. Then, (2c) ensures that no two conflicting slots are in a solution of the MIP.

When we assume that slots for a train T can only be constructed as a path in G_T, then **MIP-***slot* and **MIP-***flow* are equivalent. Again, improvements in the MIP formulation have been studied, for instance that the conflict constraints are modelled as set packing constraints in Borndörfer and Schlechte (2007), but improvements in the MIP formulation are out of this papers scope.

2.5 Heuristic Column Generation

We now propose to solve the **MIP-**$slot$ by column generation. In column generation, not all variables are considered right from the beginning, but iteratively, new variables are added to the model. While the overall MIP with all variables present is hard to solve, single slots can be constructed using a (polynomial) shortest path algorithm. Barrett et al. (2000) show that the pricing problem of the multi-commodity-flow problem is a modified shortest path problem. That is, columns for the **MIP-**$flow$ are generated by solving a shortest path problem through G_T. Our approach is to aggregate newly found variables to entire slots and add them as variables to **MIP-**$slot$. More precisely, we do not add a column for each pair (x, t) to the **MIP-**$flow$, but the whole slot y to **MIP-**$slot$. The disadvantage here is that different slots cannot be combined to new slots. On the other hand, we can apply efficient MIS heuristics to **MIP-**$slot$, as will be shown in the following chapter. Moreover and unlike Barrett et al. (2000), we apply the shortest path heuristic by Dahms et al. (2019) instead of an exact shortest path algorithm. We summarize this heuristic approach for solving the TPAP in Algorithm 1. The first step is solving the pricing problem for generating new slots for each train and it is done by the shortest path heuristic by Dahms et al. (2019). The second step is solving the MIS with the already generated variables and we will do this in the numerical experiments in three different ways. By solving the **MIP-**$slot$ with a MIP-solver, the ILS by Andrade et al. (2012) and CR.

Algorithm 1 Train Path Assignment Problem()

Init empty solution I
while *stopping criterion is not met yet* **do**
    Compute a new slot for each train without conflicts to I
    Compute a solution I among all slots found so far
end
return I

A solution I consists of slots y_T for all trains T a subset of trains $\mathcal{T}'$ without any conflicts. We refer to Nachtigall and Opitz (2014) who also solve the TPAP by column generation. In contrast, the authors iteratively add variables with negative reduced costs to the multi-commodity-flow problem. For the lower bounds, they apply a rounding heuristics on a fractional solution because the computation of an integral solution is expensive. The variables with minimum reduced costs are found in the pricing problem. Again, our approach is different as we do not compute reduced costs on the arcs. Instead, we forbid (or penalize) the construction of new slots that are on conflict to the current solution. In other words, we approximate the reduced costs on the arcs by setting them to infinity on arcs that belong to a solution I and zero everywhere else. Recall that forbidding to use arcs twice is possible with the shortest path heuristic presented in Section 2.3.

We emphasize that we cannot guarantee for optimality and might head into local optima as we construct new slots around a current solution. The CR algorithm we present in this paper is designed to solve the second part of Algorithm 1.

2.6 Related Work on TPAP

Let us now give an overview of existing approaches to the TPAP and their scopes. The following approaches have in common that they do not compute the actual slots in the MIP but only on predefined piece-wise slots that have been constructed beforehand. That is, for mesoscopic infrastructure segments, different slots are constructed beforehand and combined to a whole slot by solving the **MIP-**$flow$. The reason for this is that the instance sizes would grow too

much if each arc was considered. For example, Nachtigall and Opitz (2014) compute the reduced costs only on these piece-wise slots. We see the principle drawback that by discretizing the network, we cannot utilize its full capacity.

Borndörfer and Schlechte (2007) apply the multi-commodity-flow model with predefined piece-wise slots successfully to a subnetwork for long distance passenger trains in Germany between Hanover and Fulda. Caprara et al. (2002) have a single track in their scope where trains can meet and Kümmling et al. (2015) apply a similar column generation approach to freight trains in the east of Germany. As the instances become bigger, running time becomes a major issue. There are heuristic approaches proposed in the literature, for example by Schlechte (2012), Fuchsberger and de León (2007) and Furini and Kidd (2013). These works have in common that their application is the timetabling of large but single stations only. In contrast, our focus is the timetabling of the whole of Germany.

3 Maximum Independent Set Formulation of the TPAP

In Section 2, we have seen two MIP formulations of the TPAP, one with slot variables and one as a multi-commodity-flow problem. In this section, we will explain how the first MIP can be solved as a MIS. Then, we discuss the literature on MIS solution algorithms and explain in which respect CR is suited for the application of railway timetabling.

3.1 Transformation to the MIS Problem

The Maximum Independent Set problem (MIS) is the problem of, given an undirected graph, find an independent set of maximum size. An independent set is a subset of vertices such that no two vertices share an edge. We can transform **MIP-***slot* to the MIS in the following way. For each slot y we add a vertex. Two vertices share an edge if and only if they either belong to the same train or they have a conflict. Then, a maximum independent set in this graph, called conflict graph, is a solution to **MIP-***slot*. The approach of solving the TPAP by solving a MIS has first been studied by Zwaneveld et al. (1996), although the authors do not solve the MIS with column generation. Moreover, they define the conflict graph similarly to the graph described above, but two slots differ only in the respect that one is a temporal deviation of the other and not a spatial detour. The reason for that is that their scope is a single station only where they solve the problem with an exact solution algorithm and with instances of sizes at most 3000 vertices.

In Figure 2, we see an example of two iterations of the column generation and the respective conflict graphs. On the left, each of three trains, illustrated by three colors, has a single slot, i.e. $|S_T| = 1$. On the right, the green train has found an additional slot not in conflict to the slots of the other trains. This results in a conflict graph which admits an independent set of size 2, while on the left, a maximum independent set has size 1. Note that this illustration is a simplification as it is a projection to the plane, while in reality, each slot is a trajectory through space and time.

3.2 Related Work on MIS Solution Algorithms

The **MIP-***slot* has a weak LP-relaxation. Therefore, we apply a MIS solution algorithms to the MIS instead. There exists much work on both exact and heuristic solution algorithms for the MIS (or the Maximum Clique Problem). Since we deal with large problem instances and are only interested in near-optimal solutions, let us mention the Iterated Local Search algorithm (ILS) by Andrade et al. (2012) that is known as a state-of-the-art heuristics for the MIS in large

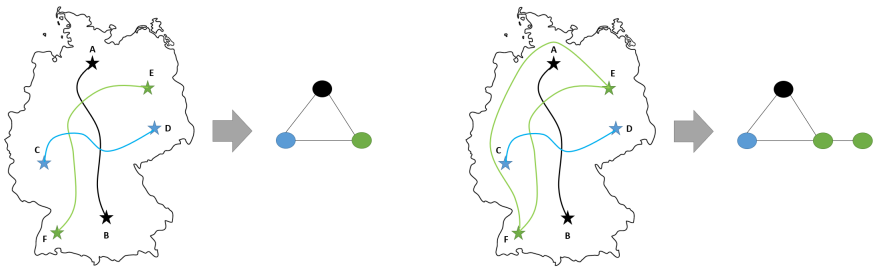

Figure 2: Three slots for three trains and the resulting conflict graph (left) and the green train with an alternative slot (right).

and sparse graphs whereas Rosin (2014) has performed best on benchmarks from Xu (2005). The former works with small but many improvements where one solution vertex is replaced by two whenever it is possible. Repeated perturbation steps ensure that local optima can be left. Our CR algorithm build on the ILS. The latter locally resolves an unfeasible solution until it is feasible. This algorithm is more likely to end up in local maxima.

4 The Conflict Resolving Algorithm

We will now present our main algorithm and prove its correctness. Details concerning parameter setting will be discussed in Section 5. Let us start with some notations in graph theory.

4.1 Notations in Graph Theory

Given an undirected graph $G = (V, E)$ of vertices V and edges $E \subseteq \binom{V}{2}$, an independent set $I \subseteq V$ is a set of vertices I such that $v, w \in I$ implies $\{v, w\} \notin E$, that is, any edge can have at most one of its vertices in I. Accordingly, we call vertices in I solution vertices. By $N(v)$ we denote the neighborhood of v, that is the set of vertices that share a common edge with v. Likewise, we define the k-neighbors of a vertex v by the vertices that can be reached by at most k edges. Let us introduce some definitions for independent sets.

Definition 4.1. With respect to an independent set I, a vertex v is called k-tight if exactly k of its neighbors are in I. A 0-tight vertex is called free.

Definition 4.2. A k-improvement is the replacement of $k-1$ vertices in I by k vertices such that I remains an independent set.

4.2 Overall Procedure

CR iteratively perturbs the current solution and then locally improves the solution by replacing $k-1$ vertices by k vertices wherever it is possible. In order to leave local maxima, we need to apply a perturbation. This comes with the cost that we might worsen the current solution. Therefore, we memorize the best solution so far and possibly restore it after some iterations. The general algorithm is shown in Algorithm 2. There is one input parameter d for the depth of the local search space defined in Section 4.4. Furthermore, there is a maximum number of iterations it_{max} and a counter h for resetting in the simulated annealing part of the solution checking.

Algorithm 2 ConflictResolving($G = (V, E)$, max number of iterations it_{max}, int d)

Init
I,$I_{current}$, $I_{best} = \emptyset$
$h = 0$
$candidates = V \setminus I$
for $it = 1, ..., it_{max}$ **do**
  Perturb(I, $candidates$)
  TightImprovements(I, d, $candidates$)
  CheckNewSolution(I,$I_{current}$, I_{best}, h, it,it_{max})
end
return I_{best}

Until some stopping criterion such as a time[1] or the iteration limit is reached, we repeat the three steps of perturbation, improvement and solution checking.

During the course of the algorithm, we keep and update a set of possible candidates that shall be considered for an improvement. We explain the details in Section 4.4.

We emphasize that the structure of the algorithm follows the ILS by Andrade et al. (2012). The differences are that firstly, we employ the tight improvements instead of merely 2-improvements. We will later show that 2-improvements are also achieved by tight improvements but the reverse does not hold. Secondly, CR has a simulated annealing solution checking whereas ILS only checks for improvements.

4.3 Perturbation

In Algorithm 3, we describe the perturbation step. First, we sample a random number that determines how many vertices are forced into the solution. All neighbors of these vertices have to leave the solution in order to preserve the property of an independent set. We emphasize that after this step, the current solution might have less vertices than before.

Algorithm 3 Perturb(Independent Set I, set $candidates$)

Init b = random number where $\mathbb{P}[b = i] = \frac{1}{2^i}$
v = random* non-solution vertex
while $b > 0$ **do**
  Insert v into I
  Delete $N(v)$ from I
  v = new random* non-solution vertex with distance at most 2 from each formerly picked vertex
  Update $candidates$
  $b = b - 1$
end

(*) Let us explain how a random vertex is picked from a set of vertices V' (the complement of I in V or the vertices of distance at most 2 from the formerly picked vertices). We assume that a ranking function r on the vertices is given. Then, we pick $min\{4, |V'|\}$ many vertices in V' uniformly at random and return the vertex that maximizes r. We chose the ranking function as the *age* of the vertex, that is, the number of iterations since the vertex was in I the last time. This is the same perturbation technique Andrade et al. (2012) use.

[1]Note that a time limit can be used similarly to the limit in terms of number of iterations as it is merely used for a linearly decreasing null sequence in the simulated annealing part.

Finally, we update the candidates by adding new candidate vertices to the end of the candidates list. This is how we ensure that perturbations will not immediately be reverted in a succeeding improvement step. Which vertices are candidates will be explained in the next subsection.

4.4 Tight Improvements

The main ingredient of CR is the step of the tight improvements. We pick a non-solution vertex v that shall be added to the solution. This comes at the cost that the solution neighbors of v have to leave the solution. In order that still, the size of the solution grows, each such solution neighbor of v has to be replaced by one of its non-solution neighbors. As this replacement can be performed recursively, we speak of an alternating tree T.

Definition 4.3. An alternating tree T of depth d with root $v \notin I$ is a tree that is partitioned into a set of solution vertices T_I and non-solution vertices $T_{\neg I}$ with the property that

1. Each vertex in T has distance at most $2d$ from v
2. For a vertex $w \in T$ we have: $w \in I \Leftrightarrow w \in T_I \Leftrightarrow w$ has odd distance to the root v
3. $w \in T_I$ implies that at least one of $N(w)$ is a child of w in T
4. $w \in T_{\neg I}$ implies that all of $N(w) \cap I$ is in T

A child v of w in a rooted tree is defined as a neighbor of w not contained in any path to the root. We denote the children of a vertex w by $c(w)$. We continue by proving some basic lemmata about alternating trees.

Lemma 4.4. *Leaves in an alternating tree are* 1*-tight in the subgraph induced by the tree. In particular, they are not in* I*.*

Proof. Let w be a leaf of T. That is, w has no child and hence by Property 3, $w \notin I$. Since T is alternating, the parent of w is in I. By Property 4, this is the only solution neighbor of w, so w is 1-tight. □

Lemma 4.5. *In an alternating tree* T *we have* $|T_{\neg I}| > |T_I|$*.*

Proof. Consider the mapping c that assigns to each vertex in T_I the set of its children which, since T is alternating, are in $T_{\neg I}$. As T is a tree, we have $c(u) \cap c(w) = \emptyset$ if $u \neq w \in T_I$. Hence, when we choose a vertex from $c(u)$, we can be sure that this vertices is not a child of any other vertex in T. Moreover, by Property 3, $c(u) \neq \emptyset$ for $u \in T_I$ so we can associate to each $u \in T_I$ at least one child in an injective mapping. Together with the root v, there is at least one more vertex in $T_{\neg I}$ than in T_I. □

An alternating tree can be found by a depth-first search similar to finding a spanning tree. Starting from the root, if the current vertex v is not in the solution and has distance less than $2d$ to the root, we add all its solution neighbors not in T to $c(v)$. If the current vertex is in the solution, add all its non-solution neighbors not in T to $c(v)$. By Lemma 4.4, we need to ensure that all leaves are 1-tight. Therefore, we loop over the leaves until all leaves are 1-tight non-solution vertices so that we are left with an alternating tree. If a leaf is in I, we discard the whole subtree rooted at its parent because we already know its parent is not 1-tight. If a leaf is not in I but also not 1-tight, we simply delete it from T and continue with vertices that hence have become leaves.

When an alternating tree is found, then we try to improve the solution by finding an augmenting set.

Definition 4.6. We define the mapping $augment : T_I \to T_{\neg I}$ from the solution vertices in an alternating tree T to the non-solution vertices with the properties

1. $augment(u) \in c(u)$ for all $u \in T_I$
2. $\{augment(u), augment(w)\} \notin E$ for all $u, w \in T_I$
3. $\{augment(u), v\} \notin E$ for all $u \in T_I$

The image $augment(T)$ together with the root v of T is called an augmenting set.

See Figure 3 how an independent set can grow from size 2 to 3 by augmenting in an alternating tree. Note that an alternating tree does not necessarily have an augmenting set.

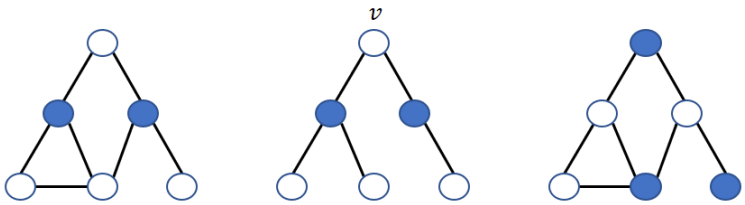

Figure 3: From left to right: An independent set of size 2 (blue), an alternating tree rooted at v and the augmenting set of size 3.

To find an augmenting set in an alternating tree, we replace each solution vertex by one of its children that together form an independent set. Note that this again forces us to compute an independent set which is NP-complete in general. This is why we make the simplification that we will only choose one non-solution child from $c(w)$ of each solution vertex $w \in T_I$. See Figure 4 for an illustration. Furthermore, we can discard all children that are adjacent to the root since the root will be part of the solution in any case.

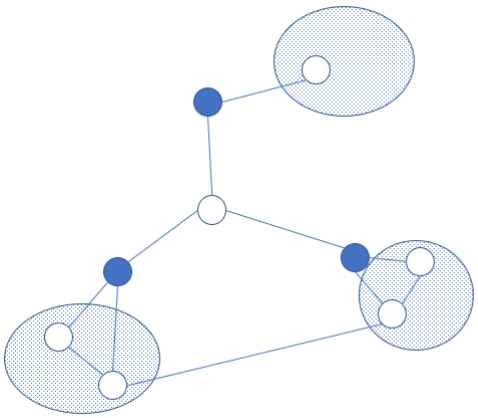

Figure 4: An alternating tree of depth 1: A vertex to be improved (in the middle), its solution neighbors (blue), and their children, grouped in three different cliques. For an augmenting set, choose one vertex from each group by DFS.

We select one vertex from each non-solution children set by a depth-first search. That is, choose one vertex in each set as long as it has no neighbors to previously chosen vertices. If

no such vertex exists, backtrack to the previous children set. Note that this is not an exact MIS algorithm, as it might be the case that we could chose two children of one solution vertex w and no children of another one w' in an augmenting set. In our application, however, we assume that two children of the same parents are likely to belong to the same train, hence same clique. Then, at most one vertex can be part of an independent set anyways. We examine the graphs from the application in greater detail in Section 5.4.

Let us now embed the concept of augmenting sets in the CR algorithm. In Algorithm 4, we see that the solution vertices of alternating trees are replaced by augmenting sets if possible. We say that T gets augmented. By Lemma 4.5, this step improves the solution. The following lemma ensures the correctness of this step.

Lemma 4.7. *The solution vertices of an alternating tree T can be replaced by an augmenting set and I remains an independent set.*

Proof. By construction, an augmenting set is an independent set itself so it suffices to show that there is no edge $e = \{u, w\}$ between a solution vertex u in $I \setminus T$ and a vertex w in the augmenting set. All vertices w in the augmenting set have been non-solution vertices in the alternating tree, hence, by Property 4, $N(w) \cap I \subseteq T$ which concludes the proof. □

Algorithm 4 TightImprovements(Independent Set I, int d, set *candidates*)

```
while candidates ≠ ∅ do
    v = first element in candidates
    T = alternating tree of depth d rooted at v
    if T can be augmented then
        Delete vertices in T from I and from candidates
        Insert augmenting set in I
        Update candidates
    end
end
```

Let us now explain the updating of the candidates. It is not necessary to consider all non-solution vertices in each iteration for a tight improvement because if a vertex' $2d$-neighborhood has not changed in tightness, an improvement is still not possible. Therefore, we only add a vertex to the candidates, both in the perturbation step and here, when it leaves the solution or when it has not been in the solution already and, in addition, there is a change in the tightness of its $2d$-neighborhood.

Moreover, we remark that the tight improvements is the main difference between CR and ILS. ILS applies merely 2-improvements, that is, improve one solution vertex by two non-solution vertices that are not joint by an edge and are both 1-tight. 2-improvements are fast in general, but in theory, the tight improvements from Algorithm 4 yield strictly better results. For the proof, consider a 2-improvement of a solution vertex u by two 1-tight vertices v, w. This improvement can also be achieved by an alternating tree rooted at v with child u and grand-child w. On the other hand, the example of Figure 3 can be improved by an alternating tree but not by a 2-improvement.

4.5 Solution Checking

Depending on the perturbation step, the improvement might not have achieved to make the solution better at the end of one iteration of Algorithm 2. This is permitted for some iterations,

especially at the beginning of the algorithm, since we changed the solution on purpose in order to leave local maxima. However, if we went in a wrong direction for too long, we want to restore the best solution so far which is done in Algorithm 5. In this sense, we can embed our solution checking in the framework of simulated annealing which as first introduced by Kirkpatrick et al. (1983). We allow a worsening only when for more than $|I|$ iterations no better solution has been found.

Algorithm 5 CheckNewSolution(I, $I_{current}$, I_{best}, h, it, it_{max})

```
if |I| ≥ |I_current| then
    I_current = I and h = 0
    if |I| ≥ |I_best| then
        I_best = I
    end
end
else
    if h ≥ |I| then
        δ = |I_best| − |I| and ω = exp(−δ it/it_max)
        Toss a coin with probability ω to show face
        if the coin shows face then
            I_current = I
        end
    end
    h = h + 1
end
```

We conclude this section with the following theorem about the correctness of CR.

Theorem 4.8. *The Conflict Resolving Algorithm yields an independent set.*

Proof. The perturbations ensure that the neighborhood of each vertex forced into the solution leaves the solution. By Lemma 4.7, the tight improvements keep I as an independent set. The solution checking just restores formerly found independent sets. □

5 Experimental Results

We apply CR to instances that are generated from real-world timetabling problems, as well as on sample graphs from two different MIS challenges, all described in Section 5.1. We compare the results to the commercial MIP solver Gurobi and the Iterated Local Search (ILS) by Andrade et al. (2012) as explained in Section 5.2. Furthermore, we provide a parameter analysis for the tree-depth of the alternating trees in Section 5.3, a graph analysis that shows the merits of CR with respect to the timetabling conflict graphs in Section 5.4 and a running time analysis in 5.5. Finally, we present and discuss the results in Sections 5.6 and 5.7.

5.1 Test Instances

We are provided real-world data for 5359 freight trains in the whole of Germany. As a benchmark, we run Algorithm 1 for 6 hours and apply Gurobi for the MIP solving with a timeout of 35 minutes per iteration. In iteration l, we export the conflict graph, denoted SingleDay$_l$ for $l \in \{2, 3, 12, 15, 37\}$. As in each iteration, the current solution of the previous iteration is the set of forbidden slots so that for each train, the slot construction oracle has to return either a new

slot not in conflict to this set or returns no slot at all. Furthermore, we are provided a smaller set of long distance freight trains only and similarly produce conflict graphs Subnetwork$_l$.

The focus of our algorithm is on solving timetabling problems. Still, for a more extensive evaluation of the algorithm, we include a set of instances from the MIS challenges on *Graphs From Codes For Correcting One Error on the Z-Channel* in Sloane (2015) and on *Benchmarks with Hidden Optimum Solutions for Graph Problems* in Xu (2005).

5.2 Comparison to other Solvers

We compare the results of CR to Gurobi[2]. Before starting the computation, we performed an exhaustive parameter tuning on the railway timetabling instances. The resulted parameters are the following: Method = 2, MIPFocus = 1, Presolve = 2, FlowCoverCuts = 2 and CliqueCuts = 2. Note that we set these parameters for all our computations.

Furthermore, we use the following MIP encoding of the MIS: $max \sum_{v \in V} x_v$ subject to $x_v + x_w \leq 1$ for conflicting slots $\{v, w\} \in E$ and $\sum_{v \in C} x_v \leq 1$ for slots v that belong to the same train C. All x_v are binary variables. As stated in Section 2.2, we are aware that there are improvements to this MIP formulation. Nevertheless, we assume that the clique cuts we force Gurobi to apply, will enhance the performance regardless of the weak formulation.

Also, we implemented the ILS as formulated by Andrade et al. (2012) in order to compare the results on the railway timetabling instances. Our own implementation will be marked as ILS$^+$.

5.3 Alternating Tree-depth Parameter

In the course of the CR algorithm, we repeatedly compute alternating trees. We see in Figure 5 two example subgraphs where alternating trees of depth 2 will be computed, that is, the 3-neighborhoods of a root (green). Both graphs are taken from the largest timetabling instance and are representative with respect to the following property. In both subgraphs, no augmenting step is possible as there is one solution vertex (black) without any 1-tight children not adjacent to the root. In general, we observe that the deeper the tree, the more likely it is that there exists a solution vertex (black) that does not have any 1-tight children which are not adjacent to the root. Such solution vertices prevent an augmenting step.

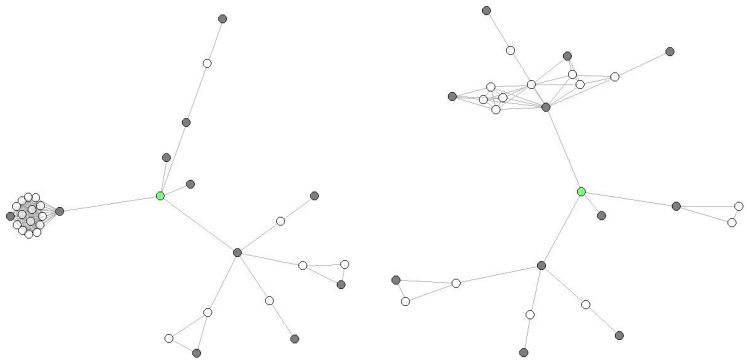

Figure 5: Subgraphs where alternating trees are computed with root (green) and with respect to a maximal solution of the MIS (black).

[2]Gurobi version 8.1.0., http://www.gurobi.com

We conclude that working with trees of depth 1 not only is computationally less complex, it is also more likely that an augmenting step is possible. This is why we fix the tree depth parameter to 1, although for other applications, deeper tree might give better results.

5.4 Graph Analysis

In Section 4, we argued that CR is more suitable for conflict graphs than ILS. In Figures 6 and 7, all taken from the largest timetabling instance, we see on the left the entire 2-neighborhood of the root of an alternating tree. We observe that they fall into cliques that presumably belong to different trains. If we computed an independent set in this subgraph, it would take a lot of computation time since these subgraphs contain many vertices and are relatively dense. On the right, all 2-neighbors whose tightness is greater than 1 or that have an edge to the root are deleted. Recall that an augmenting set is computed with a depth-first search in these subgraphs only. The computation runs in $\mathcal{O}(M)$ where M is the product of the children's sets' sizes. Often, these sets have size zero and no improvement can be made, or 1 and M does not grow. The example subgraphs are representative in the sense that only in the subgraph in Figure 6, there is a solution vertex (top right) with more than one child to choose from. All the other instances and all solution vertices provide at most one non-solution child. Hence, the depth-first search can be performed fast.

Figure 6: On the left: 2-neighborhood of the root; On the right: The subgraph where an augmenting set is computed.

On the other hand, in Figure 8, we see the same restricted subgraphs where only 1-tight children not adjacent to the root are displayed, but taken from the benchmark instance 1zc.4096. We do not show the complete 2-neighborhood here since these are too dense to visualize. Note that the average degree of the graph is 286. We see that unlike in the timetabling instances, these restricted graphs still contain many candidate vertices for an augmenting set. What is more, some non-solution vertices can be assigned to two different parents. Note that in the definition of alternating trees, this is forbidden. Hence, we might end up choosing the wrong parent for some child meaning that with another parent, we would have found an augmenting set. As in many benchmark graphs, such subgraphs occur often, we assume the CR algorithm not to perform very well on these instances, compared to the timetabling instances.

Figure 7: On the left: 2-neighborhood of the root; On the right: The subgraph where an augmenting set is computed.

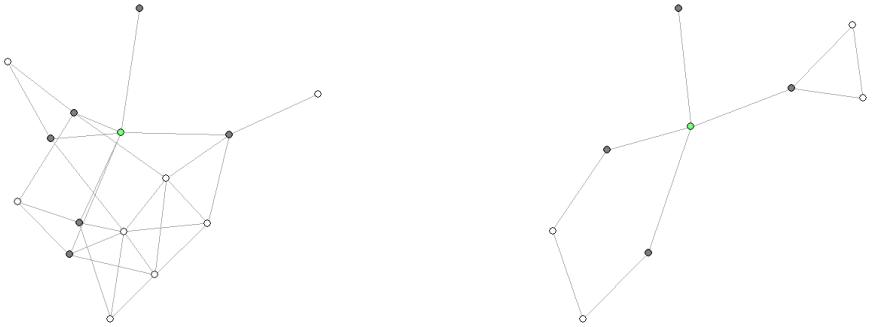

Figure 8: Subgraphs where an augmenting set will be computed in the benchmark graphs.

5.5 Objective Function Evolution in Single Runs

In Figure 9, we see the evolution of the objective function value in three runs of CR and ILS on the largest timetabling instance. First of all, we see that CR converges much faster than ILS so that even a smaller timeout than 2 minutes would have sufficed for the results. Secondly, we see more decreases in ILS indicating that the 2-improvements do not compensate the perturbation, especially in the beginning. In CR, however, the perturbations can often be compensated within the same iteration as the objective rarely decreases. Only in the last third of the running time, when ILS almost stagnates already, CR sometimes decreases which we explain by the simulated annealing solution checking that allows for a worsening.

5.6 Results

See the results of the computations of independent sets for the timetabling instances in Table 1 and the results on the benchmark graphs in Table 2. For the benchmarks results of the ILS, we refer the the paper by Grosso et al. (2008). In the light of the application in railway timetabling, we set a time limit of 2 minutes as explained in Section 2. We choose this rather short time limit in order to achieve more iterations in the process of Algorithm 1 and hence, generate

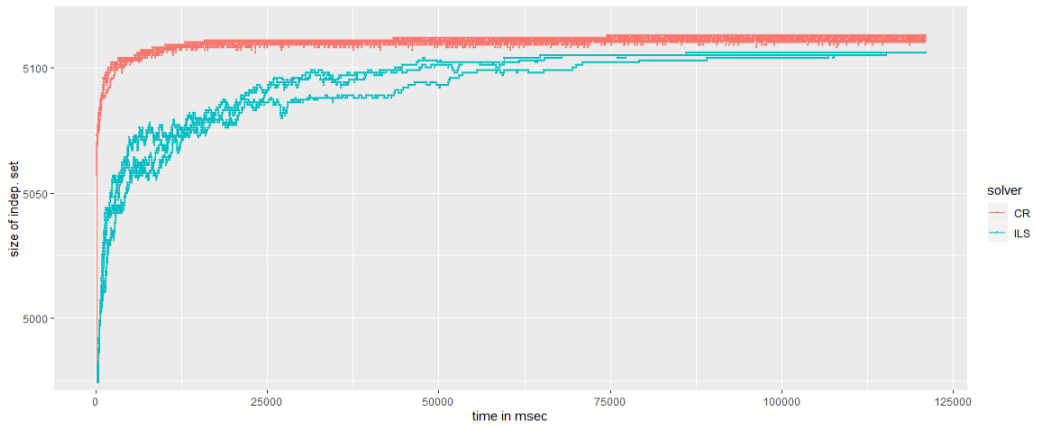

Figure 9: Running time versus Objective Function Value for CR and ILS.

more columns. Note that Andrade et al. do not set a time limit, but a limit of arc scans, so we give CR the same amount of time the ILS took to reach its limit of arc scans.

We started five runs for each instance with the randomized algorithms CR and ILS$^+$ and show the average here. All of the computations where performed on an Intel(R) Core(TM) i7-8700K.

For the CR algorithm we chose the tree depth d always as 1. We state the number of trains, that is, number of cliques in the graph, and the number of conflicts, that is, the number of edges between different cliques, in Table 1, as well. The lower and upper bounds either come from Gurobi or from the best solution we could find with larger timeouts by applying CR.

Table 1: Computational Results on the timetabling instances within 2 minutes.

Graph Instance	ILS$^+$	CR	Gurobi	Number of trains	Number of Vertices	Number of Conflicts	Lower bound	Upper bound
Subnetwork_7	**2054**	**2054**	**2054**	2327	10736	83937	2054	2065
Subnetwork_9	**2126**	**2126**	2125	2327	12317	127646	2125	2140
SingleDay_2	4392.6	**4394**	**4394**	5359	10655	41911	4394	4394
SingleDay_3	4726	**4728**	4727	5359	14175	86770	4727	4735
SingleDay_{12}	5083	**5091**	4253	5359	39426	1851550	5090	5107
SingleDay_{15}	5088	**5101**	4257	5359	45761	2172902	5101	5118
SingleDay_{37}	5109	**5113**	4274	5359	87039	11012974	5113	5359

Table 2: Computational Results on the benchmark graphs.

Graph Instance	ILS	CR	Time	Number of Vertices	Density	Lower bound	Upper bound
frb30-15-1	**30**	**30**	20	450	0.176	30	30
frb35-17-1	**35**	34	30	595	0.158	35	35
frb40-19-1	**40**	39.3	42	760	0.143	40	40
frb45-21-1	**44.6**	43.3	65	945	0.133	45	45
frb50-23-1	**49.1**	48.2	86	1150	0.121	50	50
frb53-24-1	**51.6**	51	98	1272	0.117	53	53
frb56-25-1	**54.1**	54	118	1400	0.112	56	56
frb59-26-1	**57.2**	56.2	137	1534	0.108	59	59
1dc.1024	**93.2**	93	31	1024	0.046	94	96
1zc.1024	111.3	**112**	23	1024	0.064	112	117
1zc.2048	**197.4**	196.7	53	2048	0.038	198	269
1zc.4096	**370.7**	368.8	127	4096	0.022	379	410

Finally, Figure 10 shows how the MIS Solver CR performs in comparison to Gurobi during the course of a full run of Algorithm 1 with a time limit of 6 hours. That is, in each solving step of the second procedure in the while-loop, we employ CR instead of Gurobi. The former has a time limit of 2, the latter 35 minutes because this is the amount of time the two solvers yield comparable results per iteration. In fact, we see in Figure 1 that Gurobi has no practicable solution after 2 minutes. The construction of new slots is done by the same oracle that avoids conflicts to the solution of the previous iteration. Thus, this first part of Algorithm 1 of constructing new slots, that is, finding new variables, takes the same time regardless of the MIS solver, namely about 30-40 minutes per iteration.

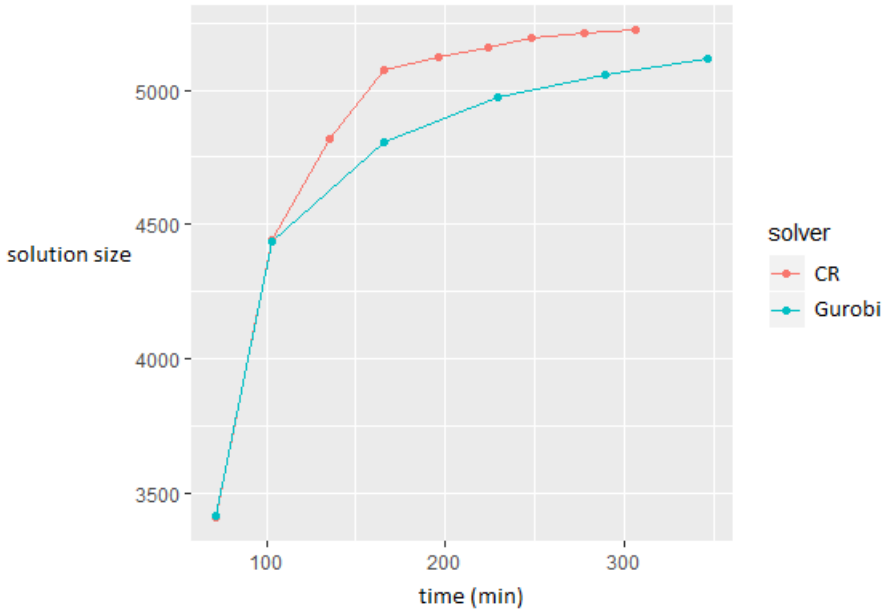

Figure 10: Evolution of Algorithm 1 with different solution algorithms with time limit of 6h on 5359 trains.

5.7 Discussion of the Results

Overall, we see in Table 1 that the randomized heuristics CR and ILS yield much better results than Gurobi on the timetabling instances. We are aware that the performance of Gurobi can be increased by a better modelling of the cliques in the conflict graph. But the tuned parameters that already force Gurobi to apply clique cuts and the huge gaps between the solutions of Gurobi and the randomized algorithms indicate that there is little hope that Gurobi can produce similar results. Only on the really easy instances where the optimum is found within 2 minutes, Gurobi performs similarly well. In particular, for the subnetwork instances all solvers perform equally well. Furthermore, we observe that the larger the instance, the better performs CR in comparison to both ILS and especially Gurobi. We deduce that the better performance comes with the more complex tight improvements compared to the simpler 2-improvements of ILS and the simulated annealing. Although the differences in performance between ILS and CR is only slight, with regard to the presumably even larger instances that will occur in the application of planning more than just a single day, we extrapolate that CR will be the best choice.

In the benchmark graphs, CR and ILS perform comparably well, as can be seen in Table 2, although ILS often has slightly better results. Again, the smallest instance can be solved to optimality by either of the algorithms. This motivates us that CR can not only be applied to conflict graphs, but is also in general a good heuristics.

Finally, we see our hypothesis affirmed that with a fast MIS heuristics in Algorithm 1, we can set a short time limit for each iteration and overall, achieve better results. That is, with a time limit of 2 minutes, we have enough iterations to assign 100 additional trains, with CR compared to Gurobi in the TPAP as can be seen in Figure 10 within 6 hours of computation. We see the major benefit of our approach in modeling the TPAP as a MIS and solving it with column generation while the effect on a slightly better MIS heuristic such as CR compared to ILS, only marginally affects the overall number of trains that can be scheduled in a given time.

6 Conclusions and Outlook

In this paper we proposed a column generation framework for the TPAP. While for the variable generation we call an oracle, we solve the TPAP as a MIS in the conflict graph with the MIS heuristics CR that extends the state-of-the-art heuristics ILS. We proved in experimental computations that indeed, for the timetabling problem, CR performs better than ILS and even more than a commercial MIP solver.

Moreover, we could prove with real-world data that when applying CR to the TPAP, we can set a lower time limit in each iteration and hence get more iterations in total which again enables us to schedule 100 out of 5359 more trains within a time window of 6 hours compared to applying a MIP solver. In particular, we could prove that our column generation approach enables us to schedule all freight trains in Germany in reasonable computation time. This has not been achieved before and is of great use for the infrastructure manager DB Netz.

As a possible prospect, we want to mention that the CR algorithm can be applied to even larger conflict graphs. It would be interesting if we were provided real-world data from a timetabling problem with more than just a single day. If many partitionings of the planning periods of the trains are given, the resulting conflict graphs will increase by a factor of the number of partitions found.

Furthermore, we see a potential in the parallelization of the timetabling process in Algorithm 1. While the slot construction oracle can run on a different machine for different trains, also the heuristic can compute while new slots are generated and then added to the model online, that is, while the MIS is being solved.

Finally, it is imaginable that in the very end of this process, an exact algorithm with a lot of running time is applied to solve the final MIS optimally. Recall that we use the heuristic to avoid that in each iteration, we need to spend a lot of running time on solving the MIS. If, however, this is done only once at the very end, the total running time can still be limited and better results can be achieved.

References

Andrade, D. V., Resende, M. G. C., and Werneck, R. F. F. (2012). Fast local search for the maximum independent set problem. *J. Heuristics*, 18(4):525–547.

Barrett, C., Jacob, R., and Marathe, M. (2000). Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837.

Borndörfer, R. and Schlechte, T. (2007). Models for railway track allocation. In Liebchen, C., Ahuja, R., and Mesa, J., editors, *ATMOS 2007*, volume 7.

Cacchiani, V., Caprara, A., and Toth, P. (2010). Non-cyclic train timetabling and comparability graphs. *Operations Research Letters*, 38(3):179 – 184.

Caprara, A. (2015). Timetabling and assignment problems in railway planning and integer multicommodity flow. *Networks*, 66(1):1–10.

Caprara, A., Fischetti, M., Guida, P., Monaci, M., Sacco, G., and Toth, P. (2002). Solution of real-world train timetabling problems. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*.

Dahms, F., Pöhle, D., Frank, A.-L., and Kühn, S. (2019). Transforming automatic scheduling in a working application for a railway infrastructure manager. *Rail Norrköping Conference 2019*.

Fuchsberger, M. and de León, J. M. (2007). Solving the train scheduling problem in a main station area via a resource constrained space-time integer multi-commodity flow. *Master's thesis, Institute for Operations Research ETH Zurich*.

Furini, F. and Kidd, M. P. (2013). A fast heuristic approach for train timetabling in a railway node. *Electronic Notes in Discrete Mathematics*, 41:205 – 212.

Grosso, A., Locatelli, M., and Pullan, W. (2008). Simple ingredients leading to very efficient heuristics for the maximum clique problem. *Journal of Heuristics*, 14(6):587–612.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science*, 220:671–680.

Kümmling, M., Großmann, P., and Opitz, J. (2015). Maximisation of homogenous rail freight train paths at a given level of quality. *27th European Conference On Operational Research, Glasgow*.

Nachtigall, K. and Opitz, J. (2014). Modelling and solving a train path assignment model. *Proceedings of the International Conference on Operations Research*.

Pöhle, D. (2016). Strategische Planung und Optimierung der Kapazität in Eisenbahnnetzen unter Nutzung automatischer Taktfahrplanung. *Disserta Verlag*.

Reisch, J., Großmann, P., and Kliewer, N. (2019). Conflict resolving - a maximum independent set heuristics for solving maxsat. *Proceedings of the 22nd International Multiconference Information Society*, 1:67–71.

Rosin, C. D. (2014). Unweighted stochastic local search can be effective for random CSP benchmarks. *CoRR*, abs/1411.7480.

Schlechte, T. (2012). Railway track allocation: Models and algorithms. *PhD thesis, Technische Universität Berlin*.

Sloane, N. J. A. (2015). Challenge problems: Independent sets in graphs. *https://oeis.org/A265032/a265032.html*.

Xu, K. (2005). Bhoslib: Benchmarks with hidden optimum solutions for graph problems. *http://www.nlsde.buaa.edu.cn/ kexu/ benchmarks/graph-benchmarks.htm*.

Zwaneveld, P. J. et al. (1996). Routing trains through railway stations: Model formulation and algorithms. *Transportation Science*, 30(3):181–194.