Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Efficient and Complete Conflict Graph Generation Schemes for Railroad Scheduling

Steffen Wonner

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Heiko Geppert
Commenced:	Juni 2, 2021
Completed:	December 2, 2021

Abstract

Railway networks already play a major role in today's transportation of passengers and goods. With the growing efforts to protect the environment and climate, its role is likely to become even more important in the near future. To handle this appropriately we need a way to schedule trains in a fast and efficient way. A recent publication for computer network scheduling made conflict graphs in their field viable, by building it up dynamically till a solution can be found. We to took this idea and implemented it for joint routing and railroad scheduling. We added a Potential Conflict mechanism to their algorithm to improve on its performance and compare different strategies to find the best way to extend the conflict graph for our domain. The evaluation shows that with those two adaptions we are able to scale the conflict graph solution from a single train station to the German railroad network.

Kurzfassung

Schienen Netzwerke spielen bereits heute eine große Rolle beim Transport von Personen und Gütern. Und mit den wachsenden Anstrengungen die Umwelt und das Klima zu schützen wird diese Rolle in naher Zukunft warscheinlich noch wichtiger. Um dem gerecht zu werden brauchen wir eine schnelle und effiziente Methode Fahrpläne zu erstellen. Eine neue Publikation im Bereich der Computer Netzwerke verwendet Conflict Graphs, indem diese dynamisch aufgebaut werden bis eine Lösung enthalten ist. Wir haben diese Idee auf das Streckenführungs- und Ablaufkoordinierungsproblem für Züge übertragen. Um die Performance weiter zu verbessern haben wir außerdem den Algorithmus um den Potential Conflict Mechanisus erweitert. Wir haben darüber hinaus verschiedene Strategien erprobt, um den effizientesten Weg zu finden, den Graphen für dieses Problem aufzubauen. Unsere Evaluation zeigt, dass wir mit diesen beiden Anpassungen nicht mehr nur den Verkehr für einzelne Bahnhöfe, sondern für Länder der Größe Deutschlands planen können.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Trains are an important means of transport, for people and goods alike. With politics starting to push more consequent for climate and environmental protection their role is likely to grow even further. Most parties have their plans to consolidate Deutsche Bahn [Wit21], especially the German green party [Die21a; Die21b]. But also Deutsche Bahn itself plans to improve by becoming CO_2 neutral till 2040 [Deu21a]. This is would be an important step as a major player in German mobility and transport with 4.874 billion passengers and 255 million tons of goods in 2019 [Deu21b]. Other companies also (re)discover the advantages of trains for their logistics. DHL has plans to increase the usage of trains from 2% to 20%, with one train replacing up to 25 trucks [Bar21]. Coca-Cola wants to connect 13 sites with trains and avoid three million truck kilometers and 1900 tons of CO_2 [Süd21].

An increase in trains requires an efficient usage of the existing rail capacities, as building new infrastructure can take a very long time [Tag21]. This is commonly known as railroad scheduling and describes different approaches on creating efficient timetables, by associating trains with departure and arrival times. It often is discussed in combination with the train routing problem. Train routing assigns trains to available tacks in the railroad network. We want to discuss this joint routing and scheduling problem in this thesis. Not many papers were published in this area in the last years, most papers are older than 15 years, many of which are from the last century. While many approaches were discussed they all share a common problem, which is scalability. The paper focuses on small sections of the network, like stations or junctions, with only a few trains and still take multiple hours. Because of this recent papers have shifted their focus to related problems and optimize other aspects. One example is rolling stock scheduling where the composition, location, and shunting of trains is also considered [BEF+21]. Because of this, the base problem did not much improve recently and more reason developments in similar areas did not get adapted.

In contrast, routing in computer networks was intensely studied in the last decade. Rapid development in hardware and requirements in this area combined with high economic relevance made this to a popular topic. Many papers were published and a lot of different approaches were discussed [NBT+19; NTA+]. One example is the approach presented and extended by Falk et al. [FDR20; FGD+21].

The conflict graph approach which was used by Falk et al. [FDR20; FGD+21] was already in use for railroad scheduling [LLER11] in the 1990s and 2000s. While then the scope was limited to very small sections of the complete railroad network these new papers consider complete computer networks. They introduced changes to the conflict graph which aim to reduce the cost of building it up, which does improve scalability. As the two types of networks are similar in many points, this might provide a base to scale conflict graphs to complete railroad networks.

The goal is a solution that scales well, while still finding good solutions. To achieve good scalability they can only build up partially as this is the most time-consuming factor. Which simultaneously limits the possible solution which can be found. This means that the conflict graph has to be built up strategically so that it contains a solution early on. In this thesis, we present different strategies to create conflict graphs for the domain of joint routing and railroad scheduling.

In Chapter 2 we will present the data structures and algorithms we used. We present the algorithm we adapted in its original form, without any modification. In the Chapter 3 we present earlier approaches to railroad scheduling and the algorithms they are often based on. We will also look at approaches that already utilized conflict graphs, what network they use it for and what important differences they have compared to our implementation. In Chapter 4 we will talk about railroad specific requirements and which restrictions we need, to get a model as realistic as possible in order to get valid results. We will also show how they are reflected in our implementation. We also formalize the restrains from the previous chapter and discuss the system model we derive from those the restrains. Chapter 5 presents our evaluation pipeline and discusses some of their implementation details. In Chapter 6 we define the structure of a strategy and different sub-strategies we experimented with. In Chapter 7 we first discuss the data our evaluation runs on, present the results of the different strategies and compare them to each other. We will also compare the capacity of our algorithm to those of older papers. We then discuss some questions which might be interesting for future work in Chapter 8. And lastly in Chapter 9 we sum up the results of our work.

2 Background

In this chapter, we will explain undirected, bidirected, and conflict graphs which are the basis for the algorithms used in this paper. We will also explain the algorithm proposed by Falk et al. [FDR20; FGD+21] which we aim to adapt for railroad scheduling.

A graph is a data structure describing the relation between object. It can be defined as a tuple $G(V, E)$, where V is a nonempty set of nodes representing the data and $E \subset V \times V$ representing the relations between them. In an undirected graph two edges (v_1, v_2) and (v_2, v_1) equivalent. In an directed graph those are two different edges, and the order defines the traversal direction. Also graphs can be weighted which is formalized as a triple $G(V, E, w)$. $w(e) \rightarrow \mathbb{R} : e \in E$ is a function which maps a value to every edge. This is commonly used to represent the cost of using an edge.

In this thesis. we use a directed, weighted graph to represent the railroad network. The edges represent a rail segment, the direction of the edge defines the direction the train can traverse it, and the weight represents the length of the segment. The vertices represent places where two or more rail segments connect. It might be a switch or simply a connection of two rail segments. We will use this graph to calculate routes between stations. A route is a list of rail segments that a train can traverse in this order to get from one station to another.

Further, we will use conflict graphs to model the solution space of the joint routing and scheduling problem. A conflict graph is an undirected graph commonly used in concurrency control [CB; Cha04; PY15]. Vertices require one or more resources and the edges between them represent a conflict in those requirements. Each applicant is represented by multiple vertices varying in the resources required or the time at which the resources are needed. Two vertices representing the same applicant can not conflict. Solving a conflict graph comes up to solving the independent vertex set problem. This means a set of nodes has to be found, which contains no edge. Every set of nonconflicting vertices is a valid solution, if every applicant has at least one selected vertex the solution is complete. Has an applicant more than one vertex in a solution one of them is selected, the others are discarded.

In this thesis, we want to utilize a conflict graph to schedule train lines on a railroad network. The lines are the applicants that we want to schedule. The line defines a list of stations that are visited in this order, the speed of the train, and an interval at which the line needs to be serviced. The vertices are configurations, which define a concrete route and the start of the first interval. The routes are made up of rail segments which are the resources for which we need to prevent concurrent access. Concurrent access would mean a violation of security requirements or even a train crash. After the first time, which is defined by the configuration, a line is serviced in the interval it is set to.

The big drawback of conflict graphs is the time it takes to build them. Adding a single configuration requires us to check against all existing configurations for conflicts which is expensive. The papers of Falk et al. [FDR20; FGD+21] propose an algorithm which ties to solve this problem by building the conflict graph in iterations. They aim to extend the conflict graph in a way so that the conflict

graph can be solved before the majority of configurations are created. The algorithm starts on a conflict graph with a few initial configurations. It then tries to find a set of nonconflicting configurations in which every line is represented once. For this, it selects the line with the worst perspective out of the lines not yet represented (cf. Algorithm 2.1). This is evaluated by the number of valid configurations, which are those that do not conflict with the configurations selected up to this point.

As a first tie-breaker the number of conflicts of the line is used. A higher number implies that the line is more likely to get shadowed by another line. Here shadowed means that it can not be selected because all of its configurations are in conflict with already selected configurations. As a second tie-breaker, the id of the lines is used to get a deterministic behavior. When the line is the line determined all valid configurations are evaluated. Here the goal is to select a configuration that has the least negative influence on the schedulability of other lines which is measured with the shadowrating (cf. Algorithm 2.2). To get the shadowrating of a configuration we iterate over all lines which have at least one conflicting configuration and sum up the eligible configuration to total configuration ratio. Here we need to check for cases where all the remaining configurations would be shadowed. To make such configuration unfavorable we fix the ratio to α. Falk et al. [FDR20; FGD+21] proposed a value of 1000 for α, which we adopted. This high value causes the algorithm to only select the configuration if all other configurations also block at least one line completely. The configuration with the smallest shadowrating is selected and added to the solution set.

In the paper of Falk et al. [FDR20; FGD+21] the so-called rerun mechanism has been shown to improve results. When the greedy flow heuristic algorithm did not find a complete solution the rerun mechanism triggers another run on the same conflict graph, giving the flows which could not be scheduled a higher priority. We use this mechanism by default, but strategies might deactivate it as it might not produce optimal results for all of them.

When a complete solution is found, either by a normal run or a rerun, the algorithm stops and returns the solution set. Did neither the normal run nor the rerun find a complete solution more configurations are added to the conflict graph. Which configuration gets added is determined by the Percentile strategy. Which configuration gets added is determined by a strategy. Falk et al. [FDR20; FGD+21] used a strategy they called Percentile strategy. Was the strategy not able to create more configuration, or in some cases enough configuration, the algorithm will return with the last incomplete solution it found. This is shown in 2.3.

The desirable result is a complete solution, but depending on the data this might not exist or can not be found by the heuristic. To determine the quality of a result and compare them we define the number of scheduled lines as the quality of a result. This means a solution is better than other solutions if it schedules at least a line more. Implicitly this also defines complete solutions as the best solutions.

Besides good solutions we also want scalability. Both are important, but contradict each other, as a complete conflict graph will lead to the best solutions, but scalability requires the conflict graph to be as small as possible. This means we have to find a good Pareto optimal, considering those two goals.

Algorithm 2.1 selects the next Line. As proposed by Falk et al. [FDR20; FGD+21].

```
1:  procedure SelectLine(lines)
2:      nextLine = lines[0];
3:      eligibleConfigurations = GetEligibleConfigurations(lines[0]);
4:      allConfigurations = GetAllConfigurations(lines[0]);
5:      for all l in lines do
6:                                          // Primary criteria: least eligible configurations
7:          if eligibleConfigurations < GetEligibleConfigurations(l) then
8:              continue;
9:          else if eligibleConfigurations == GetEligibleConfigurations(l) then
10:                                             // First Tie-Breaker: total configuration
11:             if allConfigurations > GetAllConfigurations(l) then
12:                 continue;
13:             else if allConfigurations == GetAllConfigurations(l) then
14:                                                 // Second Tie-Breaker: unique id
15:                 if nextLine.Id < l.Id then
16:                     continue;
17:                 end if
18:             end if
19:         end if
20:         nextLine = l;
21:         eligibleConfigurations = GetEligibleConfigurations(l);
22:         allConfigurations = GetAllConfigurations(l);
23:     end for
24:     return nextLine;
25: end procedure
```

Algorithm 2.2 calculates the shadowrating for a configuration. As shown in Falk et al. [FDR20; FGD+21].

```
1:  procedure GetSchadowRating(c)
2:      schadowRating = 0;
3:      L_nieghbour ← Lines which have at least one conflict with c;
4:      for all l_nieghbour in L_nieghbour do
5:          schadowCount ← number of eligible configuration of l_nieghbour shadowed by c;
6:          eligibleCount ← number of eligible configuration of l_nieghbour;
7:          δ = schadowCount \eligibleCount;
8:          if δ == 1 then
9:              schadowRating += α;
10:         else
11:             schadowRating += δ;
12:         end if
13:     end for
14:     return schadowRating;
15: end procedure
```

Algorithm 2.3 scheduling Algorithm. As proposed by Falk et al. [FDR20; FGD+21].

```
 1: procedure SCHEDULER(strategyName)
 2:     strategy = GetStrategy(strategyName);
 3:     strategy.Init();                                  // Creates initial configurations
 4:     repeat
 5:         solution = Solve();
 6:         if solution.NotScheduled.IsEmpty() then
 7:             return solution.Scheduled;
 8:         end if
 9:         priority = solution.NotScheduled;
10:         solution = Solve(priority);
11:         if solution.NotScheduled.IsEmpty() then
12:             return solution.Scheduled;
13:         end if
14:         if strategy.AddConfigurations() then
15:             return solution.Scheduled;            // Could not create new Configurations
16:         end if
17:     until true
18: end procedure
```

3 Related Work

In this chapter, we will present other works in the area of railway scheduling, which approaches are commonly used, and how they are used. We also show that there is still work to be done as the approaches are not very scalable.

The paper by Lusby et al. [LLER11] written in 2011 is a survey paper about railroad scheduling. It is a quite comprehensive collection containing papers back till the 1980s. It shows that up to that time, integer linear programming (ILP) was the dominating approach on railway scheduling.

Cacchiani et al. (2008) [CCT08] consider a single one-way track between two major stations and a number of minor stations in between. Trains are defined by the stations, a list of consecutive stations, which they service. Trains are only able to overtake each other within stations and lower bounds restrict the intervals between two consecutive arrival or departure times. They present a heuristic approach based on the LP Relaxation, arguing that an exact solution would require too much computational power. The test cases presented contained up to 102 stations with 41 trains or 17 stations with 221 trains.

Borndörfer et al. (2005) [BGL+06] present an auction system, where companies bid on connections. The auction consists of multiple rounds, whit a bidding phase and an allocation phase each round. In the bidding phase, all bids are submitted simultaneously, then in the allocation phase, a linear programming-based optimizer decides which bids are accepted. The tests contained up to 737 trains, this instance took 3 days to compute.

Borndörfer and Schlechte (2007) [BS08] proposed improvements on the linear programming representation of the problem. They encode conflicts with variables instead of constraints. This allows for a more easy appliance of the column generation techniques. The test contained 37 stations and up to 570 trains and was solved in about 16 hours.

Later publications in this area mostly focus on rolling stock instead of pure scheduling. This means that the type of trains, waiting periods, coupling/decoupling, and moving trains from end stations to the next start station are also considered. A prominent paper is from Borndörfer et al. (2021) [BEF+21]. It was written in cooperation with Deutsche Bahn and seems to be the most holistic approach published yet. Details on how the problem is solved are not disclosed.

Another approach already used in the 90s and early 2000s was the conflict graph. Because of the cost of building up a conflict graph, it was only feasible to compute small parts of the network. Thus it was used to calculate the routing within train stations or bigger junctions.

Zwaneveld et. al. [KEZ97; ZKR+] have published multiple papers proposing a routing solution for railway stations. The algorithm requires the structure of the train station and for all trains the time of arrival and departure and the tracks on which they enter and leave the station. The algorithm then considers all valid routes for those parameters. The paper proposes a node package problem-based

algorithm, which is semantically identical to a conflict graph but also uses linear programming and LP-relaxation aspects. The evaluations were conducted on single train stations with 18 trains scheduled over an hour.

Cornelson and Stefano [CD07] worked on a simplified network representing a station with fixed arriving and departing times. The trains also have fixed entry and exit points, all valid paths for those are considered. The conflict graph is built up completely before trying to solve it and represents only the train station. The paper did not provide any information about evaluation sizes or runtime.

Lusby et al. [LLRE11] present an approach similar to a conflict graph. They use a node set approach to route trains through junctions. This is defined as a set S of elements and a set of subsets $X = X_1, X_2, ..., X_n$ of the elements of S, a valid solution is a packing P for which holds that $X_j \cap X_k = \emptyset, X_j \neq X_k \in P$. In this paper, each node represents a certain rail segment at a certain point of time, the subsets in X representing valid routes through the junction, the nodes in it encoding the position of the train in space and time. The algorithm then tries to find one X_i for every train so that they do not have a common node. The information needed for this step could be represented as a conflict graph. The paper instead uses linear programming. In the paper, all nodes for the set packing problem are created beforehand. As trains can have multiple routes, can accelerate and decelerate, the number of nodes can be quite high. In the example a single route of a train created up to 492,907 nodes, causing the examples with 9 trains and 19 routes to contain 1.7 million nodes.

Delorme et al. [DRG01] propose two approaches, one using linear programming and one based on the unicost set packing problem. The second approach generates initial solutions with the greedy randomized adaptative search procedure. It then tries to refine the solution by a local search phase. The evaluation had up to 97 trains.

The common weakness of all those papers is scalability. The paper evaluates their approach on small networks with a few trains. The the papers on ILP need 16 hours [BS08] or three days [BGL+06] for less than 1000 trains. Papers with conflict graph approaches do not disclose the computational time their evaluation needed, but evaluations with less than 100 trains [BGL+06; DRG01] suggest that they have a similar problem with scalability. Deutsche Bahn Group had 23.466 passenger trains each day [Deu21b].

Falk et al. [FDR20; FGD+21] proposed a conflict graph-based solution for routing in computer networks. While working on another type of network, the approach is similar to those for railroad networks. The two big differences are that this paper considers the complete network instead of a small section and then the conflict graph is built up as needed instead of creating it completely direct in the beginning. The Greedy Flow heuristic (GFH) is used to solve the conflict graph. If it does not find a complete solution the Percentile Strategy determines which configuration should be added before GFH algorithm tries to solve it again. This new approach to the conflict graph can drastically speed up the algorithm. The paper does present a evaluation with 800 flows, the train equivalent, that is solved in less than 3 minutes.

This approach has the potential to improve railway scheduling by adapting it. This could speed up the process drastically and may allow to schedule a realistic number of trains for whole networks instead of local portions.

4 System model

In this chapter we discuss the requirements we have, to create a context that is as realistic as possible for railroad scheduling. We also discuss which changes have to be applied to the conflict graph algorithm presented by Falk et al. [FDR20; FGD+21] to adapt to this new context. We then want to formalize the conditions and assumptions which we assume for this thesis.

Earlier approaches did work on prefabricated conflict graphs, in contrast, we build it up dynamically. This means that we have to dynamically detect conflicts between configurations. We base the definition, of which configurations are in conflict, on the regulations of Deutsche Bahn.

The Deutsche Bahn does use the system driving in “festen Raumabstand” [Pac16]. This means that the network is divided into segments of the same length (For Deutsche Bahn it is roughly 1000 meter [Pac16]). Those segments are prefaced with signals which allow trains to enter only if the three following conditions are fulfilled:

1. No other train is on the segment
2. The overlap (ger: “Durchrutschweg”) behind the segment has to be free
3. The previous train has to be protected by a stop signal

This is the system used on nearly all railroads today [Pac16].

Implementing this would have multiple disadvantages. Firstly we can only sparsely extract the required segments from OSM, we would need to generate the missing ones our self, matching the acquired segments, or generate them ourselves, which is out of scope for this thesis. It also would drastically slow down the algorithm as the conflict detector has quadratic runtime in relation to the number of segments in all routes.

To avoid those problems we decided to adapt “bewegter Raumabstand”. Here sensors have to monitor the distance to the train ahead and decelerate or stop if the trains get to close to each other. In reality, this is seldom implemented yet because of the complexity. We choose this approach because of lesser computational requirements and because with an appropriately chosen distance results can be transferred to “festen Raumabstand” systems. Also the EU plans to unify train control with the European Train Control System(ETCS) which will on level three either work with virtual “fester Raumabstand” or with “bewegter Raumabstand” [All21; DB 14; DB 18; EU21].

As long as the results are not transferred to a real railroad network, the chosen safety distance only influences the number of trains required to use the full capacity of a rail segment. Because of this, we chose to use the 1300 meter the Deutsche Bahn uses for “festen Raumabstand” without further adaptation.

Based on those regulations we decided to implement the conflict detection so that two trains are in conflict when they traverse the same rail segment with less than 1300 meters between them. This represents a valid system in the real world, is comparable to the system most commonly used, and is easy to implement. It also will be used on the European network eventually [DB 14; DB 18].

The context created by all those assumptions and decisions differs in some aspects greatly from the context of Falk et al. [FDR20; FGD+21]. This has a great impact on the performance. One of those is that a cycle is between $250\mu s$ and $2000\mu s$ with a granularity of $1\mu s$ for computer networks, while on railway networks cycle are 15 to 60 minutes with a granularity of 1 minute. This means that on computer networks we have 4 to 130 times more configurations for an applicant. Due to that, we expect strategies to perform worse as the differences in the timing of the configuration available are a lot smaller than in the computer network. To compensate for the performance loss we introduced another optimization. For this, we exploited that the smaller number of configurations per route means, that the size of the conflict graph depends dominantly on the number of routes. We introduced a relation between routes which we called Potential Conflict. Those relations mark pair of routes that overlap. This means that there is at least one rail segment that they both traverse. Only then a conflict between routes can occur and we can limit the search for conflicts to those which belong to a route with Potential Conflict.

We consider a railway network. The network consists of rail segments, switch nodes, and stations nodes. Switch nodes connect multiple rail segments and allow a train to pass to any outgoing railway segment. Every station can be a source or a destination station for a line, trains traverse the network according to the schedule.

Switches in reality only allow for a specific combination of entry and exit points. As incorporating this restriction would only affect pathfinding, we decided to ignore it. The algorithm solving the conflict graph does not check routes for validity and thus is not affected by this decision.

Our representation of the network has fewer parallel tracks and thus less capacity. Because of this, we decided to model the trains with a length of 0 meters. A train is not able to accelerate, slow down or wait. We only consider normal operation - power outage, or break down of any hardware is out of scope. Each train has a safety distance of 1300 in front of it which other trains are not allowed to enter. This simulates driving in "bewegtem Raumabstand"[Pac16], the size of this distance is the same the Deutsche Bahn uses for the "festen Raumabstand"[Pac16]. This is necessary to get results that are transferable to real scenarios.

Trains represent the notion of a line for certain parameters. A line does consist of a route from a source station to a destination station. A train start for the first time at $t_0 + \phi$ where t_0 is a common time-reference and ϕ is the offset. After this it departs in intervals $t_{interval}$. This means the departure times are defined as $k * t_{interval} + \phi + t_0, k \in \mathbb{N}$. Source, destination, speed, and $t_{interval}$ are set line parameters and do not change during the lifetime of a line. Before creating a new schedule all lines have to be added specifying those parameters for the planer. In addition, one or more routes have to be given, from which the planer assigns one together with an offset ϕ, which has to be in the range of $[0, t_{interval}[$

As a continuous time variable ϕ would yield infinitely possible configurations. We choose an appropriated granularity with 1 minute. This does reflect the granularity of real train schedules.

For the speed of the trains, we decided on a uniform 100 km/h. This estimate on an average speed is based on the top speeds available at OpenRailwayMap[Rei21]. As this value only influences the capacity of the rail segments the precision does not matter for evaluating this algorithm.

Most train stations on the railroad graph are represented by multiple vertices. As it is a nontrivial task to decide whether they are placed on the graph in a way that would increase the capacity of the station, we decided to select one of the vertices randomly to represent the station. With that, the capacity of a station is limited to 60 outgoing trains per hour per outgoing edge. This number is unrealistic high but compensates for the fact that most edges represent more than a single track. Because of terminus stations and stations at the outskirt of the network it is to be expected that for some stations all traffic will leave on the same vertex. The same arguments hold for inbound trains. This means that we have to expect bottlenecks in front of train stations. In our experience, with the intervals we used in our evaluation, around 20 lines can pass the bottleneck in each direction.

5 Implementation

In this chapter, we will present the toolchain we implemented and used to evaluate the different conflict graph building strategies. It does get the base data from OSM, prepares them, evaluates our strategies, and creates diagrams from the collected data.

We extract the railroad network from OpenStreetMaps (OSM) with a tool created for this purpose [PGRR]. The exported graph edges represent the rails and vertices represent a point on the rails, switches or train stations. The data consists of a file containing the rail segments represented as an adjacency list and a file containing a list of train stations with their names and coordinates.

As we need to generate routes for our trains we have to run a pathfinding algorithm on the railroad network. This requires coordinates for all vertices in the graph. For this, we modified the tool so that the station file contains that information as well.

The exported graph is a bidirectional graph with many very short edges, often representing less than 10 meters of rails. This is due to OSM representing paths as a list of points, which causes points to be close together where curves has to be represented. In the first step we remove all vertices which have exactly two edges (and are not a train station) and contract the two edges. This reduces the number of edges to less than $\frac{1}{10}$.

We then convert the graph to a unidirectional graph by replacing all edges with two new ones. The two edges are unidirectional edges that point in opposite directions so that trains can traverse in both directions without interfering with each other. We choose to do this to get a more accurate representation of the real network, as 2 (or more) rails are standard and normally they are one ways [Pac16]. This might cause some segments to be represented with more tracks than they have in reality, if every single track is in the OSM data. This is unlikely to have an influence on our implementation, as it should only occur sparsely and pathfinding will always use the shortest one. We save the new graph in the same format as the extraction tool.

First, we generate lines on the currently used graph. For this, we manually selected the main stations of the biggest cities on the graph and then used A* to find the shortest routes between them. We choose an interval and speed for the line based on the cities connected. With this we want to represent the different loads on high and low frequented rail sections. If both cities are in a defined group of important cities, a faster interval is set than when less important cities are involved. We save the generated lines as line templates to a file.

Our strategy evaluation tool takes three files. The adjacency list, the line template list, and the task file. The task file contains a list of strategies that are to be evaluated and the parameters for the strategies. A second list defines the number of lines for each run. At the start of each run the lines are selected, all strategies are evaluated on this list for this run. A flag defines whether the lines are selected randomly or starting from the beginning of the file, when there are more lines the needed.

It then initiates a conflict graph for the first strategy with the data from the adjacency list. This data is required as we need the detect conflict when adding configurations. For this, we need the length of rail segments for the collision detection. We implemented lines, routes, and configurations in a hierarchical structure. This has multiple advantages: Separating the routes from the lines allows lines to have any number of routes. Associating the configurations with routes instead of directly with the lines enables us to optimize adding new configurations to the conflict graph. For this, we introduce Potential Conflicts between routes. Those mark pairs of routes which share one or more common rail segments. If we now add a new configuration it can only conflict with configuration from a route with a Potential Conflict. Because of this, we can limit the search of conflicting configurations to those and thus speed up the process.

To solve the conflict graph the tool uses the algorithm described by Falk et al. [FDR20; FGD+21], which we described in chapter 2. We added a mechanism that collects different timings and intermediate results while solving the graph for evaluation purposes.

If the graph is solved the conflict graph is cleared and the next strategy is selected. After iterating over all strategies the next run is started by selecting new lines and going back to the first strategy in the task list. After finishing the task all collected data is saved to a file.

We used jupyter notebook to import the evaluation files, prepare the data and create diagrams with pyplot and sealion.

6 Strategies

The strategies described in this chapter determine in which order the configurations are added to the conflict graph. Each strategy consists of three sub-strategies, namely the line-strategy, the offset-strategy, and the route-strategies. The line-strategy determines which lines will get additional configurations, the offset-strategy determines the offsets for the new configurations and the route-strategy selects a route for every new configuration.

In this thesis we will not use any route strategies. We calculated the shortest paths for every pair of stations with the largest cities on the graph. This has shown that the second shortest path differs on less than 1% of the route from the shortest path, for nearly all station pairs. For the third shortest path this still holds for the majority of the stations' pairs. This means that the paths are too similar for our purpose. As the creation of sensible routes is out of the scope of this paper we limit ourselves to line- and offset-strategies.

6.1 Offset-Strategies

We will now describe the offset-strategies and what idea they are based of. The strategy does determine the offset for a new configuration. Every line will get initial configurations in the preparing phase. If not stated otherwise those are based on the same rules as the configurations created later on.

The **Random** strategy randomly selects a yet not used offset. We implemented this as a reference for the evaluation of other strategies.

The **Percentile** strategy proposed by Falk et al. [FDR20; FGD+21] includes a route-strategy which we will ignore for the reasons mentioned, the AllLines line-strategies which we will discuss later on, and the offset-strategies, for which we will use the name in this paper. The offset-strategy calculates the time in which 75% of the computer network applicants can finish transmission. The next offset is determined by adding this value to the last offset. If the next offset is larger than the interval, it is reset and the initial offset is increased by 1. The goal is to skip until we expect most currently active network allocations to be gone and see if it is possible to schedule it there.

The percentile strategy was originally created for computer networks and relies on the transmission time being significantly shorter than the interval. For trains, this is very unlikely, especially for inter-city trains where the inverse can be true. This causes offset increase bigger than the cycle time and thus the configurations are created from first to last (cf. Figure 6.1). To transfer the idea to railroad networks we implemented the **Local-Avoidance** strategy. Here the time the offset increased is determined by the time the individual train needs to traverse the safety distance. This causes the new configuration to avoid all conflicts the last configuration has. This results in a configuration distribution which Percentile is meant to create (cf. Figure 6.2).

Figure 6.1: Configuration creation order of the Percentile strategy on railway networks. (P: created in the preparation phase)

Figure 6.2: Configuration creation order which the Percentile strategy should cause. (P: created in the preparation phase)

Both, the LocalAvoidance and Percentile strategy, have the problem that it takes a high number of steps until configurations are distributed over the complete interval. For LocalAvoidance the increase is only 1 or 2 minutes depending on the speed of the train, thus it requires at least 15 configurations to cover an interval of 30 minutes, which is common. To cover the interval faster we came up with the **Bruteforce** strategy. The Bruteforce strategy increases the offset for new configuration by an amount that is manually set beforehand. With a good chosen step size the configuration faster covers the complete interval.

The disadvantage of Bruteforce is that that it requires a manually chosen parameter which is the same for all lines. Also, it still starts at one end of the interval and slowly covers the rest from there. To get a more dynamic approach we implemented the **DivideAndConquer** strategy. For every line, the biggest gap between the existing configurations is determined and a configuration is created in the middle (cf. Figure 6.3 and Algorithm 6.1). This means the next configuration is always created within the biggest possible time gap to the existing configuration, resulting in the most uniform distribution we can achieve in this situation. This does consider every line individually, which means that it does not require some sort of compromise for lines with different intervals.

Figure 6.3: Configuration creation order of the DivideAndConquer strategy. (P: created in the preparation phase)

Algorithm 6.1 selects the next configuration for the DivideAndConquer strategy.

```
1:  procedure DeterminOffset
2:      existingConfigurations = GetExistingConfigurations();
3:      gapsize = 0;
4:      centralindex = 0;
5:      gapstart = 0;
6:      for i = 0; i < existingConfigurations.Length; i++ do
7:          if existingConfigurations[i] then          // Configuration with this offset exists
8:              if start == i then                     // No gap, update start
9:                  start = i + 1;
10:                 continue;
11:             else                                   // Gap found
12:                 if i - start > gapsize then        // Is gap the biggest yet
13:                     gapsize = i - start
14:                     centralindex = (start + i) / 2;
15:                 end if
16:             end if
17:         end if
18:     end for
19:     if existingConfigurations.Length - start > gapsize then   // Handels end of array
20:         gapsize = i - start
21:         centralindex = (start + i) / 2;
22:     end if
23:     return centralindex;
24: end procedure
```

With all those strategies all lines get new configurations with the same pattern. This means that lines that start at the same station will always have pairs of conflicting configurations for the rail segment in front of the station. To cause lines to get configurations in an individual patterns we will seed the lines by selecting the configurations in the preparation phase randomly, for the DivideAndConquer strategy (cf. Figure 6.4).

Figure 6.4: Configuration creation order of the seeded DivideAndConquer strategy. (P: created in the preparation phase)

6.2 Line-Strategies

The line-strategy determines for which lines configurations are created. The trivial solution is the **All-Lines** strategies, here all lines get an equal amount of configurations. This is the line-strategy used by Falk et al. [FDR20; FGD+21] in the percentile strategy they proposed. It is also the line-strategy we use later on to compare configuration-strategies.

Another approach is the **ConflictCount** strategies. Here we tried to determine lines that are hard to schedule by the number of conflicts that they currently have (cf. Algorithm 6.2). The strategy creates a ranking based on this number and adds new configurations on lines based on the position of the line in the ranking.

Algorithm 6.2 creates configurations for the ConflictCount strategy.

```
1:  procedure ExtendConflictGraph
2:      ranking = new SortedList();
3:      lines = GetAllLines();
4:      for all line in lines do
5:          if line.AllConfigurationCreated then          // Ignore lines which already have all
    configurations
6:              continue;
7:          end if
8:          ranking.Add(line.GetNumberOfConflicts(), line);
9:      end for
10:     for i = 0; i < ranking.Length; i++ do
11:         position = i+1 / ranking.Length;              // Calculate position in ranking
12:         numberOfNewConfigurations = GetNumberOfNewConfigurations(position);
13:         CreateConfiguration(line, numberOfNewConfigurations);
14:     end for
15: end procedure
```

We suspected that the number of conflicts might not be the best metric to determine how hard it is to schedule a line. Many competing configurations from one line are valued the same as many lines each with a few conflicting configurations. To improve on the ranking heuristic we propose the **ConflictRatio** strategy. We try to get a more accurate estimate by considering the conflicts proportionately to the number of Potential Conflicts a line has. We will also call this a ConflictCount variation.

To see if the heuristics we came up with have an effect we implemented the third strategy. The **InverseConflictCount** strategy uses the same heuristic as ConflictCount but sorts the lines in the inverse order. We will also refer to this as a ConflictCount variation.

The implementation shown in Algorithm 6.2 vary for other ConflictCount variation only in line 8. InversConflictCount does multiply the number of conflicts with -1 to invert the order. And ConflictRatio does divide it by the number of Potential Conflicts the route has.

With the **NotScheduled** strategy we tried to restrict the number of lines that get additional configurations to a minimum (cf. Algorithm 6.3). At the end of an iteration, all lines which were not successfully scheduled are selected to get new configurations. As the strategy works based on the results of the solving or rerun phase the rerun mechanism which was introduced by Falk et al. [FDR20; FGD+21] does influence the strategy greatly. In chapter 7.3 we compared both variations, and show that they should be deactivated for this strategy.

Algorithm 6.3 implementation of the NotScheduled strategy.

```
1: procedure ExtendConflictGraph
2:     numberOfNewConfigurations = GetNumberOfNewConfigurations();
3:     for all line in GetUnscheduledLines() do
4:         AddConfigurations(line, numberOfNewConfigurations);
5:     end for
6: end procedure
```

The disadvantage of NotScheduled is that it can cause problems when from a conflicting pair of lines one gets always scheduled. Normally this behavior causes the rerun mechanism to prioritize the other line to solve such locks, but we do not use that with this strategy. To prevent such locks from stopping the algorithm early, we implemented the **ExtendedNotScheduled** strategy. We added that if a line has all possible configurations and still can not be scheduled, all lines with Potential Conflicts get new configurations instead. This is a trade off as it causes more and not as targeted configurations, because we do not know which Potential Conflict causes the problem. It is more complex und thus obviously slower than NotScheduled, but might result in better solutions.

With the combination of conflict- and line-strategy we try to minimize the resulting conflict graph. The line-select minimizes the number of lines that are extended and the configuration-select aims to make lines with as few configurations as possible schedulable.

7 Evaluation

In this chapter, we will discuss the empiric evaluation of different strategies. For this, we will describe our evaluation environment, present the data which we used for our evaluation, and the metrics we observe.

7.1 Enviroment and Data

Our evaluation tool is implemented in C# .Net Version 5.0.402. It runs on a server with two AMD EPYC 7401 24-Core Processors, each with 48 Threads, and 128GB RAM. The OS is Ubuntu 20.04.3 LTS.

The Deutsche Bahn has some data about their railroad network available [DB 16], but the data is incomplete to a degree where we can not use it for our evaluation. Therefor, we decided to use data provided by OpenStreetMap. This allows us to get graphs of different sizes and areas. We converted the data we get from OSM to a directional graph and removed unnecessary vertices (cf. Chapter 5).

For the evaluation, we picked three graphs, Germany and the two federal states Baden-Württemberg and Nordrhein-Westfalen (cf. table 7.1). We decided to do our evaluation on German railroad networks because it is ample and dense. Baden-Württemberg is the smallest of the three and already of a size where railroad scheduling is reasonably complicated and way larger than any railroad networks used in earlier papers [LLER11]. For the second stage we choose the network for Nordrhein-Westfalen, it is larger than the Baden-Württemberg graph but does represent roughly the same area. This means we can see the performance of the strategies on a more dense network. Lastly, we choose Germany as our third graph to see the performance on very large graphs and test out how many lines we can schedule in this realistically sized scenario.

To get a realistic scale on the traffic we looked at the statistic published by Deutsche Bahn which states that in the year 2019 about 24.000 passenger trains were on the network each day [Deu21b]. Which in our model is equivalent to 1000 lines. Normally most of them would be during the day, but our network capacity is not sufficient for that (as discussed in chapter 4) so we have to distribute

	OpenStreetMaps		**Optimized Graph**		**Reductions**	
Region	**Vertices**	**Edges**	**Vertices**	**Edges**	**Vertices**	**Edges**
Baden-Württemberg	183,458	301,964	12,915	38,404	92.96%	87.28%
Nordrhein-Westfalen	266,836	543,124	32,616	87,588	67.77%	83.87%
Germany	1,350,756	2,888,708	216,535	675,772	83.96%	76.60%

Table 7.1: The graph sizes before and after optimization.

them evenly. We created those lines by selecting the main stations from federal state capitals and filling them with the main stations of the 34 biggest cities to 50 stations. We created lines between each of the capitals with an interval of 30 minutes. We then filled it up with lines that have intervals of 60 minutes until every station is the source station of 20 lines. For the destination station, we selected the stations closest to the source station. All lines operate with a speed of 100 km/h as we discussed in Chapter 4. We used this as our baseline for the amount of traffic on a graph, we call this normal traffic. To get low and very low traffic we filled the connection only up to 15 and 10 respectively.

For the Nordrhein-Westfalen we selected the main station of the 35 biggest cities. We create lines between each of the 10 biggest cities with an interval of 30 minutes and fill them up to 10, 15, or 20 the same way as we did on the Germany graph. On the Baden-Württemberg graph, we did the same with the 20 biggest cities.

We determined the number of Cities based on the observation that most of the time more than 20 outgoing connections could not be scheduled (as discussed in chapter 4). The rail segment in front of the station node is a bottleneck which causes this limit. Because of this, we selected 20 cities for Baden-Württemberg. To get to the 24.000 trains a day with 20 outgoing, mostly 60-minute cycle, lines we need 50 cities. Thus we selected 50 cities for the German network. For Nordrhein-Westfalen, we decided to select 35 as it is the midpoint of the other two.

As the primary metric for evaluation, we choose the ratio of created configurations to schedulable lines. This represents the efficiency of the strategy for building up the graph. It is the major factor in the runtime of the Greedy Flow Heuristic. It also is an indicator for the number of iterations the overall algorithm has gone through and thus the number of times the Greedy Flow Heuristic was executed. As a second metric, we choose the runtime. As we aim for scalability the runtime can be a misleading factor especially for evaluation on small graphs. The strategies do influence the runtime in three ways, the runtime they require to add new configurations, the number of iterations, and the number of configurations they create. While for small railroad networks the time needed to extend the conflict graph is negligible, for larger networks it gets a major factor for the overall runtime. Because of this, we decided to make the graph buildup efficiency our primary metric, as it is the best indicator for scalability.

In all the following graphs, which plot the not scheduled lines, two effects can be seen to varying degrees. One is a slight bump in the number of lines not scheduled after around half of the conflict graph is built up. The other is the reduced number of lines scheduled after the complete graph is built up. Both effects lessen with bigger graphs and more traffic. We think that this occurs when newly added configurations cause the evaluation to rate a line better than before without improving the schedulability. After half the graph is built up with the AllLines strategy, an evenly spread coverage with enough choices is reached. The configurations created after this fill gaps, but have already existing configurations with a similar offset. Thus causing the line to be better rated without being easier to schedule. The fact that the ConflictCount strategies have this effect less pronounced supports this assumption as they cause the lines to hit this point at varying iterations. The bump in the last iteration is likely due to the same effect.

For the evaluation we look at the two types of sub-strategies we have separately. The first is the configuration-selection sub-strategy which decides which offset a new configuration has and the second group is the line-selection sub-strategy which determines which lines get new configurations. We then compare them within the groups and create our strategy from the best sub-strategies. In the

configuration-select group are the Percentile, Local-Avoidance, Bruteforce, and DivideAndConquer strategies. We also include the Random strategy as a reference strategy. All of them use the AllLines sub-strategy so that we can compare the effect of different configuration distribution patterns without additional effects from the line strategy. In the lines-selection group, we have ConflictCount, ConflictRatio, and Not-Scheduled strategies. Here we used the seeded DiviedAndConquer strategy, as it is the best configuration-selection strategy, for all of them to be comparable.

7.2 Configuration-Select Substrategies

We evaluate the configuration-selection group first as all strategies have the same line-select sub-strategy. this allows us to find the best configuration-select strategy without noise from varying line-selection strategies in the data.

As a first step, we compared different parameters for the Percentile and the Bruteforce strategies, to find the optimal parameter for each of them. We run the strategies with a large number of different parameters and compared the number of iterations they needed to schedule most of the lines.

We run this on all 3 networks with all intervals from 1 to 25 for the Bruteforce strategy. All results are close together with slightly better results around 13 and 25 intervals(cf. Figures 7.1 to 7.3). We choose an interval of 13 as it seems to produce the most consistent results over iterations and different graphs. Both values are quite large compared to the interval length used for the lines, this could be a sign that the difference of the configuration overall is quite small and thus creating a few configurations distributed over the whole range is preferable. It is also noticeable that the second value is roughly a multiple of the first one and both cause a configuration with nearly the maximal offset for lines with 30 and 60 intervals in the first iteration.

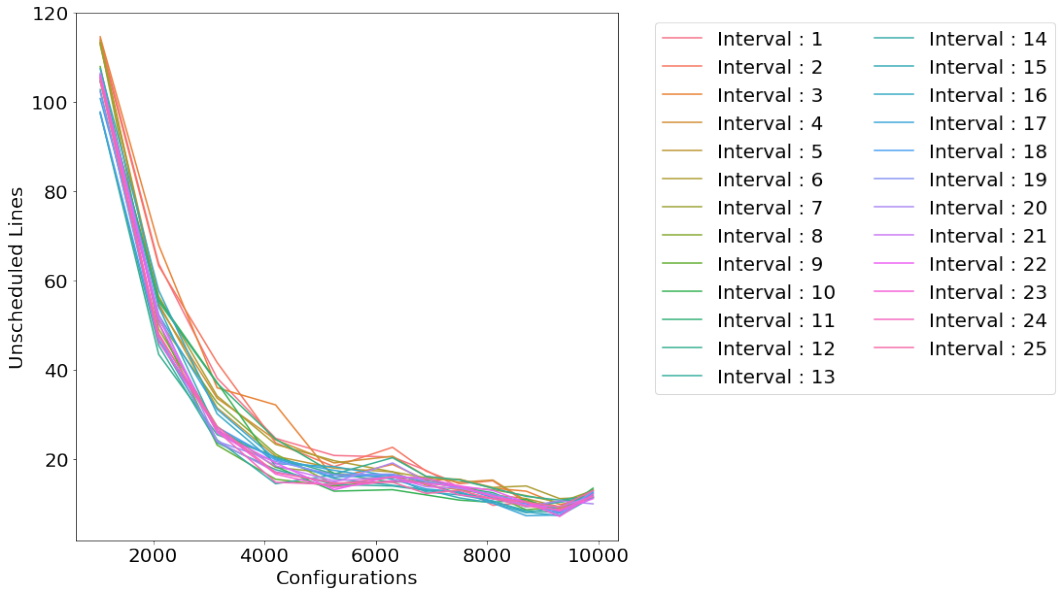

Figure 7.1: Bruteforce strategy on Baden-Württemberg graph with different parameter.

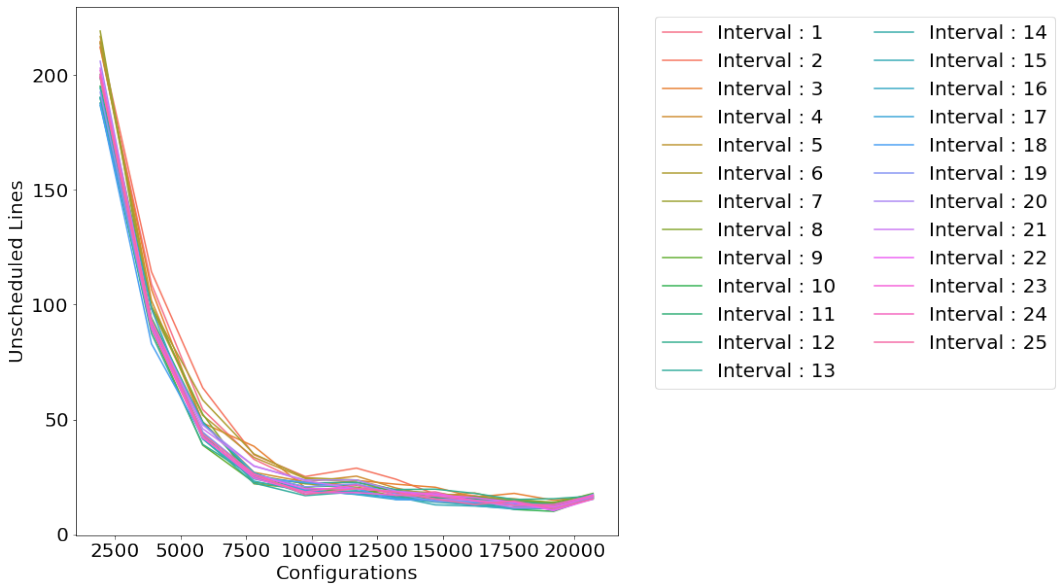

Figure 7.2: Bruteforce strategy on Nordrhein-Westfalen network with different parameter.

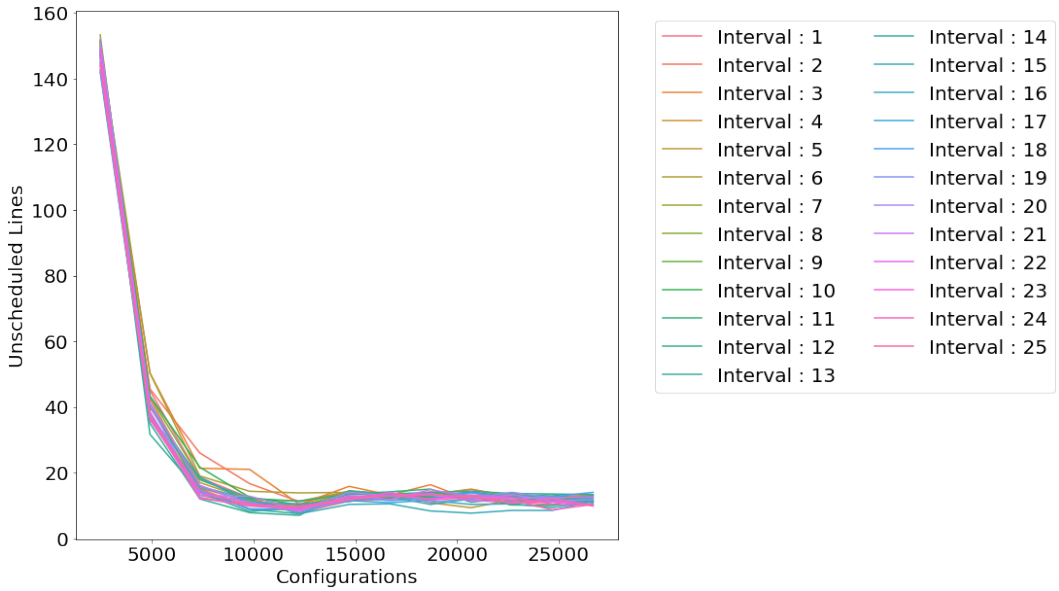

Figure 7.3: Bruteforce strategy on Germany graph with different parameter.

For the Percentile strategy, we have similar results. We selected values evenly spread out between 10% and 120% in an first step. As as second step we added more values around 60% as it had the best results in the first round. For the Baden-Württemberg graph and the Germany graph, a percentile value of 0.65 does produce the best results with most of the other parameters resulting in negligible differences. For the more dense network of Nordrhein-Westfalen, all parameters produce approximately the same results.

Comparing the four strategies of this group against each other shows that the differences are small. Percentile and Bruteforce have an advantage over LocalAvoidance and DivideAndConquer. But all of them are worse than our reference strategy Random, see Figure 7.4. This is most likely due to all lines acquiring new configurations in the same pattern. To see if we can improve based on this assumption we modified the DivideAndConquer strategy so that the initial configurations are selected randomly, adding later configurations based on the gaps unique to each line.

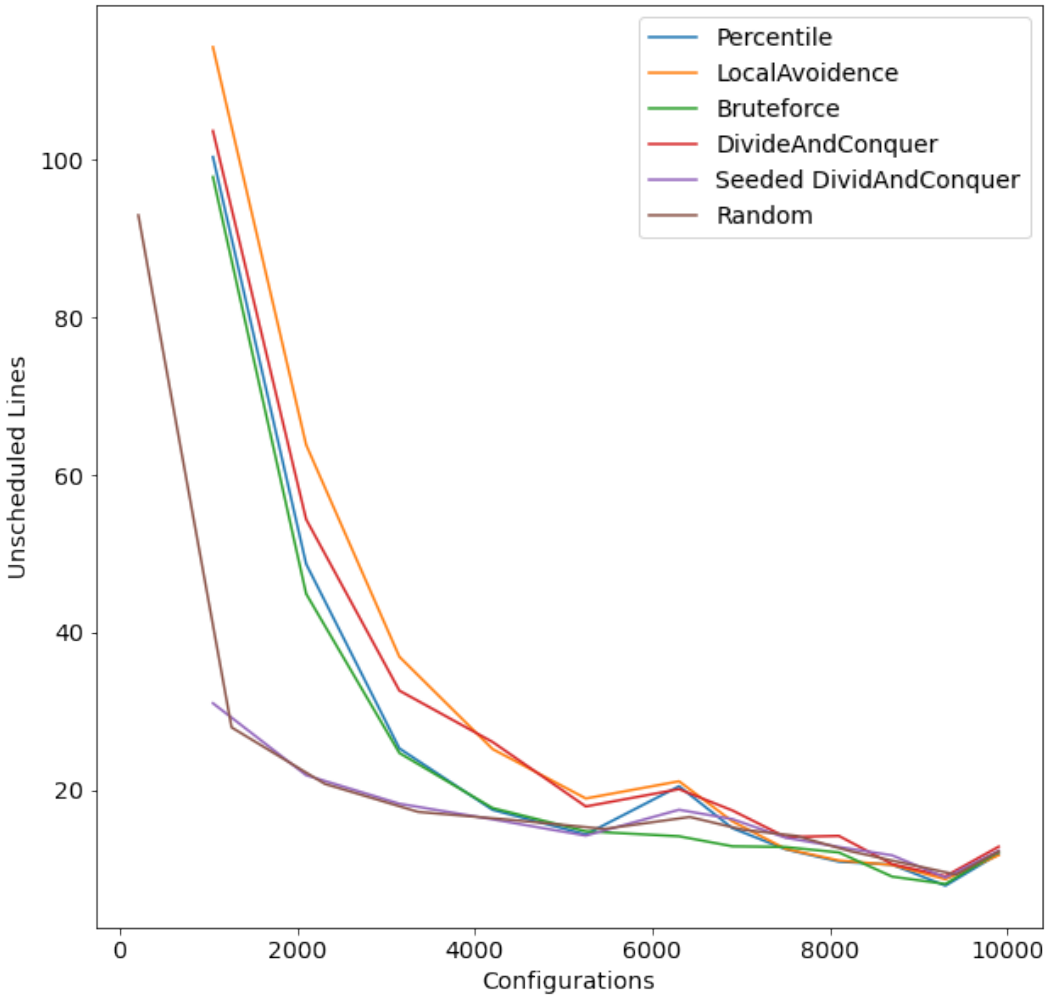

Figure 7.4: Result of the randomized seed for DivideAndConquer on Baden-Württemberg graph.

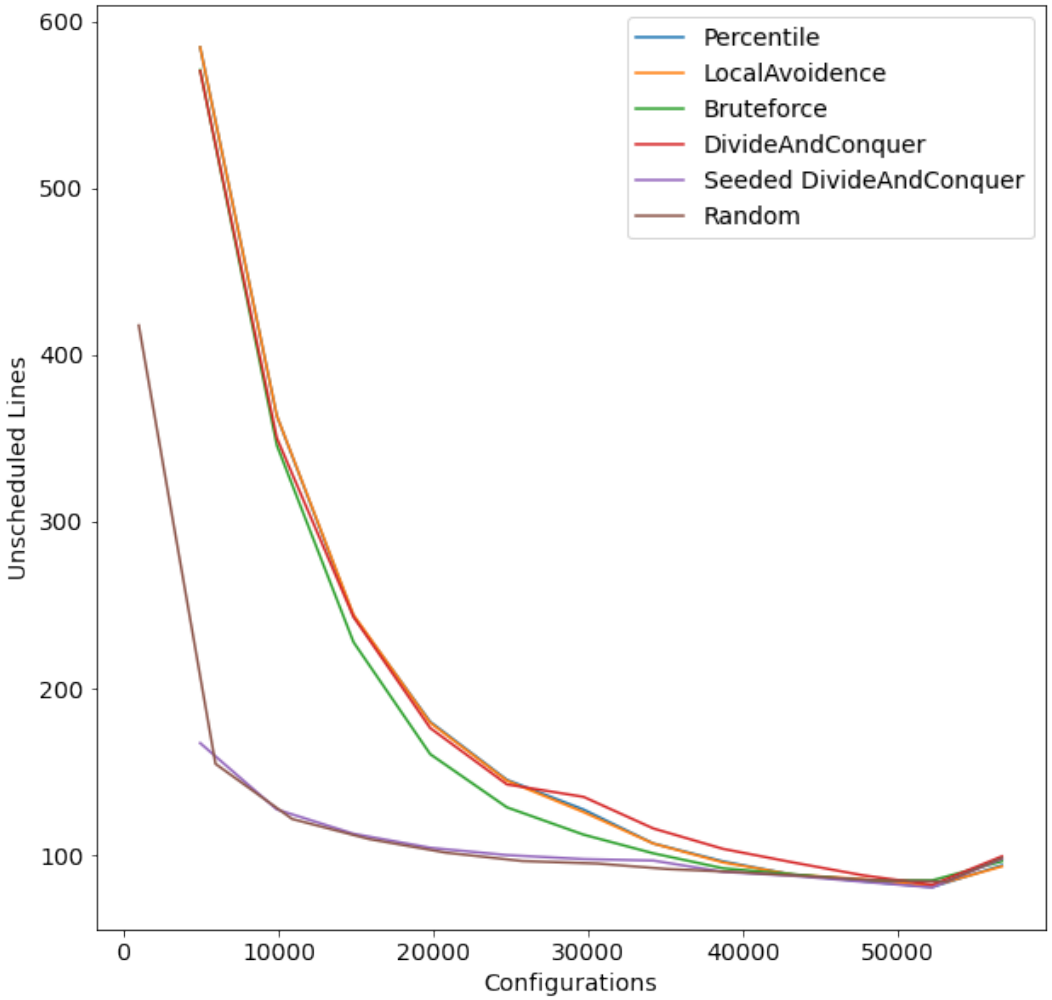

Figure 7.5: Result of the randomized seed for DivideAndConquer on Germany graph.

The results of the modified DiviedeAndConquer strategy do confirm our suspicion that the uniform configuration pattern does cause bad results. If DivideAndConquere is initiated with a few random configurations it has the same results as the Random strategy (cf. Figure 7.5). The first iteration even performing slightly better.

We decided to select the seeded DiviedeAndConquer strategy as the most promising. The disadvantage of the Random strategy is that it can create big time gaps between configurations. This has only very small effects in our setup as the evaluation with different parameters for Bruteforce and Percentile have shown. It is to be expected that this effect increases with the number of configurations per line. This is assumption is supported by the results of Falk et al. [FDR20; FGD+21] as in their work the different parameters did cause a significant difference in the results.

7.3 Line-Select Substrategies

We have to determine the best configuration for the strategies first before we can compare them against each other. We use the seeded DivideAndConquer configuration-select strategy this chapter. This prevents effects from varying configuration-select strategies.

For ConflictCount, InverseConflictCount, and ConflictRatio this is more complex to determine the best configuration than for the configuration-select strategies. The strategies all sort the lines and then add configuration based on the position in this ranking. We have to consider which percentage of lines should get new configurations, how the new configurations are distributed between them, and how many overall configurations should be created. We investigated a linear and an exponential approach with varying percentages of lines receiving new configurations and a different numbers of overall configurations.

The results show that the configurations cause no significant difference in the number of lines that are scheduled. One configuration resulted in a better runtime with only a very small deviation over the runs. This configuration adds 8 configurations to the top 20% of the lines, 4 to the top 40%, and 2 to the top 60%. The performance improvement is likely to cause by the high number of configurations added in each iteration.

Comparing the different strategies shows that the ConflictCount strategy and its variations have negligible effects (cf. Figure 7.6). The bump after around half of the configurations have been created is less pronounced but the overall results are not significantly better. As this bump gets less pronounced the more traffic is handled the impact is not large enough to justify those strategies. The only notable advantage is a mean runtime of about 3/4 of the AllLines strategy (cf. Figure 7.7).

The fact that ConflictCount and InverseConflictCount perform nearly identically shows that the used sorting heuristic is not better than selecting them randomly. The improvement on the bump does indicate that not adding configurations equally to all lines has potential, but a better heuristic has to be found.

All three have no significant differences. We would choose ConflictCount as it is the most simple to implement and also has, because of this, a very slight performance advantage on small networks (Baden-Württemberg size or smaller).

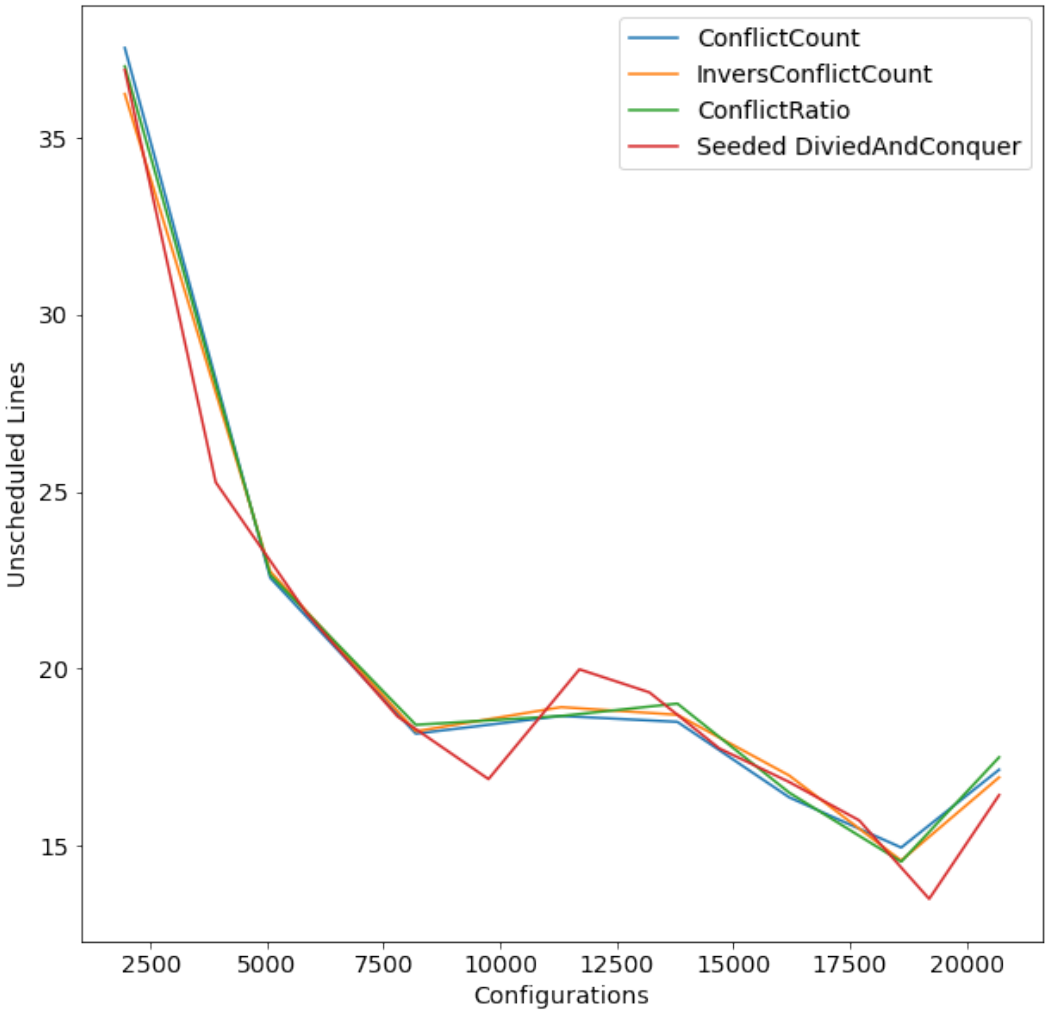

Figure 7.6: Comparison of the different ConflictCount variations on the NRW network.

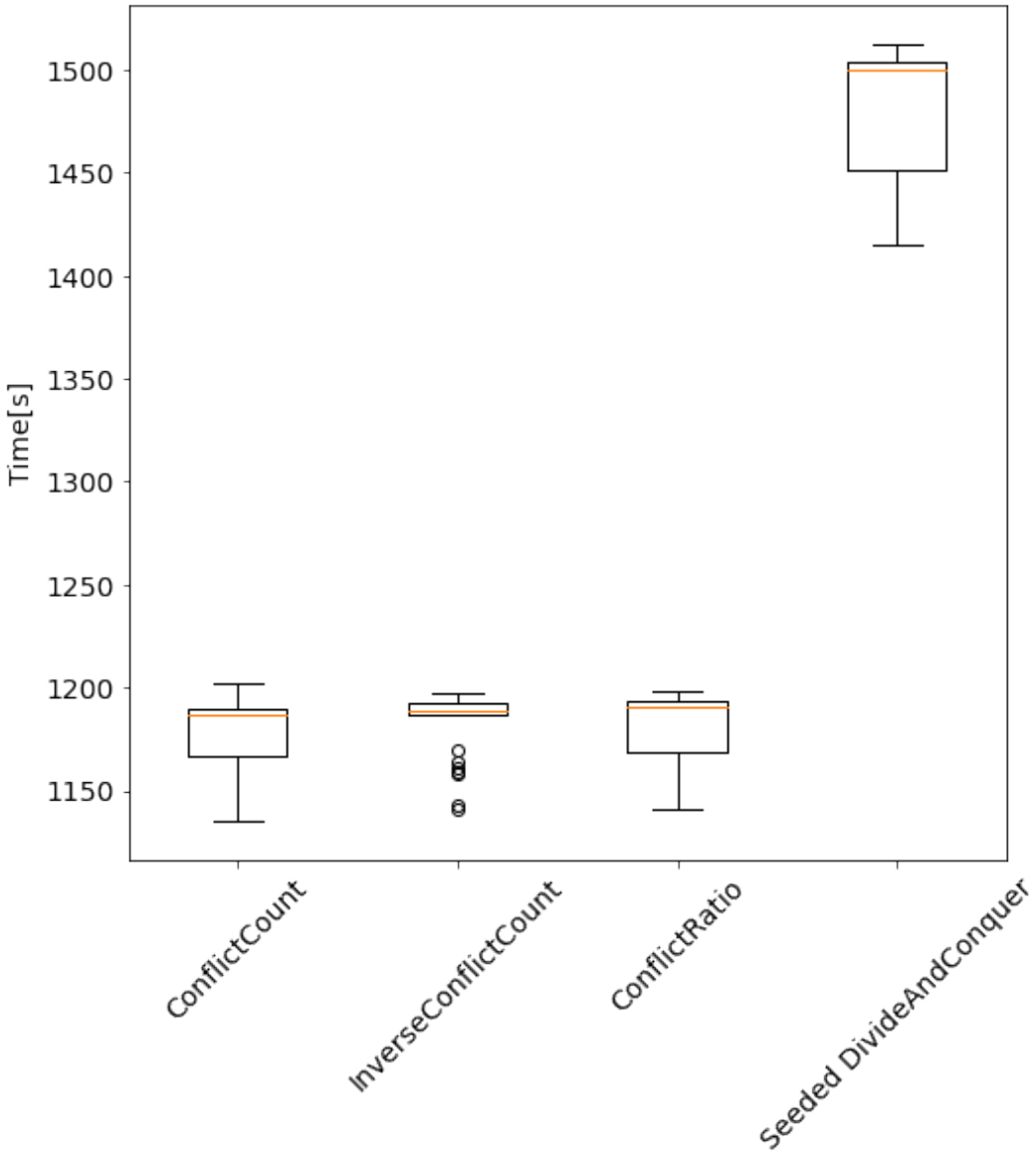

Figure 7.7: Runtime comparison of the ConflictCount variations to seeded DivideAndConquer on the German network.

We run the NotScheduled strategy with and without the rerun mechanism. We expected that the rerun mechanism would cause worse results as the lines which the strategy detects as not scheduled are not selected in a run with equal chances for each line. We ignore the time required for the preparation phase for this comparison, as this phase is identical. For normal traffic, the strategy creates around 1.8 times more configuration when reruns are enabled. This results in around 8 times the runtime (cf. Figure 7.8). This additional cost results on average in 10 more scheduled lines, which is 1% of all lines (cf. Figure 7.9). For low traffic, rerun does schedule 20 lines more, with only 3.5 times the runtime. While for scenarios with low traffic this trade off might be the better choice, we chose to deactivate reruns for this strategy for better scalability.

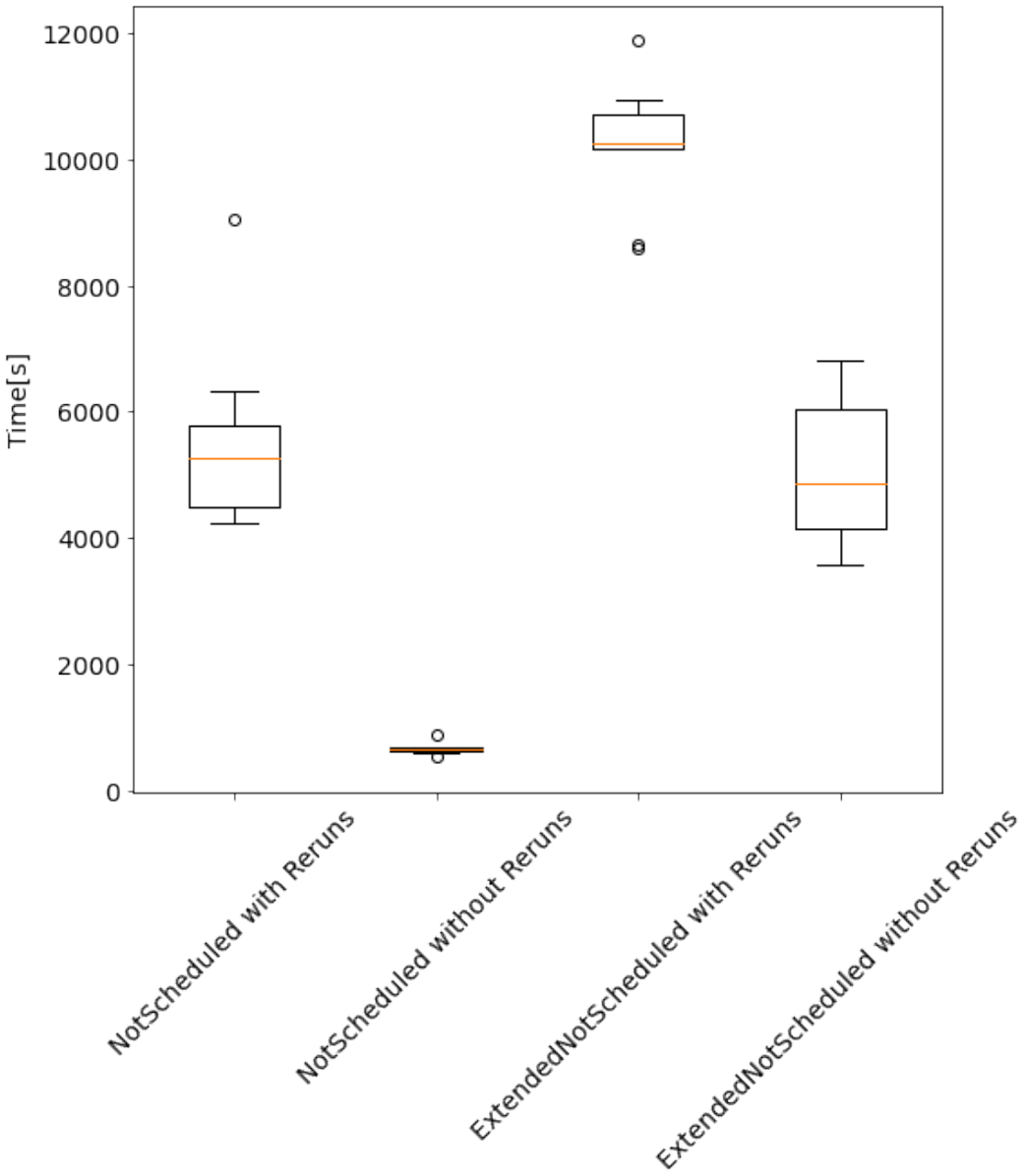

Figure 7.8: Runtime with and without the rerun mechanism on the German network.

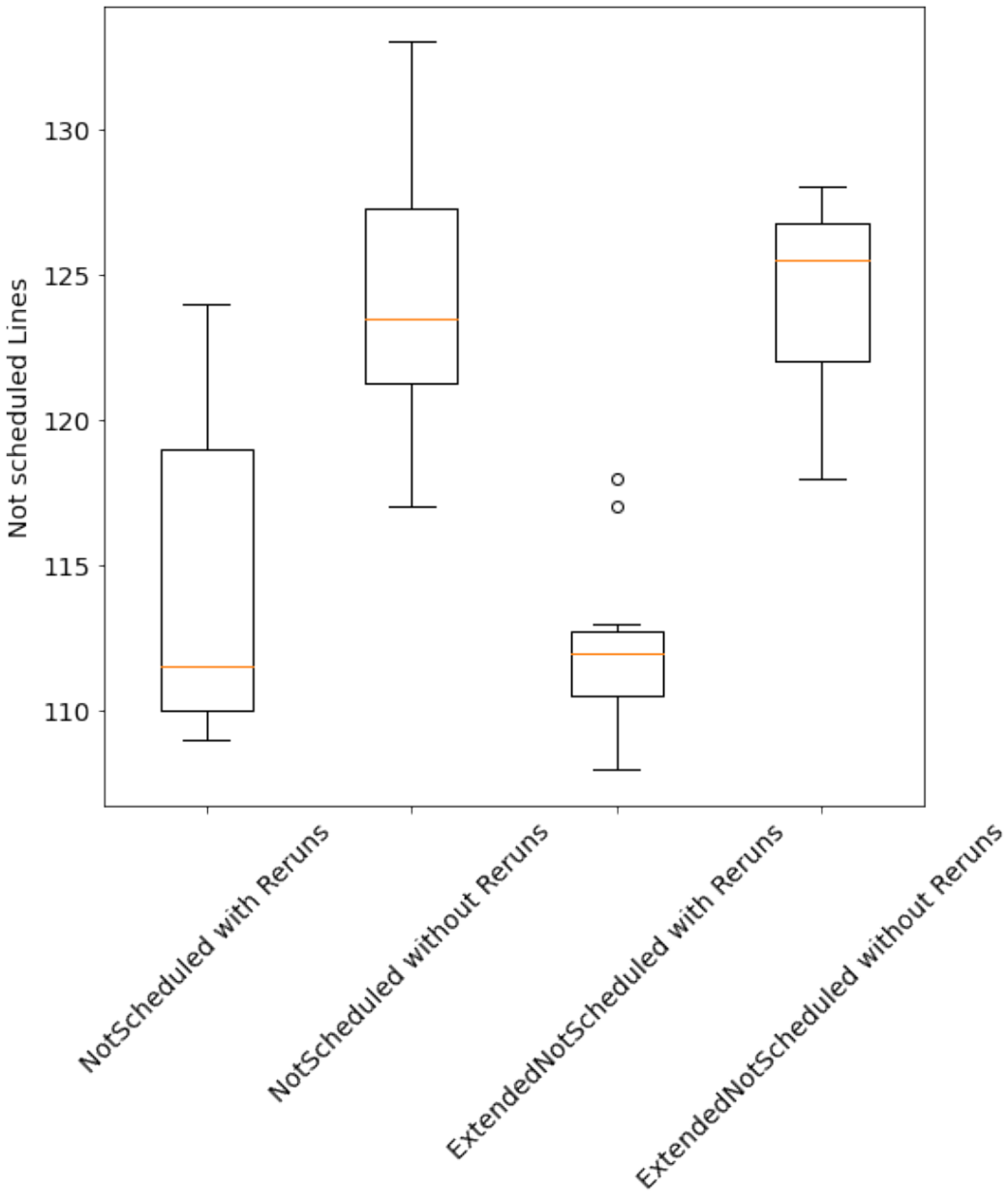

Figure 7.9: The unscheduled lines with and without rerun mechanism on the German network.

We tried to improve further on the NotScheduled strategy with the ExtendedNotScheduled strategy. We evaluated this strategy with and without reruns also. With normal traffic, it builds up nearly the complete conflict graph and still schedules 1-3 lines less than its NotScheduled equivalent. We expected to perform better with low or very low traffic, which is not the case. This means the strategy is worse than NotScheduled in generic cases. It might be worth testing this on data where the lines are distributed more unevenly, but this is out of the scope of this thesis.

If we would only consider the quality of the result the AllLines/seeded DivideAndConquer would be the best Strategy in our evaluations (cf. Figure 7.10). As our main objective was scaleability the best line-select strategy we found is the NotScheduled strategy. It only creates around 20% of the conflict graph and has a runtime of around 2/3 of the ConflictCount variations with normal traffic. The downside is that it typically schedules the least number of lines. On the Germany graph with normal traffic, other strategies could not schedule about 9.3% of the lines while for NotScheduled this value is around 10.5%. It is also very likely that the way we generate our lines is disadvantageous for NotScheduled, as our traffic is evenly distributed. For scenarios where some hot spots have drastically more traffic than other parts of the network NotScheduled might perform even better compared to other Strategies as it can distinguish those regions and build up the conflict graph accordingly.

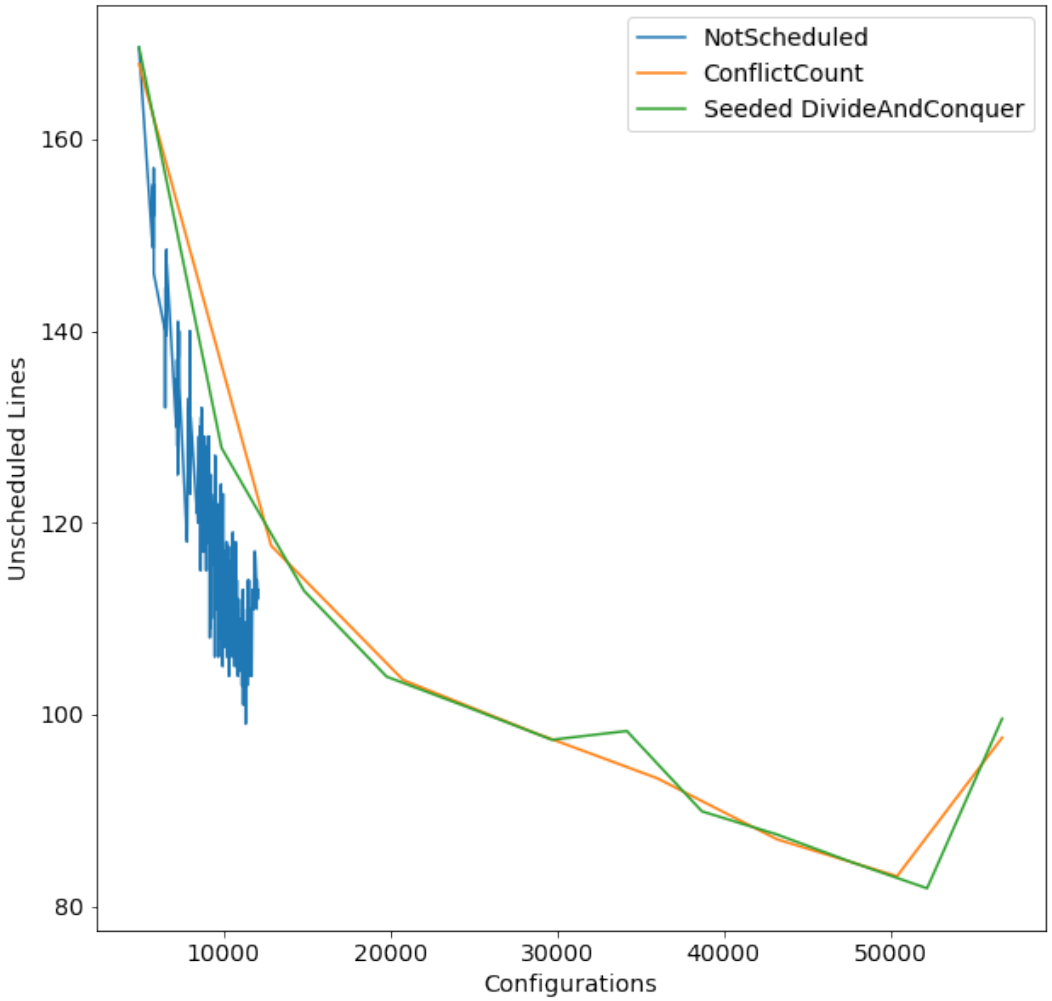

Figure 7.10: Comparison of the unscheduled lines with our best strategies on the German network.

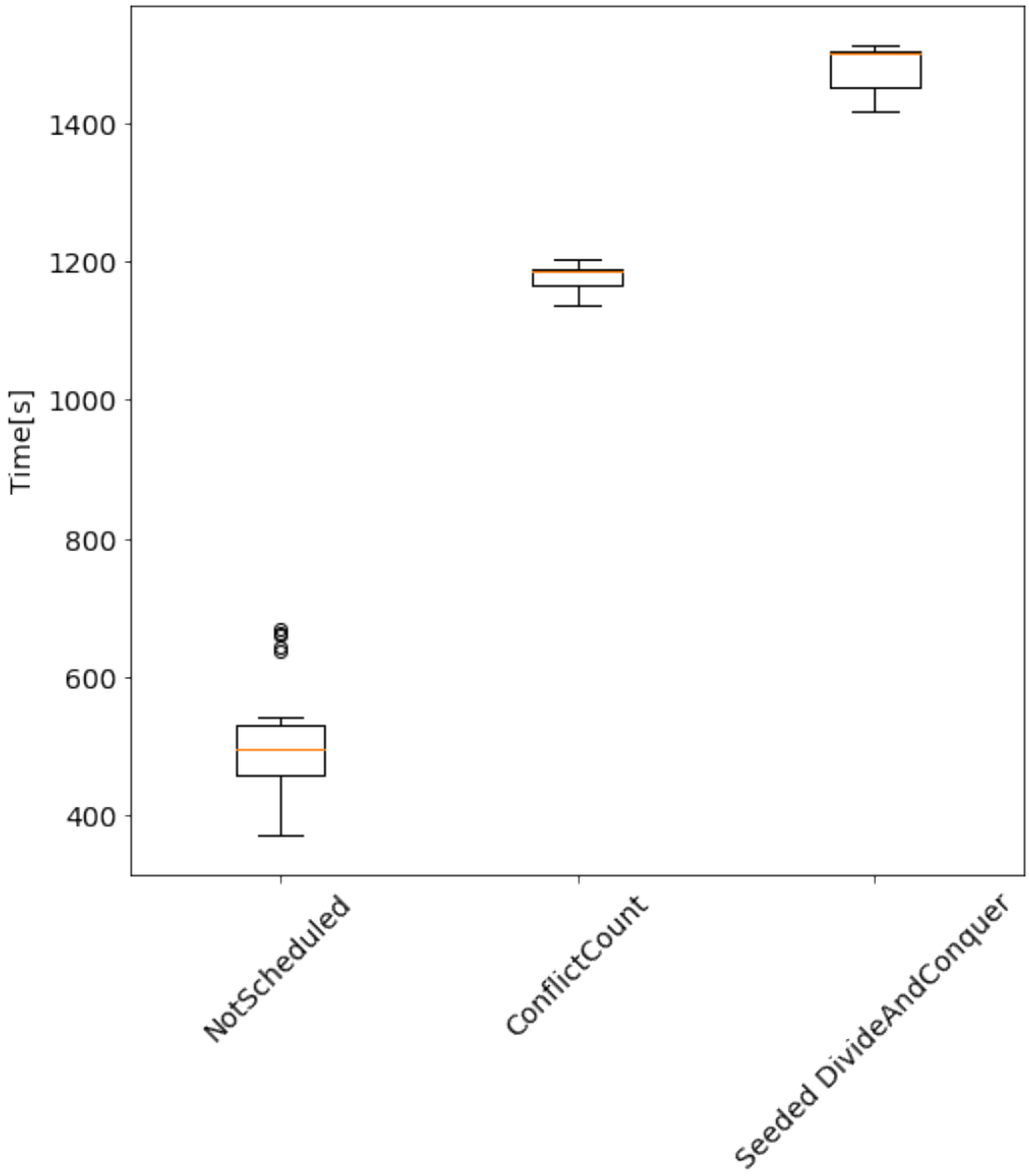

Figure 7.11: Comparison of the runtime with our best strategies on the German network.

7.4 Potential Conflicts

The time to add a new configuration without any optimization depends on the number of existing configurations V, the average length of the routes $\overline{L}$, and the average size of the hypercycles $\overline{H}$ of every pair of lines. This means that the complexity is $O(|V| * \overline{L} * \overline{H})$.Therefore, the time required to extend the conflict graph grows with every iteration significantly. With the Potential Conflict mechanism, the complexity is $O(|PC_v| * \overline{L} * \overline{H})$, where PC_v is set of tuples defining pairs of routes which can conflict with the new configuration. The cardinality of PC_v depends on the density of the lines and the reach of the lines, but it always holds that $PC <= |V|^2$. This is a runtime/memory

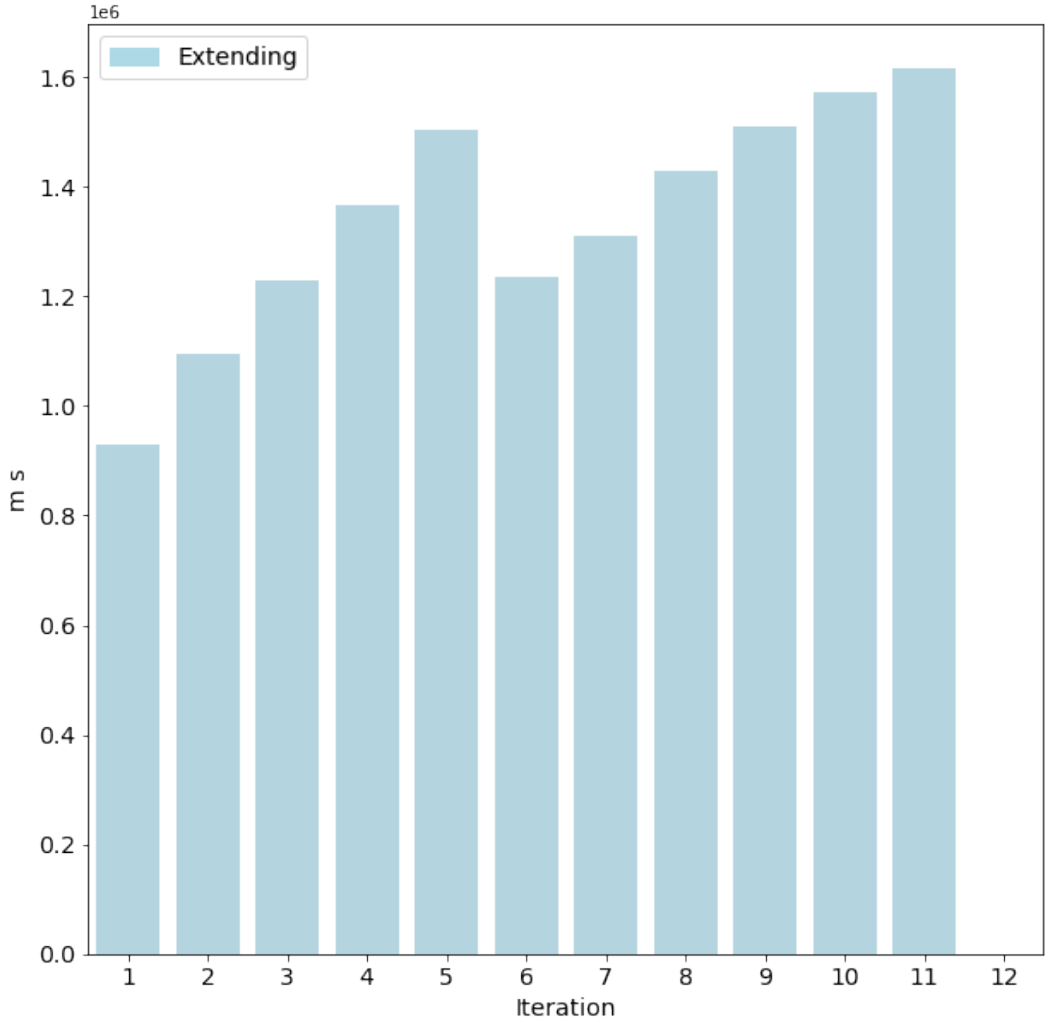

Figure 7.12: Runtime of the extension phase in each iteration for seeded DivideAndConquer on the German network.

trade-off, with a memory usage of $O(PC)$. We expect PC to upper bound in real scenarios, as the network capacity limits the line density. More lines are likely to cover a wider area, resulting a geographic separation with no or few potential conflicts.

The evaluation shows that the time is growth is linear over the iterations (cf. Figures 7.12 and 7.13). The speed up after the fifth iteration is due to the lines with a cycle if 30 minutes being completed. The last iteration has no extension phase as either the graph is built up completely or a complete solution is found.

With 1000 lines there are 1.000.000 possible combinations. Lines can not conflict with themselves and the association is bidirectional which means there are 499.500 combinations of lines which could theoretically have conflicting configurations. On the German network with normal traffic, which consists of 1000 lines, we have 30.986 Potential Conflicts. This means that while extending the graph only around 6.2% of the lines have to be considered (the number of configurations is hard

to estimate as they are not evenly distributed). This has a great impact on the overall runtime as most Potential Conflict checks replace many conflict checks. This is further amplified by the fact that Potential Conflict checks are simpler and thus faster.

The ConflictCount/Seeded DivideAndConquer strategy on the German network with normal traffic takes 13 minutes to finish the preparation phase and 8.5 minutes to extend the conflict graph. As no complete solution exists the conflict graph is built up completely. If we now assume that those 8.5 minutes are 6.2% of the time required without Potential Conflict we can estimate that it would take 2.3 hours, compared to 0.35 hours with Potential Conflicts.

Compared to that the NotScheduled/Seeded DividedAndConquer strategy does only build up around 20% of the conflict graph. It takes 13 minutes for the preparation phase and only 1.7 minutes for extending. With the same estimation, this means we still reduce the time from around 27 minutes to less than 15 minutes.

The mechanism could be refined further by adding additional information. At the moment they only mark the pair of routes which can conflict. It could be beneficial to save the segment and time offset for the first, or for all, conflicting segments to reduce the number of rail segments which have to be iterated. In addition, knowing the last possible segment on which a conflict is possible would allow to end the iteration early. As routs on the German network consist of hundreds of segments a significant improvement on the runtime might be possible.

7.5 Comparison

We now want to assess our results. For this, we have two comparisons. Firstly the result of other papers using conflict graphs and secondly the 24.000 trains on the real network [Deu21b].

Not many papers in the field of railroad scheduling and similar problems contain the data of evaluation results. Of those which do most evaluations handle less than 100 unique trains. The biggest evaluation we found utilizing conflict graphs is from Delorme et al. [DRG01] with 97 unique trains. The results did take between 16 minutes and 166 minutes.

The largest evaluation overall is from Borndörfer et al. (2005) [BGL+06], with 737 unique trains calculated in 3 days. A bit smaller, but likely as scalable, is Borndörfer and Schlechte (2007) [BS08] published a test with 570 unique trains, which was solved in 16 hours.

To get a better comparison we will consider unique trains to be lines. With normal traffic on the German network, we have 1000 lines, which is significantly more around 1.5 more than Borndörfer et al. (2005) [BGL+06]. We solve this in less than 25 minutes which is 0.5% of the time required by them.

Each day 24.000 passenger trains are deployed on the real German railroad network. Our 1000 lines with cycles of 60 minutes or less equal to 29.760 trains in 24 hours (240 lines between state capitals have a cycle of only 30 minutes). This does even cover the additional 2.569 freight trains each day [Deu21b].

This means we reached a level of scalability where real railroad networks can be handled. This shows that our approach scales better than previous approaches and can handle country-sized cases.

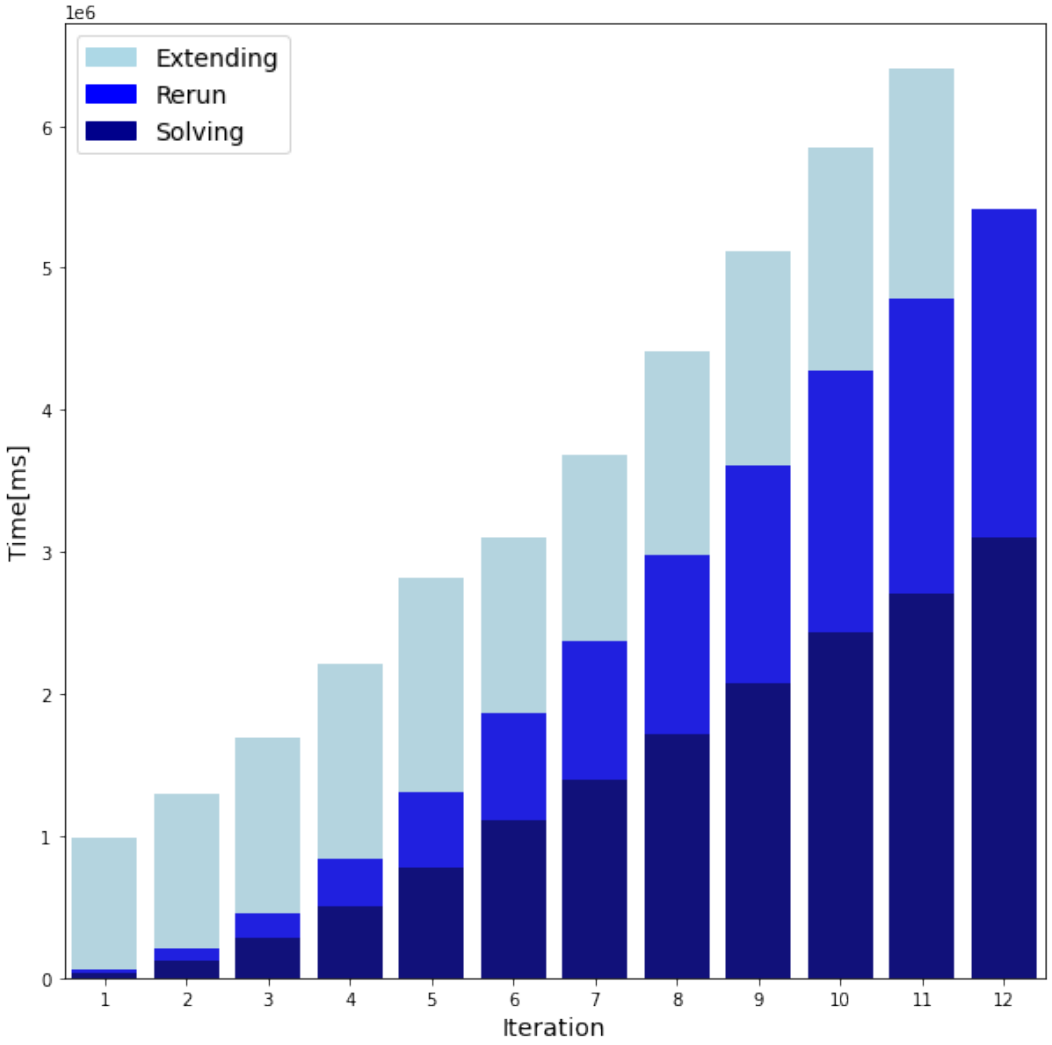

Figure 7.13: Runtime distribution of the three phases over the iterations for seeded DivideAndConquer on the German network. The last iteration has no extending phase as the graph is complete or a complete solution has been found.

8 Future Work

In this chapter, we discuss ideas that came up during the creation of this thesis and have the potential to improve the approach further but were out of scope.

We only implemented strictly periodic lines, but in reality passenger trains often are suspended in the early morning and fewer fright trains are used during the daytime. When implementing this, these limitations could be utilized to further reduce the necessary conflict checks.

Including non-strictly periodic lines might benefit from allowing slight variation for some occasions. This could make it easier to schedule lines that overlap with multiple lines. This would allow a nightly fright train to evade the first and last passenger train when those departures vary slightly from their normal departure time.

One major difference between computer networks and railroad networks is the capacity of links. In computer networks, a link is a single cable and it is very rare that multiple cables connect the same two components. In railroad networks, a link is made up of multiple tracks, which enable multiple trains with the same timing on the link or overtaking. To use this, the current implementation would require the creation of multiple routes for each train. But encoding every possible route on each link would cause a massive amount of routes which in turn would ruin the runtime even for short routes. This could possibly be circumvented by replacing links with abstract representation which encodes the number of tracks and options to overtake in a capacity value. This could allow for a more optimized usage of complex links without sacrificing performance.

Also, connections in computer networks have a source and a destination, in railway networks, there are normally stopovers. This could be implemented as multiple lines which are connected by some restrains. This could improve performance as the segments are smaller and therefore have fewer Potential Conflicts. We have seen this effect on the German network that the time for extending the conflict graph reduced significantly after the lines between the state capitals had all their configurations (cf. Figure 7.12 after iteration 5). As the other lines are mostly spatially separated, they have significantly fewer Potential Conflicts.

Furthermore, the inclusion of alternative routes and selection strategies for them was out of the scope of this paper. This might be relevant for real-world usage and should be looked into.

From an implementation point of view, it could be an interesting task to try to improve on the strategies by parallelizing them. It would be necessary to cache the newly create configurations and check for conflicts between them, but as they are a lot less than those already in the conflict graph after a few iterations the time saving might be worth the overhead.

9 Conclusion

In this thesis, we presented an approach to make conflict graphs viable to solve the joint routing and scheduling problem for country-sized railroad networks. As the creation of conflict graphs is very expensive we verified that small parts of the conflict graph are sufficient to find good solutions, which was already done for computer networks. For this, we present strategies that build the conflict graph dynamically so that good solutions can be found early. We further speed up the creation of the graph by introducing the Potential Conflict mechanism, which drastically reduces the the cost of adding new configurations. The results of our proof-of-concept implementation have shown that the combination of those two optimizations is quite potent. Our approach can handle larger cases than previous works and still be faster. It can schedule traffic equivalent to that of Germany in less than 25 minutes.

Bibliography

[All21] Allianz pro Schiene. *allianz-pro-schiene.de*. 2021. URL: https://www.allianz-pro-schiene.de/glossar/etcs-european-train-control-system/ (cit. on p. 21).

[Bar21] J. Bartlitz. *tagesschau.de*. 2021. URL: https://www.tagesschau.de/wirtschaft/unternehmen/bahn-dhl-postzuege-101.html (cit. on p. 13).

[BEF+21] R. Borndörfer, T. Eßer, P. Frankenberger, A. Huck, C. Jobmann, B. Krostitz, K. Kuchenbecker, K. Mohrhagen, P. Nagl, M. Peterson, M. Reuther, T. Schang, M. Schoch, H. Schülldorf, P. Schütz, T. Therolf, K. Waas, S. Weider. "Deutsche Bahn Schedules Train Rotations Using Hypergraph Optimization". In: *INFORMS Journal on Applied Analytics* 51.1 (Feb. 2021), pp. 42–62. ISSN: 0092-2102, 1526-551X. DOI: 10.1287/inte.2020.1069. URL: http://pubsonline.informs.org/doi/10.1287/inte.2020.1069 (visited on 06/15/2021) (cit. on pp. 13, 19).

[BGL+06] R. Borndörfer, M. Grötschel, S. Lukac, K. Mitusch, T. Schlechte, S. Schultz, A. Tanner. "An Auctioning Approach to Railway Slot Allocation". In: *Competition and Regulation in Network Industries* 1.2 (June 2006), pp. 163–196. ISSN: 1783-5917, 2399-2956. DOI: 10.1177/178359170600100204. URL: http://journals.sagepub.com/doi/10.1177/178359170600100204 (visited on 09/14/2021) (cit. on pp. 19, 20, 47).

[BS08] R. Borndörfer, T. Schlechte. "Solving Railway Track Allocation Problems". In: *Operations Research Proceedings 2007*. Ed. by J. Kalcsics, S. Nickel. Vol. 2007. Series Title: Operations Research Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 117–122. ISBN: 978-3-540-77902-5 978-3-540-77903-2. DOI: 10.1007/978-3-540-77903-2_18. URL: http://link.springer.com/10.1007/978-3-540-77903-2_18 (visited on 09/14/2021) (cit. on pp. 19, 20, 47).

[CB] G. J. Chaitin, P. O. Box. "REGISTER ALLOCATION & SPILLING VIA GRAPH COLORING". In: (), p. 8 (cit. on p. 15).

[CCT08] V. Cacchiani, A. Caprara, P. Toth. "A column generation approach to train timetabling on a corridor". In: *4OR* 6.2 (June 2008), pp. 125–142. ISSN: 1619-4500, 1614-2411. DOI: 10.1007/s10288-007-0037-5. URL: http://link.springer.com/10.1007/s10288-007-0037-5 (visited on 09/14/2021) (cit. on p. 19).

[CD07] S. Cornelsen, G. Di Stefano. "Track assignment". In: *Journal of Discrete Algorithms* 5.2 (June 2007), pp. 250–261. ISSN: 15708667. DOI: 10.1016/j.jda.2006.05.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S1570866706000475 (visited on 07/27/2021) (cit. on p. 20).

[Cha04] G. Chaitin. "Register allocation and spilling via graph coloring". In: *ACM SIGPLAN Notices* 39.4 (Apr. 2004), pp. 66–74. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/989393.989403. URL: https://dl.acm.org/doi/10.1145/989393.989403 (visited on 11/23/2021) (cit. on p. 15).

[DB 14] DB Netz AG. *deutschebahn.com*. 2014. URL: https://www.deutschebahn.com/resource/blob/1303328/d9556ec0c860abb53cf07bfcb693f79d/Anhang_Themendienst_ETCS-data.pdf (cit. on pp. 21, 22).

[DB 16] DB Netz AG. *deutschebahn.com*. 2016. URL: https://data.deutschebahn.com/dataset.groups.datasets.html (cit. on p. 33).

[DB 18] DB Netz AG. *deutschebahn.com*. 2018. URL: https://fahrweg.dbnetze.com/resource/blob/4119016/461729e9fed0107df85271ba1bbddf8b/etcsbroschuere_2018-data.pdf (cit. on pp. 21, 22).

[Deu21a] Deutsche Bahn AG. *deutschebahn.com*. 2021. URL: https://gruen.deutschebahn.com/de/strategie (cit. on p. 13).

[Deu21b] Deutsche Bahn AG. *deutschebahn.com*. 2021. URL: https://www.deutschebahn.com/resource/blob/6066940/3d1c3864381befc7b3f3ea7b9a675922/DuF2020-data.pdf (cit. on pp. 13, 20, 33, 47).

[Die21a] Die Grünen. *gruene-bundestag.de*. 2021. URL: https://www.gruene-bundestag.de/themen/mobilitaet/gruene-strategie-fuer-eine-starke-bahn (cit. on p. 13).

[Die21b] Die Grünen. *gruene-bundestag.de*. 2021. URL: https://www.gruene-bundestag.de/themen/mobilitaet/5-punkte-plan-fuer-die-deutsche-bahn (cit. on p. 13).

[DRG01] X. Delorme, J. Rodriguez, X. Gandibleux. "Heuristics for railway infrastructure saturation". In: *Electronic Notes in Theoretical Computer Science* 50.1 (Aug. 2001), pp. 39–53. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)00164-1. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066104001641 (visited on 07/26/2021) (cit. on pp. 20, 47).

[EU21] EU. *europa.eu*. 2021. URL: https://www.era.europa.eu/activities/european-rail-traffic-management-system-ertms_en (cit. on p. 21).

[FDR20] J. Falk, F. Durr, K. Rothermel. "Time-Triggered Traffic Planning for Data Networks with Conflict Graphs". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). Sydney, Australia: IEEE, Apr. 2020, pp. 124–136. ISBN: 978-1-72815-499-2. DOI: 10.1109/RTAS48715.2020.00-12. URL: https://ieeexplore.ieee.org/document/9113114/ (visited on 06/15/2021) (cit. on pp. 13, 15–18, 20–22, 26, 27, 29, 30, 38).

[FGD+21] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. "Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows with Conflict Graphs". In: *arXiv:2105.01988 [cs]* (May 5, 2021). arXiv: 2105.01988. URL: http://arxiv.org/abs/2105.01988 (visited on 06/15/2021) (cit. on pp. 13, 15–18, 20–22, 26, 27, 29, 30, 38).

[KEZ97] L. G. Kroon, H. Edwin Romeijn, P. J. Zwaneveld. "Routing trains through railway stations: complexity issues". In: *European Journal of Operational Research* 98.3 (May 1997), pp. 485–498. ISSN: 03772217. DOI: 10.1016/S0377-2217(95)00342-8. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377221795003428 (visited on 07/06/2021) (cit. on p. 19).

[LLER11] R. M. Lusby, J. Larsen, M. Ehrgott, D. Ryan. “Railway track allocation: models and methods”. In: *OR Spectrum* 33.4 (Oct. 2011), pp. 843–883. ISSN: 0171-6468, 1436-6304. DOI: 10.1007/s00291-009-0189-0. URL: http://link.springer.com/10.1007/s00291-009-0189-0 (visited on 06/15/2021) (cit. on pp. 13, 19, 33).

[LLRE11] R. Lusby, J. Larsen, D. Ryan, M. Ehrgott. “Routing Trains Through Railway Junctions: A New Set-Packing Approach”. In: *Transportation Science* 45.2 (May 2011), pp. 228–245. ISSN: 0041-1655, 1526-5447. DOI: 10.1287/trsc.1100.0362. URL: http://pubsonline.informs.org/doi/abs/10.1287/trsc.1100.0362 (visited on 09/09/2021) (cit. on p. 20).

[NBT+19] A. Nasrallah, V. Balasubramanian, A. Thyagaturu, M. Reisslein, H. ElBakoury. “TSN Algorithms for Large Scale Networks: A Survey and Conceptual Comparison”. In: *arXiv:1905.08478 [cs]* (June 19, 2019). arXiv: 1905.08478. URL: http://arxiv.org/abs/1905.08478 (visited on 10/23/2021) (cit. on p. 13).

[NTA+] A. Nasrallah, A. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, H. ElBakoury. “Ultra-Low Latency (ULL) Networks: A Comprehensive Survey Covering the IEEE TSN Standard and Related ULL Research”. In: (), p. 60 (cit. on p. 13).

[Pac16] J. Pachl. *Systemtechnik des Schienenverkehrs*. Wiesbaden: Springer Fachmedien Wiesbaden, 2016. ISBN: 978-3-658-12985-9 978-3-658-12986-6. DOI: 10.1007/978-3-658-12986-6. URL: http://link.springer.com/10.1007/978-3-658-12986-6 (visited on 10/26/2021) (cit. on pp. 21, 22, 25).

[PGRR] L. Philipp, L. Ganter, M. Richter, L. Ronsch. “Use-case oriented Extraction and Processing of Open Street Map Data”. In: (), p. 13 (cit. on p. 25).

[PY15] Po-Ya Hsu, Yao-Wen Chang. “Non-stitch triple patterning-aware routing based on conflict graph pre-coloring”. In: *The 20th Asia and South Pacific Design Automation Conference*. 2015 20th Asia and South Pacific Design Automation Conference (ASP-DAC). Chiba, Japan: IEEE, Jan. 2015, pp. 390–395. ISBN: 978-1-4799-7792-5. DOI: 10.1109/ASPDAC.2015.7059036. URL: http://ieeexplore.ieee.org/document/7059036/ (visited on 11/23/2021) (cit. on p. 15).

[Rei21] M. Reichert. *openrailwaymap.org*. 2021. URL: https://openrailwaymap.org/ (cit. on p. 23).

[Süd21] Süddeutsche Zeitung. *www.sueddeutsche.de*. 2021. URL: https://www.sueddeutsche.de/auto/bahn-gueterverkehr-1.5199534 (cit. on p. 13).

[Tag21] Tagesschau. *tagesschau.de*. 2021. URL: https://www.tagesschau.de/inland/stuttgart21-chronologie-101.html (cit. on p. 13).

[Wit21] M. Wittler. *spiegel.de*. 2021. URL: https://www.spiegel.de/auto/wahlkampf-wie-die-parteien-die-deutsche-bahn-staerken-wollen-a-dda38256-4801-4a08-9839-4675f99725ce (cit. on p. 13).

[ZKR+] P. J. Zwaneveld, L. G. Kroon, H. E. Romeijn, M. Salomon, S. Dauzere-Peres. “Routing Trains Through Railway Stations Model Formulation and Algorithms^”. In: (), p. 15 (cit. on p. 19).

All links were last followed on November 18, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature