Bachelor of Computer Science — Graduation project

Universiteit van Amsterdam

Selecting sets of disjoint paths in a railway graph

V.S. Jaddoe

22nd June 2005

Student number: 0222518
E-mail address: `vjaddoe@science.uva.nl`

Supervisor: Hans Bodlaender (Utrecht University)

Signed: Inge Bethke (Universiteit van Amsterdam)

Abstract

A procedure to find an as large as possible set of disjoint paths in a graph at runtime can be useful to maintain railway traffic flow in a large station. This can be done by mapping trains to routes in such a way, that as much as possible trains can be simultaneously routed through the station.

A railway infrastructure can be modelled as a graph where switches and crossings are taken into account. When switches and crossings are represented by vertices, pathfinding algorithms may return paths which are, due to the physical construction of switches and crossings, impossible. To solve this problem, the double vertex graph method has been chosen to model a railway infrastructure as a graph.

Preprocessing has been used to optimise calculation speed at runtime. All possible paths between two signals, and for each path, a maximum speed, priority and the set of conflicting paths are calculated in advance.

The problem of finding a set of disjoint paths for a given set of trains that is as close as possible to an optimal set, has been formulated as the problem of finding the maximum independent set in a graph. To solve this problem, an independent set selection heuristic has been used in conjunction with an instance reduction algorithm.

A discrete event simulator has been used to test both procedures and to simulate the train service, the actual state of the infrastructure and the mapping of trains to routes at an existing railway station with an existing schedule.

Using this simulator, it has been shown that the developed method is able to independently map a set of trains to a set of routes, which has been manually marked as the best set beforehand, within the time specified.

Contents

Preface

Some people wondered why a Computer Science student would choose a graduation project related to the railways, and not a subject in the more popular fields, such as computer vision and distributed computing.

As a daily traveller between my city of residence and the university, I am often delayed. As a student, I am taught to be critical. That means I am also critical when a train traffic controller decides to put my train on hold for ten minutes, causing me to miss my connection.

One day, while standing in the train, between moody commuters and sleepy students, I wondered why we had to wait. Is it true, that there is no room for us at the station? Is it true, that another train crosses ours and that this crossing is unavoidable? What if train A would get route X and train B would get route Y, would that make the waiting time shorter? At that moment the signal went green. Then, while the doors of the 'connecting' train closed right in front of my nose, I decided I should take this subject in consideration when choosing a Bachelor graduation project.

During the preliminary investigation on this subject, I discovered that the railways provide an 'almost infinite' source of potential Computer Science projects. Computer Science may help to solve many optimisation problems. A simple 'Google Scholar' or 'ACM Guide' search on 'railways' yields an extensive list of problems from the railways, solved by computer scientists. These projects range from optimal time-table or schedule generation and optimal usage of single track sections, to optimal management of railway crew, modelling and optimisation of train routing and formal verification of safety systems.

These examples affirm that doing a B.Sc graduation project about a railway related subject is not so strange as it sounds.

Acknowledgements

I thank Dr. Hans Bodlaender for his guidance and support and for being so kind to accept the task of being my supervisor. Without Hans, I would not be able to do this project. That would be a great pity!

I would also like to thank Dick Middelkoop, Léanneke Loeve, Leo Lodder and Jarie Potuijt at ProRail for providing track topology and schedule data to make it possible to test my theory in real situations.

CHAPTER 1

Introduction

There are many possible causes for train delays. Some delays happen at large stations with multiple platforms and tracks, where multiple trains may enter or leave the same station simultaneously. Some of these stations are very sensitive to delayed trains. One delayed train may cause other trains to be delayed as well. In a worst case, this might trigger a long lasting domino effect where many subsequent trains get delayed too.

If many trains get delayed at a large station, the traffic controller may not be able to optimally route all delayed trains. Sometimes, trains do not get a path assigned and are forced to stop at a red signal, even if there would be an alternative path available if another train would get an alternative path. The traffic controller may not have the time to quickly find and set these alternative paths.

A Decision Support System may aid the traffic controller in finding out in real-time which train should be assigned to what path, in such a way that as much trains as possible get a path assigned and get *safely* routed through the railway station. This thesis might be regarded as a continuation of some of the work done by Zwaneveld et al.[5].

1.1 Train routing

In The Netherlands, trains are routed through a station using predefined (or precalculated [5]) paths. As long as there are no disruptions while executing the schedule (e.g., no infrastructure problems such as broken switches, no delayed trains), these predefined paths are sufficient and no custom paths have to be set manually.

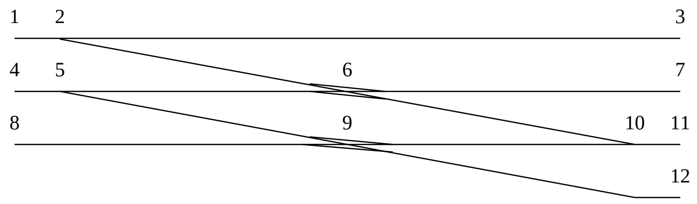

Figure 1.1: Example of a track topology. The crossings at 6 and 9 are in fact double slips (i.e., paths such as $[2, 6, 7]$ and $[5, 6, 10]$ are possible).

Consider the track topology in Figure 1.1 and the scheduled situation where train A is routed through $[7, 6, 2, 1]$ at $t = 0$. Train B is routed through $[4, 5, 6, 10, 11]$ at $t = 1$. Train C is routed through $[8, 9, 12]$, also at $t = 1$.

Train A and B both share switch 6, but the time difference of one time unit should be assumed to be enough to have two 'sequentially disjoint' paths.

If train A is delayed by two time units, and if train B is delayed by one time unit, both prespecified paths claim the same switch. Fortunately, at $t = 2$, train C released its path, consequently, switch 9 becomes available for train B. After the traffic controller has manually set a new path for train B, both train A and train B may continue without increasing their delays.

Using this example, it has been shown that the predefined paths are not necessarily the optimal paths if a train is delayed. If the traffic controller would not have set a new path for B, train A or train B would have had to wait, despite the fact that there actually *are* disjoint paths available for both trains.

This example is a 'toy example'. If there were more (delayed) trains, and a more complex railway infrastructure, it would be unfeasible to manually try all different possibilities in order to get a large set of disjoint paths. Manually calculating all possible mappings of trains to paths costs time, and it is undesirable if trains would have to wait for the traffic controller to assign an 'optimal' path.

Choosing a mapping of a set of trains to a set of disjoint paths *at runtime*, is the most important subject in this Bachelors' thesis. An aim is to choose a set of disjoint paths that is approaching the optimal set of disjoint paths.

1.2 Track topology representation

The track topology in Figure 1.1 could be interpreted as a graph, where each (single or double slip) switch and each crossing is represented as a vertex, and each track section between two vertices is an edge.

Unfortunately, this graph representation cannot be used to calculate paths between two vertices. Due to the physical construction of railway switches, it is impossible to map a path such as $[1, 2, 6, 5, 9, 10, 6, 7]$ to a train, assuming this train is not going to change direction while driving its path. A regular pathfinding algorithm would find these unwanted paths and would not be able to distinct between the physically possible and physically impossible paths.

The difficulties raised by representing switches and crossings as vertices, will be referenced by 'vertex problem' in the next sections.

1.3 Thesis outline

This Bachelors' thesis consists of five parts.

The first part consists of a short literature review. In this literature review, possible solutions to the vertex problem will be discussed. It will also be pointed out why representing a railway topology as a directed graph is not sufficient to solve the vertex problem.

In a practical situation, decisions for the mapping of trains to paths must be made at runtime; it is unacceptable when trains would be delayed due to the running time of the disjoint path set selection procedure. An important tool to make sure the procedure runs fast enough is preprocessing. A preprocessing algorithm that calculates all paths between two vertices and assigns priorities to paths, will be described in the second part of this thesis.

The third section will cover the procedure to find the 'best' set of disjoint paths, given a set of available paths to choose from. This section also covers an instance reduction algorithm that has been developed to improve both speed and result.

To test the disjoint path set selection procedure, a small discrete event simulator has been implemented. This program simulates train movements and keeps the occupation status of the railway infrastructure. The fourth section describes the details and assumptions made while implementing the discrete event simulator.

The last section in this thesis contains the results of a case study, where the developed method for selecting a set of disjoint paths has been applied to an existing railway station using a truly executed train schedule.

CHAPTER 2

Representing track topologies as graphs

In the Introduction, the vertex problem has been raised: assuming each switch and each crossing is represented as a vertex, and each track section between two vertices is an edge, an off-the-shelf pathfinding algorithm might find impossible paths. An example of an impossible path is shown in Figure 2.1.

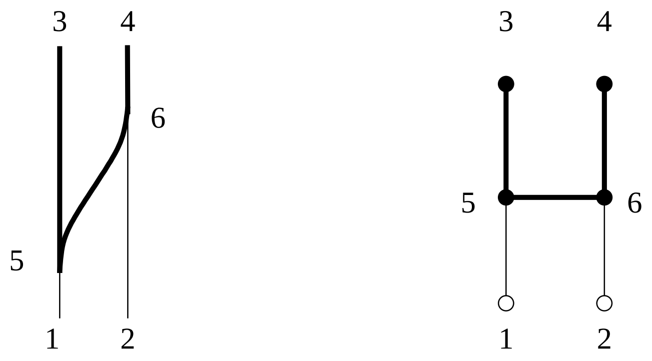

Figure 2.1: A physically impossible path via a switch. Filled vertices and thick edges are reserved by the path.

Assuming that there are no cycles in the railway topology, the most trivial solution to this problem is to model the railway as a directed graph. This solution does solve the impossible path cases for switches and double slips (Figure 2.2).

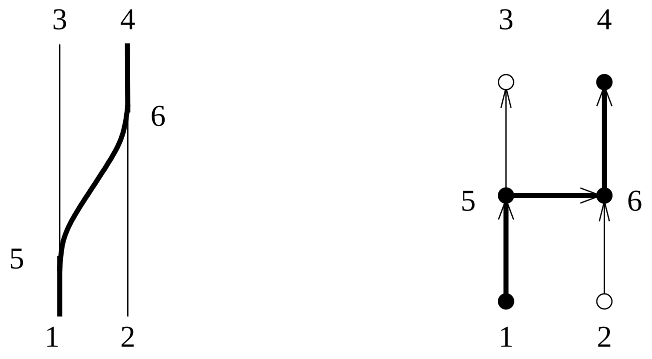

Figure 2.2: For switches, physically impossible paths can be prevented using directed graphs.

Unfortunately, this not-so-trivial problem cannot be solved by a trivial solution. An algorithm using a directed graph for single slips and crossings may return unwanted impossible paths, as shown in Figure 2.3.

Adding a set of restrictions R to the graph $G = (V, E, R)$ might be a possible solution. This set could contain information about the special vertices, or impossible vertex-sequences. For example, in Figure 2.3, $[2, 5, 4]$ and $[1, 5, 3]$ are impossible vertex sequences. These lists can be added to R. Customised versions of

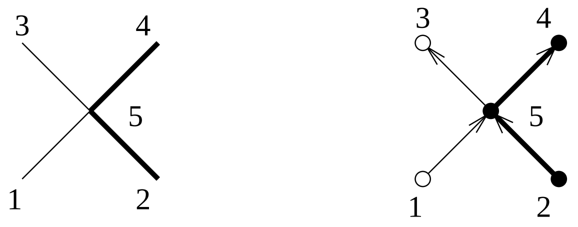

Figure 2.3: A physically impossible path on a crossing.

the regular pathfinding algorithms should take the extra information in R into account to prevent returning impossible paths.

However, this approach does have its disadvantages. By default, the algorithm might calculate all paths, including the impossible ones, and then filter the impossible paths using R. For a large list of impossible vertex-sequences, this could become a slow process.

There are better solutions. In the remainder of this section the results of a very short literature review on this topic will be discussed.

2.1 Decomposition of vertices in a directed graph

It has already been shown why using normal directed graphs only, is not sufficient. There is a method, however, to solve the special case of a crossing in a directed graph. This method is used by Hansen [1] to model railway interlocking systems.

Example 2.1.1. The crossing in Figure 2.3 can be split up in two directed subgraphs:

$$G_/ = \left(\left\{1, 5_/, 4\right\}, \left\{\left(1, 5_/\right), \left(5_/, 4\right)\right\}\right) \tag{2.1}$$

and

$$G_\backslash = \left(\left\{2, 5_\backslash, 3\right\}, \left\{\left(2, 5_\backslash\right), \left(5_\backslash, 3\right)\right\}\right) \tag{2.2}$$

The shared vertex 5 has been split up in two separate vertices $5_/$ and $5_\backslash$. In this model, it is impossible to go from 2 to 4 via $5_/$ or $5_\backslash$.

To prevent the simultaneous reserving of two paths $\left[1, 5_/, 4\right]$ and $\left[2, 5_\backslash, 3\right]$ by train A and train B, there is a symmetric occupancy relation between $5_/$ and $5_\backslash$. If $5_/$ is reserved by train A, then $5_\backslash$ is also reserved by train A. Train B will not be able to reserve the path $\left[2, 5_\backslash, 3\right]$, because $5_\backslash$ is reserved by A.

The set R of graph $G = (V, E, R)$ would contain sets of decomposed vertices. In Example 2.1.1, R would contain $\left\{5_/, 5_\backslash\right\}$. An algorithm for pathfinding should mark all vertices in the same set as occupied.

Using this method to represent railway topologies, the topology in Figure 2.4 can be represented by the graph in Figure 2.5.

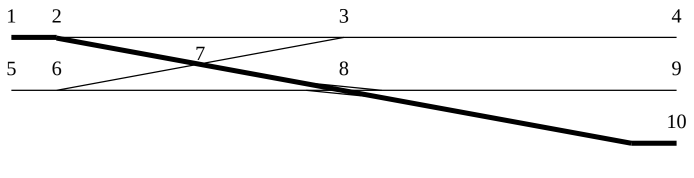

Figure 2.4: An example track topology with switches $\{2, 3, 6\}$, a crossing $\{7\}$, a double slip $\{8\}$, and some track ends $\{1, 5, 4, 9, 10\}$. A path from 1 to 10 via 2, 7 and 8 has been set.

The problem of finding an optimal set of disjoint paths, could then be interpreted in terms of a multicommodity flow network problem, where each train is a commodity and each track element (vertex, edge) has a capacity of exactly one train.

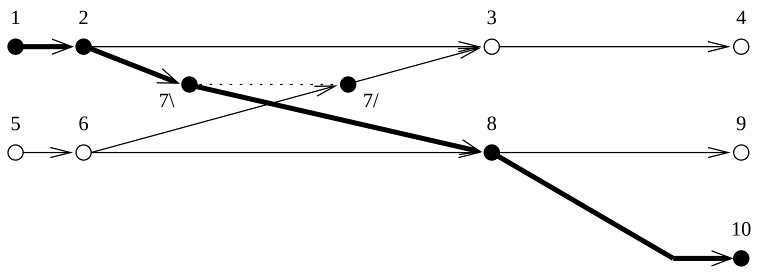

Figure 2.5: Representation of a track topology as a directed graph. The crossing vertex 7 has been decomposed into two vertices $7_/$ and $7_\backslash$. Vertex $7_/$ is also reserved, making a path $[5, 7_/, 3]$ impossible. An alternative way to model the double slip 8 is to decompose it into four switches and a (decomposed) crossing.

2.2 Double vertex graphs

The vertex problem is a direct result of representing switches and crossings as vertices. A method called double vertex graphs [3], where a graph is composed of undirected edges e_i and vertex pairs (v_i, v_i°) instead of single vertices, is suggested by Montigel.

Every vertex v_i must have a 'partner' v_i°. A path in a double vertex graph always has the following form: $\ldots, v_i, v_i^\circ, e_i, v_j, v_j^\circ, e_j, v_k, v_k^\circ, e_k, \ldots$ (i.e., each vertex is visited directly before or after its partner, and after a vertex and its partner have been visited, a normal edge to a vertex from a different pair is used).

Definition 2.2.1 (Double vertex graph). A double vertex graph is defined as a triplet $(V, \circ, \rightarrow)$, where:

$V :$	a finite set of vertices	(2.3)
$\circ : V \mapsto V$	a bijection, joining mapping to map v_i to v_i°	(2.4)
$\rightarrow :\subseteq V \times V$	the edge relation	(2.5)

Using this technique, the track topology in Figure 2.4 can be represented by the double vertex graph in Figure 2.6.

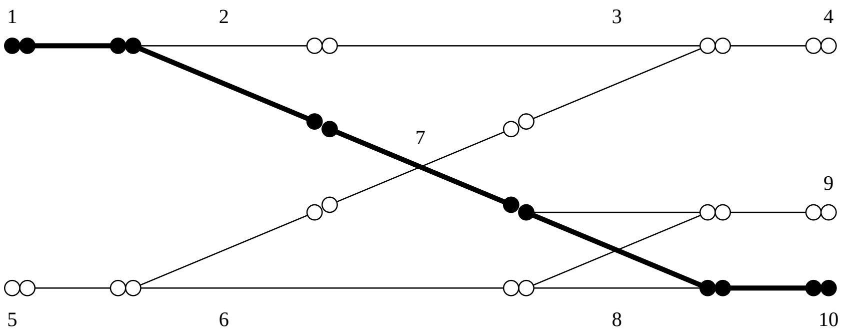

Figure 2.6: Representation of a track topology using the double vertex graph method. It resolves the problem of pathfinding algorithms finding physically impossible paths.

Although the double vertex graph resolves the problem of pathfinding algorithms finding physically impossible paths, it does not implement a good notion of track element occupancy. In Figure 2.6, it is still possible to reserve another path $[5, 6, 7, 3, 4]$, because all vertices required by this path are still available.

To resolve this problem, Montigel defines the formal track topology.

Definition 2.2.2 (Formal track topology). A formal track topology is a double vertex graph $(V, \circ, \rightarrow, \sim)$, extended with an equivalence relation $\sim$, which divides V into equivalence classes.

Definition 2.2.3 (Disjoint paths). Two paths p_1 and p_2 in a formal track topology are disjoint if and only if p_1 and p_2 do not share any equivalence class.

Each equivalence class may contain one or more track elements. A more formal definition of the formal track topology and track elements, can be found in [3].

If a vertex in a track element's equivalence class is reserved, then all other vertices in the same equivalence class are reserved by the path as well. Figure 2.7 shows the track topology of Figure 2.4 modelled as a formal track topology graph.

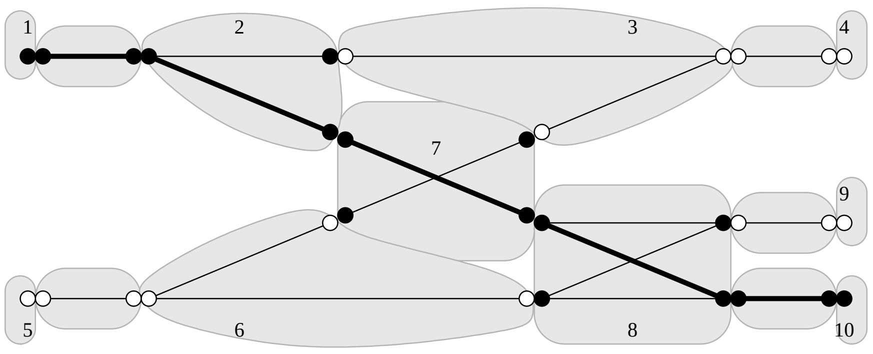

Figure 2.7: Representation of a track topology using the formal track topology method. Each vertex belongs to one track element, marked with the grey coloured areas. If one vertex inside a track element is reserved by a path, then all vertices in the same track element are reserved as well.

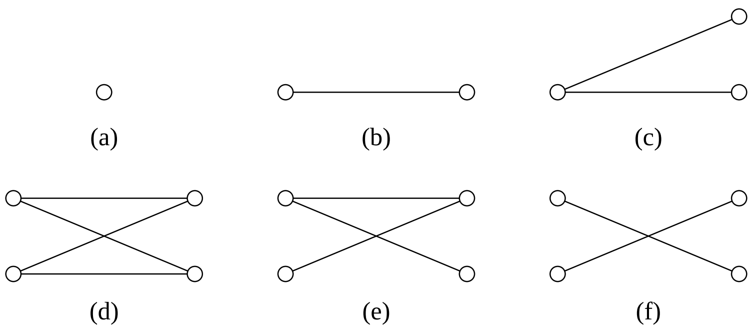

Figure 2.8: An overview of basic track elements in a double vertex graph.

a) track end

b) track section

c) switch

d) double slip

e) single slip

f) crossing

In Figure 2.8, the basic track elements are shown. The topology in Figure 2.7 consist of the basic track elements.

Three-way switches (switches with three legs instead of two) can be modelled using two normal switches (at a very short distance), with all vertices in the same equivalence class, as shown in Figure 2.9.

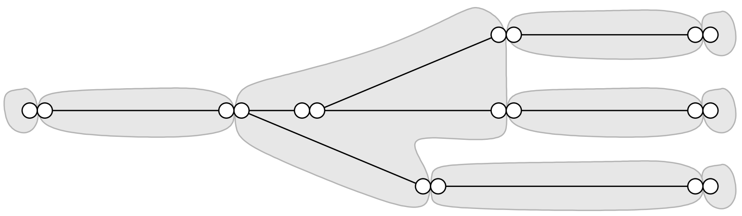

Figure 2.9: A three-way switch modelled as a formal track topology.

2.3 Conclusions

There has to be chosen between vertex decomposition and double vertex graphs as a method to represent railway topologies. Both methods prevent the calculation of physically impossible paths.

Vertex decomposition has not been fully formalised, in contrast to double vertex graphs. The directed graphs used by the vertex decomposition method is against the ‘nature’ of the bidirectional railways, while double vertex graphs provide a good solution for paths in both directions.

Furthermore, the double vertex graph has proved itself in railway simulation software such as OpenTrack[2].

Hence, the double vertex graph method was chosen to represent railway topologies.

CHAPTER 3

Pathfinding and preprocessing

In order to improve the performance of the path selection procedure, preprocessing has been used to do CPU-intensive work in advance. The procedure is used to process a formal track topology by finding and storing paths and to calculate and store metadata for each calculated path.

Assignment of a path to a train should be done fast, we do not want trains to *wait* for the selection procedure to return the best path. According to the 'time-memory trade off', the choice for a faster execution speed will cost more memory. It was assumed there is always enough free memory available to store the information generated by the preprocessing algorithm.

Generally, preprocessing is only done once, after a change of the physical track layout. Since a physical railway infrastructure is not changed very often, a high execution speed of the preprocessing algorithm is not essential.

3.1 Signal vertices

It is unnecessary to find all paths between $V \times V$, because most paths do not make any sense (e.g., paths from a vertex in a switch track element to a vertex in a crossing track element). That is why the preprocessor only calculates routes.

Definition 3.1.1 (Route). A route is a path where the first vertex and the last vertex are both signal vertices.

There are three signal vertex types:

a departure-only signal A route always starts in a departure signal vertex. This signal type is mostly used at the start of a railway track.

a combined arrival/departure signal A route may start or end in this vertex type.

an arrival-only signal A route always ends in an arrival signal vertex. There is no physical light signal attached to this vertex, because it is impossible to depart from an arrival signal. This signal type is mostly used at track ends (mostly buffer stops).

Furthermore, each signal has a direction to distinct between the bidirectional routes in the double vertex graph. In Figure 3.1, the usage of all signal types is shown.

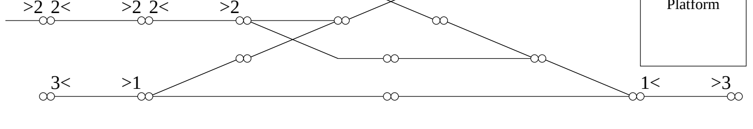

Figure 3.1: The usage of departure-only, combined arrival/departure and arrival-only signals at a small terminus station.

3.2 Preprocessing steps

The preprocessing procedure consists of the following steps:

1. For each pair of signal vertices, store all routes between those vertices in a dictionary data structure. Retrieving an element from a dictionary can be done in $O(1)$ time.
2. For each calculated route r, calculate the set of routes C that share one or more equivalence classes of r. This makes it easier to ensure exclusion of conflicting routes.
3. For each switch vertex in the track topology graph, store the set of paths using this vertex. This makes it easier to exclude all paths using this switch vertex if the switch breaks down.
4. For each calculated route, calculate some useful metadata, such as maximum speed, length and priority.

3.3 Pathfinding algorithm

To find all paths between two vertices in a double vertex graph, Montigel defines Algorithm 3.3.1, based on the depth-first search strategy:

Algorithm 3.3.1: FINDALLPATHS($v_{\text{start}}, v_{\text{end}}$)

```
procedure FINDALLPATHSR($p, v_{\text{end}}$)
  Get last vertex $v$ in path $p$
  $v_{\text{last}} \leftarrow p_{|p|-1}$
  if $v_{\text{last}}.mark = \text{NotOK}$ or $v^{\circ}_{\text{last}}.mark \neq \text{Unknown}$
    then There is no path from $v_{\text{last}}$ to $v_{\text{end}}$, or the algorithm did pass
         from the opposite side
         return
  if $v_{\text{last}} = v_{\text{end}}$
    then $v_{\text{last}}.mark \leftarrow \text{OK}$
         $p$ is a path from $v_{\text{start}}$ to $v_{\text{end}}$, add this solution to the set of solutions
         $solution \leftarrow solution \cup \{p\}$
         return
  if $v_{\text{last}}.mark = \text{Unknown}$
    then $v_{\text{last}}.mark \leftarrow \text{NotOK}$
         $found\_path \leftarrow$ false
         Get all vertices $v_{\text{connected}}$ that are connected to $v^{\circ}_{\text{last}}$ with an edge
         for all $v_{\text{connected}} \in \{\rightarrow \mid v^{\circ}_{\text{last}} \rightarrow v_{\text{connected}}\}$
           do FINDALLPATHSR($p + [v_{\text{connected}}], v_{\text{end}}$)
              if $v_{\text{connected}}.mark = \text{OK}$
                then $found\_path \leftarrow$ true
         if $found\_path$
           then $v_{\text{last}} \leftarrow \text{OK}$
    else Get all vertices $v_{\text{connected}}$ that are connected to $v^{\circ}_{\text{last}}$ with an edge
         for all $v_{\text{connected}} \in \{\rightarrow \mid v^{\circ}_{\text{last}} \rightarrow v_{\text{connected}}\}$
           do The algorithm has already been here, and a solution was found,
              so do not mark any more, but get the solutions already found
              FINDALLPATHSR($p + [v_{\text{connected}}], v_{\text{end}}$)

main
  Mark all vertices in the double vertex graph as Unknown
  for all $v \in V$
    do $v.mark \leftarrow \text{Unknown}$
  Recursively find all paths between $v_{\text{start}}$ and $v_{\text{end}}$
  $solution \leftarrow \emptyset$
  FINDALLPATHSR($[v_{\text{start}}], v_{\text{end}}$)
```

Some programming languages use stack memory for recursion. For very large graphs, it is advised to use an iterative version of this algorithm, because there might be a recursion limit.

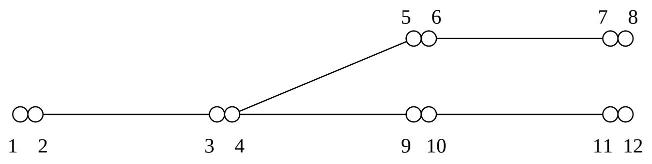

Figure 3.2: In this topology, assuming $1, 2, 7, 8, 11$ and 12 are signal vertices, the following routes are possible: $[1, 3, 5, 7]$, $[8, 6, 4, 2]$, $[1, 3, 9, 11]$, and $[12, 10, 4, 2]$.

In Figure 3.2, some examples of valid routes in a small double vertex graph are given.

3.4 Arranging routes by priority

If an algorithm has to choose between multiple possible routes between two vertices, the algorithm may choose a random route from the collection of available routes. There are good reasons, however, to prefer one route over another route.

1. Maximum speed. Switches have at most two different positions:

   curved The train will be sent to another track. The curve imposes a speed limit, in The Netherlands, most switch curves in an interchange block have a speed limit of 40 km/h or 80 km/h.

   straight The train will stay on the same track, since there is no curve in this switch position, the track section's maximum speed will apply. Symmetric switches do not have a straight position.

   If the route-set selection procedure has to choose between a route with one or more switches in a curved position, or a completely straight route, it should choose for the straight route. In general, the algorithm should choose the route with the highest maximum speed.

2. Claim made on infrastructure capacity. Some routes claim more infrastructure capacity than others. In some cases, there is no reason to choose these routes. These routes are called detour routes[5]. An example of a detour route is given in Figure 3.3.

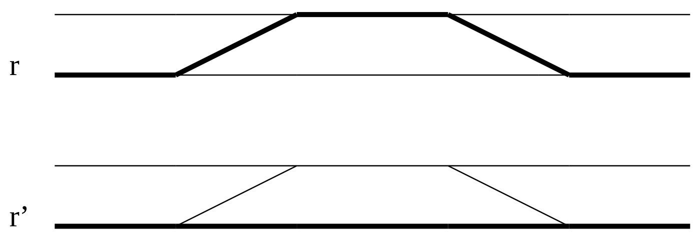

Figure 3.3: A detour route r claiming more capacity than the preferred route r'.

3. Passenger comfort. Passengers –especially those who are standing in a busy commuter train– may prefer a route to or from the platform with the least amount of sharp curves.

   Proposition 3.4.1. *A path with a minimum number of switch track elements is not always a path with a minimum number of switches in a curved position.*

   Proof. This can be proved by an example, as shown in Figure 3.4.

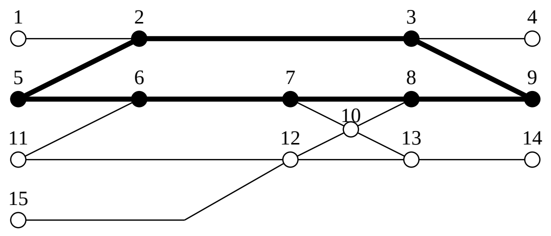

Figure 3.4: In this example, the preferred straight path has more switch track elements than the curved path (which happens to be a detour in this example).

□

3.4.1 Speed limits

When working on an abstract level, such as when modelling railway infrastructures as graphs, there is no notion of the position of a switch. To find out if a path is curvy or not, speed limits have been used.

The maximum speed r_v of a path r is defined as:

$$r_v = \min V_v \tag{3.1}$$

where V_v is the collection of maximum speeds per vertex.

This means that at least all switch vertices in the double vertex graph should contain information about the maximum speed.

Example 3.4.1. The maximum speed of switch vertex 4 in Figure 3.2 depends on the next vertex in the route. If the next vertex is 6, this implies a curved position of switch vertex 4. If the next vertex is 10, it implies a straight position of switch vertex 4.

The speed limit rules in Example 3.4.1 are saved in a dictionary-like data structure S attached to 4. If the curve imposes a speed limit of 40 km/h, this could be stored as:

$$S_{40} : \{6\}$$
$$S_{\infty} : \{10\}$$

The keys in the dictionary are the speed limits S_v, the value attached to each key is the set of vertices where S_v applies. The ∞-symbol is used to denote the track section's maximum speed.

The set notation is used to make sure switches with speed restrictions on both legs of the switch can be modelled as well. If the switch in Figure 3.2 were a symmetric switch, with a speed limit of 80 km/h on both legs, this would be stored as:

$$S_{80} : \{6, 10\}$$

Algorithm 3.4.1 determines the maximum speed of a switch vertex v_c, given the previous vertex v_p and the next vertex v_n in the route. These vertices, and their joined vertices v° are stored in set V. The statement $|V \cap S_v| \geq 1$ checks if one or more of the vertices in V also exist in S_v.

Algorithm 3.4.1: CALCULATEMAXSPEED(v_p, v_n)

```
$V \leftarrow \{v_p, v_p^\circ, v_n, v_n^\circ\}$
$v_c.maxspeed \leftarrow \infty$
$v_c^\circ.maxspeed \leftarrow \infty$
if isSwitch($v_c$) = true
  then for each $S_v \in S$
         do if $|V \cap S_v| \geq 1$
              then $v_c.maxspeed \leftarrow S_v.speed$
  else if isSwitch($c_c^\circ$) = true
  then for each $S_v \in S$
         do if $|V \cap S_v| \geq 1$
              then $v_c^\circ.maxspeed \leftarrow S_v.speed$
```

Proposition 3.4.2. *In a set of routes between two vertices, at most one route is fully straight.*

Proof. Trivial, if there are two fully straight routes between two vertices, there should be a switch to choose between these routes. Since each switch contains by definition at least one curved leg, one of those routes is not fully straight. □

3.4.2 Priority score assignment

The speed limits imposed by curves have been used to assign a priority score $p \in \{x \in \mathbb{R} \mid x \leq 0\}$ to each route.

By default, each route gets the maximum priority score 0. Then, for each switch in the route, a penalty score is subtracted from the initially assigned priority. The penalty score consists of two parts: a fixed penalty of -1, because it is a switch in curved position, and a variable amount depending on the maximum speed of the switch in its current position.

This implies that almost straight routes will get a higher priority than routes with more low-speed switches in curved position.

Assuming a maximum speed of 40 km/h in switch curves and a penalty score of $-1/40$, the curvy route in Figure 3.4 would get a score of -4.1, while the straight route would get 0.

Proposition 3.4.3. *Using this approach, there is at most one route between two vertices with a priority score of* 0*.*

Proof. Trivial, only fully straight routes get a 0 priority score. Since there is at most one fully straight route between two vertices, it is impossible to get two different routes between the same vertices with a 0 priority score. □

The priority score p of a route can be calculated using (3.2). S is a collection of possible switch speed limits, N_s is the number of switches with s as current speed limit.

$$p = \sum_{s \in S} -(\neg\delta^s_\infty + \frac{N_s}{s}) \tag{3.2}$$

$\neg\delta^i_j$ is a negation of the Kronecker delta function:

$$\neg\delta^i_j \equiv \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } i \neq j \end{cases}$$

Theorem 3.4.4. *The route is fully straight if and only if the priority score is* 0*.*

Proof. **Part 1: If the route is fully straight, then the priority score is** 0

$$s = \infty \implies \lim_{s \to \infty} (\neg\delta^s_\infty + \frac{N_s}{s}) = 0$$

Substitution in (3.2):

$$p = -0 = 0$$

Part 2: If the priority score is 0**, then the route is fully straight**

If the priority score is 0, then no penalty scores have been subtracted. A penalty score is only subtracted if there is a switch in curved position. A fully straight route does not have switches in curved position. □

CHAPTER 4

Disjoint route set selection

In this section, a solution to the problem of mapping an as large as possible set of inbound or outbound trains to free routes will be discussed. The procedure to find this as large as possible set of disjoint routes is based on a heuristic, which –despite being a heuristic– still does a good job in finding a large set of disjoint routes.

Furthermore, two methods to optimise the selection of disjoint routes, in terms of both speed and result, will be presented.

In the following example, four trains $t \in [t_0, t_1, t_2, t_3]$ are about to enter or leave a station. The trains are ordered by arrival/departure time. For each train t_i, there is a list of known available routes $r_{i,n}, n \in \mathbb{N}$, ordered by route priority. An overview of available routes can be found in Table 4.1. In this section, a (train or route) priority $i \in \mathbb{N}$ is higher than priority $i + 1$.

train	available routes per train					
t_0	$r_{0,0}$	$\mathbf{r_{0,1}}$	$r_{0,2}$	$r_{0,3}$	$r_{0,4}$	
t_1	$\mathbf{r_{1,0}}$	$r_{1,1}$				
t_2	$r_{2,0}$	$r_{2,1}$	$\mathbf{r_{2,2}}$			
t_3	$r_{3,0}$	$r_{3,1}$	$r_{3,2}$	$r_{3,3}$	$r_{3,4}$	$\mathbf{r_{3,5}}$

Table 4.1: An overview of available routes per arriving/departing train. In this example, the solution, the optimal set of disjoint routes, is the set $\{r_{0,1}, r_{1,0}, r_{2,2}, r_{3,5}\}$.

The next step is to generate a conflict graph. In this graph, each route is a vertex and there is an edge between two vertices if and only if the corresponding routes conflict.

The vertices representing the routes in the same set $(r_{i,n}, n \in \mathbb{N})$ are mutually fully connected subgraphs. This is to prevent the selection of two routes between the same signal vertices in the double vertex graph. An example conflict graph can be found in Figure 4.1.

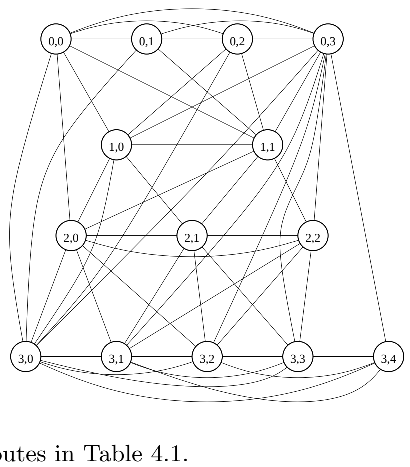

Figure 4.1: A conflict graph for the routes in Table 4.1.

4.1 Independent set calculation

The problem of mapping an as large as possible set of trains to disjoint routes has now been reworded as an independent set problem.

Definition 4.1.1 (Independent set). A set $S \subseteq V$ of a graph $G = (V, E)$ is an independent set if no edges in E connect the vertices in S.

Unfortunately, it can be easily proved that an algorithm to calculate the maximum independent set S is $\mathcal{NP}$-hard, using the clique problem.

4.1.1 Instance reduction

The problem instance, the conflict graph, can be made smaller to improve selection speed. This can be done by removing superfluous vertices from the conflict graph. The following algorithm is devised to remove routes that will never be chosen. The resulting conflict graph will be smaller and can be easily processed by an independent set algorithm or heuristic.

The algorithm can be best described using the example from Figure 4.1.

As can be seen, route $r_{1,1}$ conflicts with all routes from the higher priority train t_0. This route will never be chosen and can thus be removed from the conflict graph.

The next low-priority route is $r_{2,0}$, which conflicts with the only remaining route from train t_1. Again, this route will never be chosen and can thus be removed safely. Other superfluous vertices (ordered by removal) are: $r_{2,1}, r_{3,0}, r_{3,1}, r_{3,2}$ and $r_{3,3}$. Figure 4.2 shows the resulting conflict graph.

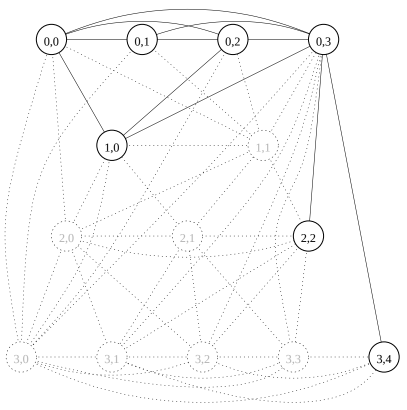

Figure 4.2: The conflict graph from Figure 4.1 after removal of superfluous vertices.

The resulting conflict graph can now be processed by an independent set algorithm or heuristic.

Note. The assignment of priorities to trains is done by assigning the highest priority (0) to the train which is expected to arrive or depart first[1]. If all routes of t_0 conflict with all routes of all other trains in the queue, then t_0 will take precedence and it will be the only train which gets a route assigned.

An algorithm for the 'best' assignment of a priority to a train depends on requirements set by railway companies and passengers, falls outside the scope of this thesis, and is probably a subject within the area of Operational Research.

The full algorithm for instance reduction is shown in Algorithm 4.1.1. In this pseudocode, $|T|$ is the number of trains and $|R_t|$ is the number of routes for train t. The algorithm consists of a main loop, which iterates over each route (minus those of the highest priority train), and a subprocedure which checks if a given route

[1]Since the Dutch Railways try to maximise the departure punctuality, they would probably give departing trains a higher priority than arriving trains.

conflicts with all routes of a certain higher priority train.

Algorithm 4.1.1: ReduceConflictGraph(M)

```
procedure CheckConflicts($r, p$)
  Process all trains with a priority higher than train $t_p$
  for $i = 0$ to $p - 1$
    do
      $c_a \leftarrow$ true
      Check if route $r$ conflicts with all routes of the current higher priority train
      for $j = 0$ to $|R_i|$
        do
          $c_a \leftarrow c_a \wedge$ IsConflicting($(i, j), r$)
          If it does, return true
          if $c_a =$ true
            then return ( true )
  return ( false )

main
  Start processing all routes of train $t_1$
  for $i = 1$ to $|T|$
    do
      for $j = 0$ to $|R_i|$
        do
          Check if route $r_{i,j}$ conflicts with all routes of all higher priority trains
          $c \leftarrow$ CheckConflicts($(i, j), i$)
          If it does, remove this route from the queue
          if $c =$ true
            then RemoveRoute($(i, j)$)
```

Lemma 4.1.1. *The maximum amount of conflict checks c required for n trains t with each train having $|R_t|$ routes to choose from is:*

$$c = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} |R_i| \cdot |R_j| \tag{4.1}$$

Proof. This can be justified by induction.

Base case

Train t_1 has $|R_1|$ routes. For all routes of t_1, all routes of all higher priority trains have to be checked. In this base case, only t_0 is a higher priority train. This means that the amount of conflict checks equals $|R_1| \cdot |R_0|$.

Induction step

The hypothesis is that for all routes of a certain train t_i all routes of all higher priority trains $\{j = 0, \ldots, i-1\}$ have to be checked:

$$c_i = \sum_{j=0}^{i-1} |R_i| \cdot |R_j| \tag{4.2}$$

This means that for the $i+1$-th train the amount of checks is c_{i+1} plus the amount of checks for the previous train c_i:

$$c = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} |R_i| \cdot |R_j|$$

□

Theorem 4.1.2. *Assuming a worst case scenario, with $|T|$ trains each having the same amount of routes $|R|$, the algorithm as presented in this section uses $O\left(|T|^2 |R|^2\right)$ time. A test to check if two routes conflict, can be done in $O(1)$ time using the conflict table produced by the preprocessing procedure.*

Proof. In (4.1) there are two nested iterations over the list of trains. This means the expression inside this double loop, is executed $|T| \cdot |T| = |T|^2$ times.

The expression which is executed $|T|^2$ times is a Cartesian product of two equal lists R. For each pair in $R \times R$, the $O(1)$ conflict check routine is executed. This routine is executed $|R \times R| = |R|^2$ times.

This makes a total running time of $O\left(|T|^2 |R|^2\right)$. □

4.1.2 Route selection heuristic

The following $O(|V|)$ heuristic[4] does a good job solving the independent set problem:

> Locate the vertex with the lowest degree v, add it to S, remove it from V and remove all edges leading to v from E. Repeating this process until the graph is empty gives a maximal independent set.

If there are two or more vertices with the same degree, choose the vertex with the highest priority (first train priority, then route priority).

4.2 Optimisation through partial routes

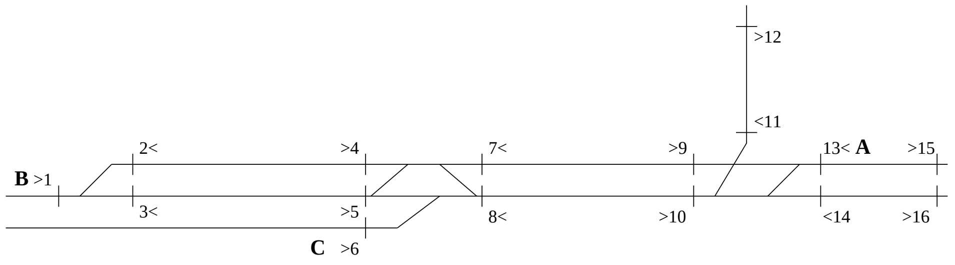

Figure 4.3: Example topology with signals and three trains requesting a route from/to a signal.

In Figure 4.3, a station's track topology is shown, and three trains, A, B and C, about to enter or leave the station.

Train A requests a route from signal 13 to signal 3. Train B requests a route from signal 1 to signal 16 and train C requests a route from signal 6 to signal 12.

Table 4.2 shows the routes for each train at each time step, according to the precalculated route assignment schedule.

time step	train	assigned route
$t = 0$	A	$[13, 7, 3]$ or $[13, 8, 3]$
$t = 1$	B	$[1, 4, 10, 16]$
$t = 2$	C	$[6, 10, 12]$

Table 4.2: Mapping of trains to routes at each time step. Assumption: one time step is long enough to let a train drive to the destination signal and release the claimed route.

Unfortunately, train A is delayed by one time unit. What is the best mapping of trains to routes minimising the delay of all trains as much as possible? Table 4.3 shows the concept of partial routes. If the assignment of the requested route is done in two or more steps, the assigned route in each step is a partial route of the originally requested route.

time step	train	route
$t = 1$	A	$[13, 7]$
$t = 1$	B	$[1, 5, 10, 16]$
$t = 2$	A	$[7, 3]$
$t = 2$	C	$[6, 10, 12]$

Table 4.3: Mapping of two partial routes to delayed train A.

4.2.1 Partial routes heuristic

The design of an exact algorithm would have taken more time than available for this project, hence a heuristic is used instead.

The heuristic will only be called if Algorithm 4.1.1 removed all routes for a certain train. If this happens, all removed routes will be replaced by a list of partial routes.

Example 4.2.1. If the route via signals $[1, 5, 10, 16]$ was removed, then the list (ordered by route length) of partial routes is $[[1, 5, 10], [1, 5]]$.

Remark. Including $[[5, 10, 16], [10, 16], [5, 10]]$ does note make any sense, because the train must get a route from signal 1 to 5 first.

Assuming that for each route, its partial routes have been precalculated by the preprocessing procedure, the list of partial routes can be retrieved in $O(1)$ time.

After all removed routes have been replaced by their partial routes, Algorithm 4.1.1 should be called again. Hopefully, one or more partial routes do not conflict with all routes of higher priority trains, so the train may continue to the next signal.

Of course the train should be rescheduled, because it should get the remainder of the originally requested route.

The actual performance (as in result, not speed) of this heuristic depends on the priority of the delayed trains. In the example at the beginning of this section, train B had a higher priority than the delayed train A. If these priorities were reversed, the assignment of (partial) routes would be different.

CHAPTER 5

Testing using discrete event simulation

To test the algorithms and heuristics to calculate the set of disjoint routes, a discrete event simulator has been used to simulate the scheduled arrival and departure of trains.

In this section, some design and implementation details of this discrete event simulator will be discussed.

5.1 Discrete event simulator tasks

The task of simulating train movements and mapping of trains to routes can be described in terms of a custom 'Producer/Consumer' problem.

- There is exactly one Producer, while there are many Consumers.
- The Producer produces an object, only after a request of a Consumer.
- The Producer is able to produce only one object at a time: if two Consumers request an object at the same time, one of these Consumers must wait.
- The Consumer consumes the object (may take a few minutes) and then notifies the Producer the object has been consumed.
- Only one object of a certain type can be consumed at a time; the Producer cannot produce an object if an equal object is being consumed.

In this simulator, the Producer is a train dispatcher/railway safety system, while the Objects are routes in a station yard and the Consumers are trains requesting routes in the station yard from one signal to another signal.

Since there is only one train dispatcher, the train dispatcher has been modelled as a token: a train requests the token and will only get access to the train dispatcher if no other train is holding the token. After retrieving a route from the train dispatcher, the train should release the token to allow other trains.

After a train has requested a route from the train dispatcher, one of the following three things may happen:

1. The train dispatcher returns the route as requested by the train. If this happens, the train will do the following:
   (a) The train process will now hold for a few seconds (to drive the route).
   (b) After it reached the destination signal, it will notify the train dispatcher to release the route.
   (c) If there are no scheduled route requests, the train will 'cancel' itself.
2. The train dispatcher returns a partial route. If this happens, the train will do the following:
   (a) The train process will now hold for a few seconds (to drive the route).
   (b) After it reached the last signal in the partial route, it will notify the train dispatcher to release the route.

   (c) The train will request the remainder of the route, this will go on until the train reached the destination signal.

3. The train dispatcher was not able to reserve the requested route. The following things will happen:
   (a) The train process will be put in a separate wait queue, the train dispatcher will be available for other trains.
   (b) As soon as the route is available (this might only happen after another train released its route), the train process will be notified by the train dispatcher.
   (c) The train will now re-request the route.

5.2 Implementation details

The implementation of the simulator and the route set selector consist of several modules. In Figure 5.1 an overview of all modules is shown.

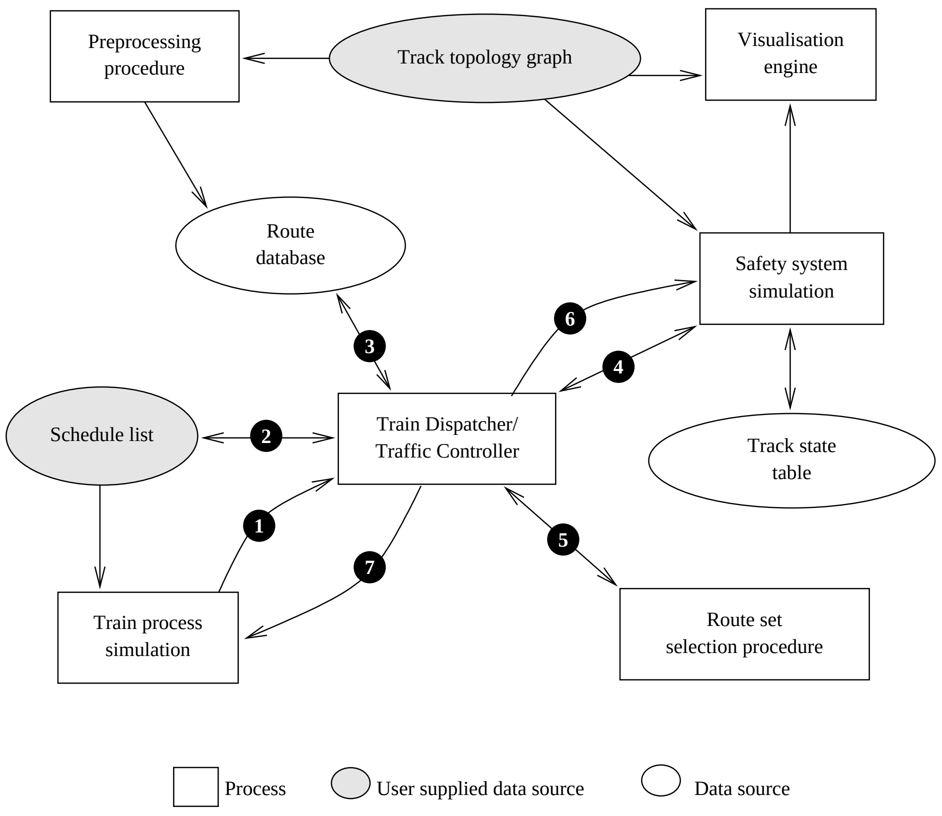

Figure 5.1: Overview of modules used by the simulator.

5.2.1 Train processes

Each train is a separate process (thread) within the simulator environment. Each train process keeps its own schedule, and announces itself at the train dispatcher when it is about to arrive at or depart from the station. After obtaining a route, the process will hold for a certain amount of time to simulate driving the route.

After a train process got through its schedule, the process will suspend.

5.2.2 Route database

The route database is generated by the preprocessor module, and is read-only in the simulation module.

The database contains:

- A set of all possible routes in the formal track topology,

- for each pair of signals a set of all possible routes between these two signals,
- and for each vertex in the double vertex graph a set of all routes using this vertex.

Routes are stored as a Route objects. A Route is composed of one or more Atomic Routes.

Definition 5.2.1 (Atomic route). A route is an atomic route if and only if the only signal vertices in the route are the first and the last vertex.

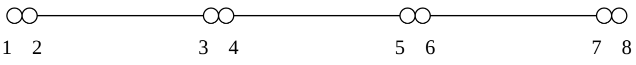

Figure 5.2: A route is composed of one or more atomic routes. In this example, all vertices are signal vertices.

The concept of Atomic Routes has been devised to make calculation and storage of Route objects more efficient. A single Atomic Route and its metadata may be shared by many Route objects. Since only references to Atomic Route objects are stored in Route objects, less storage space is used by the route database.

In Figure 5.2, the route $[1, 3, 5, 7]$ is composed of the following atomic routes: $[[1, 3], [3, 5], [5, 7]]$, while the route $[1, 3]$ is composed of one atomic route: $[[1, 3]]$.

A Route object contains the following information:

- A list of partial routes,
- a list of vertices in the route,
- a list of atomic routes that compose the route,
- the length of the route (sum of lengths of the atomic routes),
- the average maximum speed (average of length (as weight) and maximum speed of the atomic routes),
- the priority of the route (sum of priorities of the atomic routes),
- and a set of conflicting routes (union of sets of conflicting routes per atomic route).

5.2.3 Safety system

The main tasks of the safety system simulator are checking if a certain route is currently available, and keeping and updating the occupation status of each track element in the double vertex graph.

Definition 5.2.2 (Availability of a route). A route is available if and only if all track elements in this route are unoccupied.

Definition 5.2.3 (Occupation of track elements). A track element is unoccupied if and only if none of the vertices in the track element's equivalence class is occupied.

A route consists of a certain amount of vertices. Each vertex is a member of one equivalence class. If one vertex in an equivalence class is unavailable, the whole track element is occupied and all other tracks using this track elements are unavailable.

Designing a full featured safety system simulator was not a project aim in itself. There are some known limitations. These limitations are fairly easy to overcome but have, due to time constraints, not been handled.

1. In the current implementation, it is not possible to simulate splitting trains. If after splitting, the first part of the train requests a route, and it releases the route after having driven it, the track section with the second part of the train still there is also being released.
2. In the current implementation, it is not possible to map a route to a train if the last track section in the route is occupied. This makes it not possible to reliably simulate joining of two trains: the dispatcher cannot map a route to the second part of the train, because the first part occupies a track section required by the route of the second part of the train.
3. Trains are assumed to fit within a signal block. If a very long train would occupy two or more signal blocks, the safety system thinks only the first signal block is occupied.
4. If a train has more than one route requests in its schedule (e.g., trains with partial routes), it will only request the next route if it has arrived at the destination of its current route. For example, if the schedule is $[(1, 5), (5, 7)]$, the train will request a route from signal 5 to signal 7 when it arrives at signal 5.

5.2.4 Train dispatcher

The train dispatcher module dispatches route requests and releases. It is the interface between the train process and the railway infrastructure state simulation (safety and signalling, route assignment, visualisation).

Example: route request and assignment

A few minutes before a train t_0 is about to arrive at a certain signal s_{0_A}, it will request a route from s_{0_A} to s_{0_B} ❶ (see Figure 5.1).

After the dispatcher receives a route request, it will collect all trains $t_1, \ldots, t_n$ expected to arrive at, or depart from the station within a certain amount of time ❷.

Then, for all t_i in $[t_0, t_1, \ldots, t_n]$ the dispatcher module looks up all available routes from s_{i_A} to s_{i_B}.

Retrieval of available routes is done in two steps. First, all routes between s_{i_A} to s_{i_B} are looked up from the route database ❸. Then all available routes are selected using a list comprehension ❹.

The returned collection of routes between each signalpair will be ordered by priority. The result is an ordered list l of ordered lists l_i of available routes:

$$l = [l_0, l_1, \ldots, l_n], l_i = [r_0, r_1, \ldots, r_m]$$

If there is an empty list l_i, the dispatcher will expand this list with all available partial routes from s_{i_A} to a signal between s_{i_A} and s_{i_B}.

The next step is to reduce l by removing all routes which will never be chosen, using Algorithm 4.1.1. The independent set heuristic is used to get a set of disjoint conflicting routes ❺.

The resulting list l does not contain the best independent set of routes. The instance reduction algorithm may have removed conflicting full routes while some of its partial routes may not conflict with a higher priority route.

To get a better set of disjoint routes, each new empty list l_i is expanded with all available partial routes from s_{i_A} to a signal between s_{i_A} and s_{i_B}. Then, the instance reduction algorithm and the independent set heuristic will be run again ❺.

The dispatcher will now reserve the available route in l_0 with the highest priority for t_0, by marking each track element in the route as occupied by t_0 ❻, and return the route to the requesting train ❼.

Releasing routes

Releasing a route is much easier: after the train arrives at its destination signal, it will notify the dispatcher module. The dispatcher module translates this arrival notification into a release command that is sent to the safety system. The safety system simulator updates all track element that were occupied by the train in question.

CHAPTER 6

Case study

To test the performance (both speed and result) of the route set selection procedure, topology data of an existing station, 's-Hertogenbosch in The Netherlands (see Figure 6.1), has been used. The discrete event simulator used an actually carried out schedule. This made it easier to compare the results of the route set selection procedure.

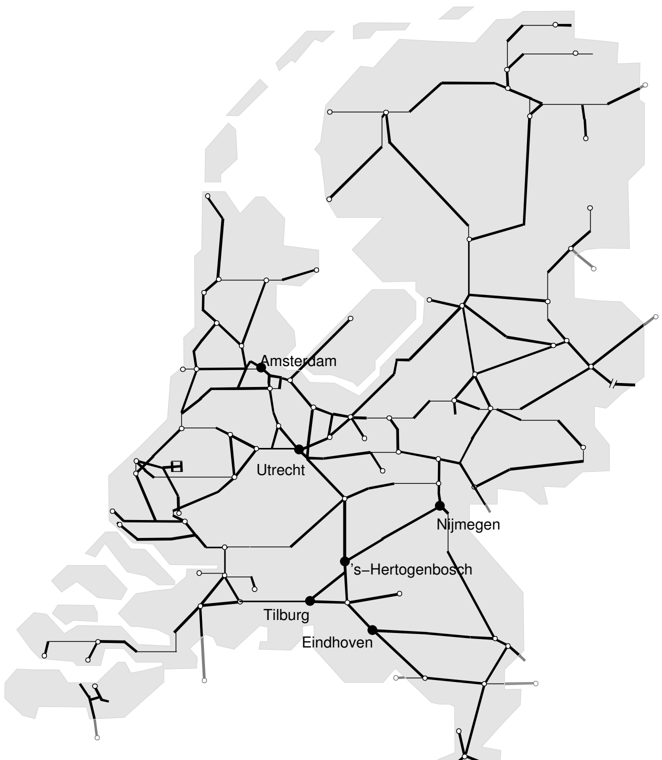

Figure 6.1: Railway map of The Netherlands.

6.1 Topology and schedule characteristics

The schedule is characterised by crossing train movements: trains from Tilburg to Nijmegen cross trains from Utrecht to Eindhoven. An added difficulty is that there are no non-conflicting junctions (fly-overs), neither at the north-side, nor at the south-side of the station.

Furthermore, some trains depart one after another from the same platform, or into the same direction with very short intervals (down to a minute). See Table 6.1 for some examples.

train	track	arrival	departure	direction/destination
3500	709B	.15/.45	.16/.46	Eindhoven
9600	706B		.17/.47	Eindhoven/Deurne
3500	703AB	.16/.46	.17/.47	Utrecht
16000	704B		.20/.50	Utrecht
3600	704B	.23/.53	.25/.55	Nijmegen/Zwolle
800/900	703AB	.28/.58	.30/.00	Utrecht/Haarlem

Table 6.1: Fragments of the simulated (18th June 2004) train schedule at 's-Hertogenbosch. The time difference between 3500 and 9600 and 3500 and 16000 ought to be enough, but in practise the 16000 is not able to depart on time, causing a delay for train 3600, which in turn causes a delay for 800/900 (because 3600 and 800/900 cross), et cetera.

To add even more 'stress' to the route set selection procedure, a schedule with shunting movements for the 'Autoslaaptrein'[1] has been chosen.

An overview of the railway track topology can be found in Figure 6.2.

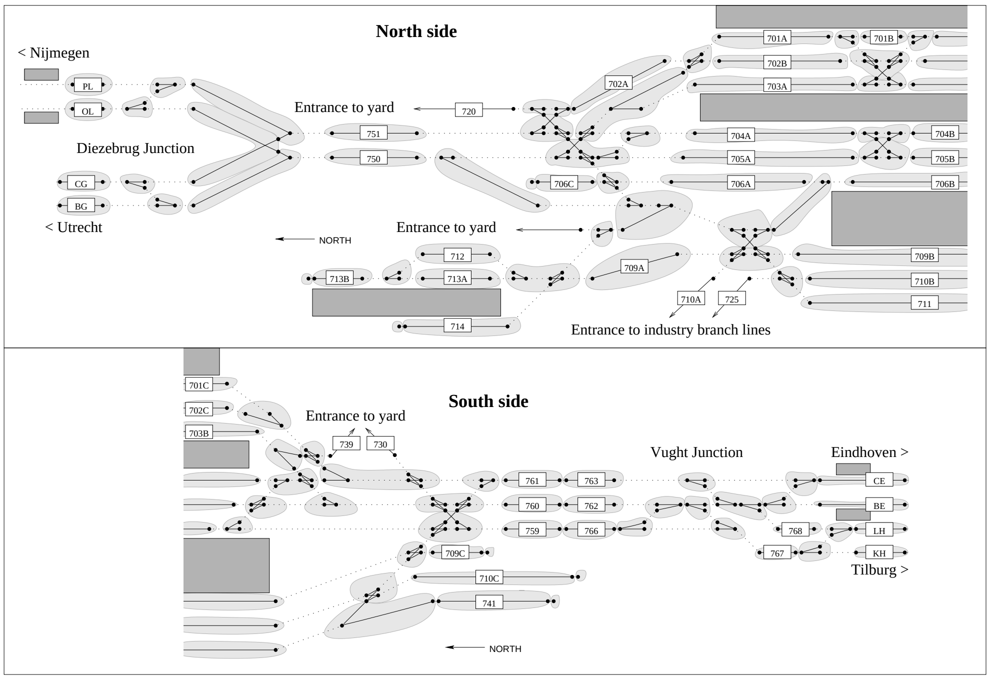

Figure 6.2: Formal track topology of 's-Hertogenbosch. The dots are the vertices of the double vertex graph, the dotted lines represent bijections between two vertices, the black lines are edges between vertices. There are some places (near crossings) where the bijection cannot be seen, because the two vertices overlap. The greyed areas are equivalence classes. Sometimes, two or more track sections belong to the same equivalence class.

The station has three platforms, with five tracks (or eight if you count the short platform tracks). The two small platforms at the north-east side are the platforms of 's-Hertogenbosch Oost (East) station, the two small platforms at the south side belong to Vught station.

The platform at the north-west side with terminus tracks 713AB and 714 is a mail platform. This platform is no longer in use, because there are no mail trains anymore. The tracks alongside this platform are sometimes used to park trains.

Furthermore, there are some tracks marked as 'Entrance to yard'. These yards are not shown, because there are no signals and remote controllable switches. It is possible to set routes from or to a yard entrance, but it is not possible to control from remote to which specific track within the yard the train is sent.

[1]International motorail train 's-Hertogenbosch – Avignon/Narbonne/Bologna. Passengers can take their car on board with them.

6.2 Results

The goals of this caste study are comparing the assigned routes with the precalculated routes, and to check the influence of the announce time and the 'look forward' time variables.

6.2.1 Comparison of route assignment with precalculated routes

The routes assigned by the route set selection procedure have been compared with the predefined routes. The at runtime selected routes equal the predefined routes, except at Vught Junction.

The difference can be explained by the inaccuracy of the simulator: departing trains seem to arrive a few minutes early at Vught Junction, causing other trains to receive curvy routes. When simulating only Vught Junction, the assigned routes equal the predefined routes.

6.2.2 Announce time

Each train requests a route a few minutes before arriving at or departing from the station. The amount of time between the route request and the actual driving of the assigned route is called announce time.

Because it has been assumed that the priority of the train depends on the time of arrival, a too long announce time may cause a bad route assignment result. For example, in Table 6.1, train 3500 departs before train 9600. If the announce time is too long, train 9600 may get a route assigned before train 3500. This behaviour is the consequence of the inability to reserve routes later in the schedule list in advance, and can be resolved by using a better simulator.

6.2.3 Look forward time

When a train requests a route, the dispatcher will gather all trains expected to arrive or depart within a certain time frame. This time frame will be referred to as look forward time. The value of the look forward time has a great influence on the result of the independent set heuristic to select the largest set of disjoint routes from a conflict graph.

For example, in Figure 6.3 a case is shown where train t_0 did not get a route because the look forward time was too large.

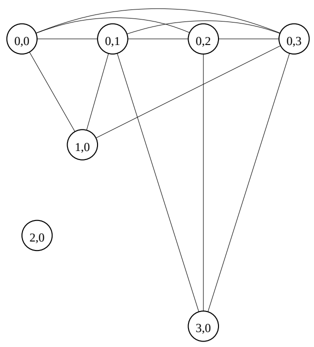

Figure 6.3: Example conflict graph after instance reduction. If the look forward time is set too high (e.g., thirty minutes) and t_0 is expected to arrive in one minute, while t_1, t_2 and t_3 arrive in twenty to thirty minutes, t_0 will not get a route. This is because $\{r_{2,0}, r_{3,0}, r_{1,0}\}$ is a larger independent set than $\{r_{2,0}, r_{0,0}\}$. Without train t_3, the independent set heuristic would return $\{r_{2,0}, r_{1,0}, r_{0,2}\}$.

This can be resolved by using a weighted independent set algorithm. The weights depend on the priority of the train. As already noted in Section 4.1, the assignment of a priority to a train falls outside the scope of this thesis and is probably a subject within the area of Operational Research.

CHAPTER 7

Conclusions

In this Bachelor's thesis, an attempt has been made to draft an 'intelligent' procedure to map routes to trains at runtime (i.e., while executing the schedule).

7.1 Findings and summary

Graphs are a natural way to represent railway topologies. Unfortunately, the usage of the standard graph $G = (V, E)$ raised the 'vertex problem', where pathfinding algorithms may return *physically impossible paths* when switches and/or crossings are modelled as vertices.

Several methods to solve the vertex problem have been discussed. All methods add extra information to the standard graph to prevent pathfinding algorithms to return physically impossible paths.

The double vertex graph method has been chosen to represent railway topologies as graphs, because this method was regarded as established. In contradistinction to methods based on directed graphs, notably vertex decomposition, the double vertex graph offers support for bidirectional railways.

Since route selection and assignment is being done at runtime, preprocessing has been used to calculate as much as possible in advance. One of the tasks of the preprocessing procedure is to calculate all routes in a track topology graph. Another task is to assign priorities to make an arrangement of routes between the same pair of vertices possible. The speed of the preprocessing procedure is not an important issue, since the preprocessing procedure should only be run after a change to the track layout.

When a train approaches a (large) station, multiple routes to its destination signal are possible. Some of these routes may conflict with routes of successive trains within a certain time frame (the look forward time).

It was shown that the problem of selecting a set of disjoint routes, can be interpreted as an independent set problem. Since an algorithm to solve an independent set is $\mathcal{NP}$-hard, and thus not suitable for usage at runtime, a $O(|V|)$ heuristic was used instead.

Given a conflict graph, this heuristic returns a set of disjoint routes. To reduce the size of the conflict graph, all low-priority routes may be removed from the graph in advance, using an instance reduction algorithm. If there is no route to the destination signal, there might be a partial route to an intermediate signal. Partials routes are useful to maintain traffic flow.

It has been shown that setting the look forward time too large, may cause that no routes are returned for the requesting train, even if there is a route available. It is advised not to set a too large look forward time, unless an *algorithm* is used to solve a weighted independent set.

7.2 Suggestions for future work

The formal track topology used in the case study was rather small (about 400 vertices, 60000 valid routes). To test the scalability of the double vertex graph implementation, preprocessor and route set selection procedure, much larger stations should be used. For example, the formal track topologies of Utrecht or Amsterdam may have thousands of vertices and millions of distinct, valid routes.

In the discrete event simulator, some features have been omitted due to time constraints. It is suggested to extend the simulator with support for joining and splitting trains and long trains occupying multiple track elements, or to use a better discrete event simulator based on double vertex graphs that supports these missing features.

Sometimes, it is better to send a train to a different platform track instead. In this thesis, it was assumed that the destination platforms tracks are decided by the train traffic controller. A possible Bachelor project would be to investigate how the decision at runtime, and constraints of sending a train to a different platform track can be formalised, and how it might be combined with the route set selection procedure.

Bibliography

[1] Kirsten Mark Hansen. Modelling railway interlocking systems. In *CAITS'96: Computer Applications In Transportation Systems*, 1996.

[2] Daniel Hürlimann. *Objektorientierte Modellierung von Infrastrukturelementen und Betriebsvorgängen im Eisenbahnwesen.* PhD thesis, Eidgenössische Technische Hochschule, Zürich, 2001.

[3] Markus Montigel. Formal representation of track topologies by double vertex graphs. In T.K.S. Murthy, J. Allan, R.J. Hill, G. Sciutto, and S. Sone, editors, *Proceedings of Railcomp '92*, pages 359–370, Washington DC, 1992. Computational Mechanics Publications.

[4] Steven S. Skiena. *The Algorithm Design Manual.* Springer-Verlag, New York, 1997.

[5] P.J. Zwaneveld, L.G. Kroon, and S.P.M. van Hoesel. Routing trains through a railway station based on a node packing model. *European J. Operational Research*, 128:14–33, 2001.