POLITECNICO DI TORINO

Master course in ICT for Smart Society

Master Thesis

Train scheduling modeling and optimization on railways network

Advisor
Prof. Monica Visintin

Gianmarco Garrisi

Ottobre 2019

Title of the thesis: Train scheduling modeling and optimization on railways network

Author: Gianmarco Garrisi

Advisor: Monica Visintin

Abstract: In this work, a mathematical formulation of the train scheduling problem is provided as a Mixed Integer Linear Program (MILP). Typically, the programs can be solved for easy cases, but computation time makes it impractical for more complex examples. Then a genetic algorithm is employed in the solution of the problem, with heuristic techniques to generate an initial population. The algorithm is applied to a number of problem instances producing feasible, though not optimal, solutions in less then one minute on a laptop computer. Some improvements are suggested to obtain better results and further reduce the computation time.

To all the people that gave me their support during my studies.

Acknowledgments

I would first like to thank my thesis advisor Prof. Cristina Cervelló-Pastor of the Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona at Universitat Politecnica de Catalunya. The door to Prof. Cervelló office was always open whenever I ran into a trouble spot or had a question about my research or writing. She consistently allowed this paper to be my own work, but steered me in the right direction whenever she thought I needed it.

I would also like to thank my advisor in Politecinco di Torino, Prof. Monica Visintin, for accepting to review my work, and giving me her very valuable comments.

I would also like to acknowledge Prof. Fabio Dovis of Politecnico di Torino as my tutor during my exchange period. Without him, this experience would have not been possible. He helped me to solve all the bureaucratic problems related to the Erasmus program whenever I felt lost.

Finally, I must express my very profound gratitude to my parents and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The objective of the project is to model the scheduling of trains on a railways network as a *Mixed Integer Linear Program* and develop a heuristic algorithm to optimize the scheduling.

In this work we are motivated by train scheduling but other situations may fit in the same mathematical model.

The creation of a new timetable is known to be an NP-hard problem. Nowadays this work is mostly done by humans[1], but some operators, like Schweizerische Bundesbahnen (SBB) are investigating the possibility to automatize the process[2]. This shows that there is an interest from the industry in this direction.

The situation that we want to address in this work is that of a network composed by busy multi-platform stations connected by at least a one-way track in each direction, that is the most common situation around Europe. We are not interested in the scenario, more common in North America or Russia, where there are far stations connected by one two-way track with loops for the intersection of trains in opposing directions.

Although the solver may be able to manage also this situation, there is a fair amount of research that specifically addresses it.

The task of scheduling trains on a network is an evolutionary process: there is almost never a complete redesign, but the new timetable is based on the previous one with some variations or adjustment. Some networks are operated on a *Clock-face schedule*, in which departures take place at the same times during the day, and they have a traffic so dense that respecting the clock-face structure is a priority. The Swiss railway network, for instance, has been adapted in such a way that the journey time between the main stations is a multiple of 30 minutes.

Another thing to consider, that is common in Europe, is the international traffic, that usually is decided before the internal traffic and becomes a constraint of the problem.

Scheduling means generating a timetable giving, for each train, arrival and departure times at various points on the network. The difficulty in this process is given by the fact that it is necessary to take into account possible conflicts between trains over a resource, that can be, for instance, a cross between two ways. The objective of the scheduling is to avoid such conflicts in the planning phase by computing the time instants in which each train should take over and release a certain resource.

The main difference with other kind of algorithms that can be employed for packets or also other kind of vehicles and infrastructure, is the fact that trains cannot overpass one another and that it is not possible to buffer them. This means, in other words, that if the entrance of a train into a section is delayed, then the section before will be occupied for a larger amount of time and no other train will be allowed into, nor through it.

Fast scheduling algorithms can also be employed to adjust the schedule in real time when some situation that had not been planned arise.

A further purpose of the algorithms is that they can be used to simulate and explore the effects of alternative draft timetable, operating policies, station layouts, and random delays or failures.

In the following chapters, we will present the *past research* and the *tools* used in the work; then, the *mathematical model* and the *heuristic* will be analyzed; finally, we will show and comment the *results* of applying the developed methods and draw some *conclusions*.

The results are obtained using the data provided by SBB for the train scheduling optimization challenge [3].

Chapter 2

State of the art

In this chapter, we will briefly describe some *past research* related to train scheduling and the *software libraries and tools* used in the proposed solution.

2.1 Related Work

Scheduling trains on a railway network is known to be an NP-hard problem with respect to the number of conflicts. Since the advent of computers, people started to think about ways to automatize the process. Some heuristics have been developed in the years to find and solve the conflicts in polynomial time.

Nedeljkovic and Norton[4], for example, developed for Westrail heuristic techniques to generate master train schedules recognizing the relative Priorities of trains. The method relies heavily on man-machine interaction.

Remaining in Western Australia, Mees[5] presents an efficient approximate algorithm which can find good feasible solutions for real-world networks quickly with modest computing resources.

Cai and Goh[6], propose an algorithm which is based on local optimality criteria in the event of a potential crossing conflict. The suboptimal but feasible solution can be obtained very quickly in polynomial time. The model can also be generalized to cater for the possibility of overtaking when the trains have different speed.

Higgins, Kozan, Ferriera wrote a series of researches in the field, employing branch and bound[7] and various local search heuristic techniques[8] for the scheduling on single line rail corridors.

Carey and Crawford[1] propose a heuristic technique to solve the problem of scheduling trains on a network of busy complex stations, like the ones that can be found all around Europe.

Some recent researches apply genetic algorithms in the field, but mostly to reduce the number of trains subject to a given passengers flow[9], or to assign drivers to trains[10].

2.2 Technology used in this work

Pyomo [11][12] is the python-based package selected to solve the optimization model proposed in Section 3.1, along with Gurobi as underlying solver.

Pyomo is a framework that offers a comfortable modeling language for describing linear programs in Python and solve them using a wide number of underlying solvers, for example the GNU Linear Programming Kit, Gurobi and others. Gurobi is a very fast solver for MILPs.

The heuristic optimization program was developed using the Rust programming language. Rust is a fast and memory efficient language, with a tiny runtime and without garbage collection, that offers guarantees of memory-safety and thread-safety thanks to a rich type system and a compiler that enforces tight checks at compile time. Moreover, its great documentation and the complete and easy to use build tools, makes it almost effortless to build and test programs, even when they make use of multiple external libraries.

The “rayon” crate[1] was used to obtain concurrency based on *work stealing*. Work stealing is a scheduling strategy for computer programs executing a dynamically multithreaded computation, that can “spawn” new threads of execution, on a computer with a fixed number of processors (or cores). It does so efficiently both in terms of execution time, memory usage, and inter-processor communication.

In a work stealing scheduler, each processor in the computer system has a queue of work items (computational tasks, threads) to perform. Each work item consists of a series of instructions, to be executed sequentially, but in the course of its execution, a work item may also spawn new work items that can feasibly be executed in parallel with its other work. These new items are initially put on the queue of the processor executing the work item. When a processor runs out of work, it looks at the queues of other processors and ”steals” their work items. In effect, work stealing distributes the scheduling work over idle processors, and as long as all processors have work to do, no scheduling overhead occurs[13].

The “oxigen” crate provides a framework for genetic algorithms. This was modified and adapted to solve this specific problem and the modified version was published on GitHub.

“serde” was used to perform efficient serialization and deserialization of json objects into native Rust data structures for input of the problem instances and output of the solutions. It uses the powerful Rust preprocessor to automatically generate serializers and deserializers implementations from and into a wide range of formats. With some attributes it is possible to guide the serde implementations, for example changing the name of fields or structures between the serialized and deserialized form.

Also the “chrono” crate has been used for the correct management of time and “time-parse” to provide a parser for the duration in the ISO format. The crate chrono can manage dates, instants and durations also considering time zones. It also provides implementations for the mathematical operations between time instants and intervals, such that it is as easy to handle them as it is to manage any other numerical data types. It is also space optimal and reasonably efficient.

Finally we use the crate “rand” for pseudo-random number generators and other functions related to randomness and “petgraph” for a flexible graph data structure providing also the implementation of some basic algorithms and tools to visualize the graph.

[1]name used by Rust developers to indicate a published library or program

All the crates are available at `https://crates.io`, the official Rust Package Registry.

Chapter 3

Methodology

In this chapter a *model of the problem* will be proposed in the form of a *Mixed Integer Linear Program*. Then a *heuristic solution* employing *genetic algorithms* will be proposed.

As input data to the problem we assume to have a draft schedule which proposes some *service intentions* composed by a list of *section requirements* and a *route*, that is a Directed Acyclic Graph (DAG) composed by multiple alternative *route paths*.

A service intention represent a railway service that has to be scheduled on the network according to a draft timetable, while a route describes all the alternative paths that a train can follow to move between the scheduled stations.

The route paths are divided in *sections*. In order to be allowed to take over a section, a train must reserve all the related *resources*, that can be, for example, the track and the railway juctions at its ends. Figure 3.1 shows some examples of route sections. The route sections are annotated with the minimum running time required by the train that execute that service to traverse the section. If two service intentions run technically on the same route but with different rolling stock (with different characteristics of maximum speed and acceleration), then two different routes must be provided. Moreover, certain paths may be discouraged by the railway manager or operator; in this case a penalty is associated to the sections.

Some sections may correspond to stations at which a service may be required to stop for passenger or freight service. In this case, a section requirement may specify connections with other services, a minimum stopping time, and some earliest or latest entry or exit time.

The objective of the solver is to

- choose a path for each service intention and
- assign a time instant to each event, i.e. decide for each section of each route, the moment a train should enter it and the moment it should exit

in order to minimize the delay and try to avoid some penalized paths.

The scheduled service intention, with its path and the assigned events, is called a train run, or sometimes, simply train.

In other terms the solutions must respect a number of constraints that are formalized into the mathematical model and enforced by the heuristic and must minimize an objective function where the delays and the penalties for running on specific route paths contribute to.

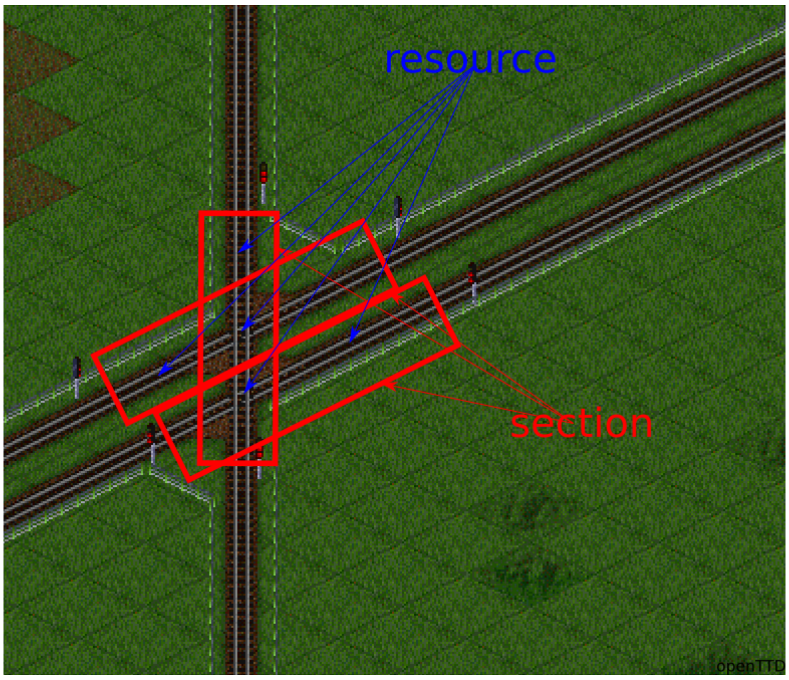


Figure 3.1: Example of route sections

3.1 A model of the problem

In the presentation of the model we will start by introducing some *notation* and the structure of the *input files*. Then the meaning of the employed *parameters* and *variables* will be explained. Finally the *objective function* and the *constraints* will be introduced and discussed.

To exemplify the various parts of the mathematical model, the sample scenario provided in the SBB github repository [3] will be used.

3.1.1 Subscripts and superscripts

We will use the following superscripts:

in to indicate a parameter or a variable related to the entrance in a section;

out to indicate a parameter or a variable related to the exit from a section;

and the subscripts to indicate an index that can vary into a set.

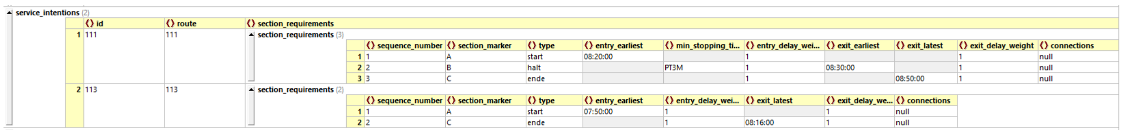
3.1.2 Input data

The input files consist in json serialized structures that represent the service intentions, the routes and the resources.

The service intention is an ordered list of section requirements. Because json serialization and deserialization does not guarantee to keep the ordering correct, each entry contains a sequence number.

The section requirement object encodes information about the earliest and latest instant a train is expected to enter and exit a station or a certain block, the minimum duration of the stop at a station and the weight of a delay after the latest intended entry or exit of the train into the required section. All the information is optional. Moreover it can contain connections with other service intentions. In this case the connection structure indicates how much it should last in order to allow passengers to comfortably change service.

Figure 3.2 shows a "grid view" of the part of json file that contains the service intentions.



service_intentions (2)

	id	route	section_requirements
1	111	111	section_requirements (3)
2	113	113	section_requirements (2)

section_requirements (3)

	sequence_number	section_marker	type	entry_earliest	min_stopping_ti...	entry_delay_wei...	exit_earliest	exit_latest	exit_delay_weight	connections
1	1	A	start	08:20:00		1			1	null
2	2	B	halt		PT3M	1	08:30:00		1	null
3	3	C	ende			1		08:50:00	1	null

section_requirements (2)

	sequence_number	section_marker	type	entry_earliest	entry_delay_wei...	exit_latest	exit_delay_we...	connections
1	1	A	start	07:50:00	1		1	null
2	2	C	ende		1	08:16:00	1	null

Figure 3.2: A grid view of the service intentions array

Each service intention is linked through a unique identifier to a route, that is a directed acyclic graph represented through its edges, that are the route sections. The graph is divided into route paths, linear sub-graphs that represent the alternative paths a train can take, for example different ways into a multi-platform station. The edges in each route path are connected in sequence and the route paths are connected to each other using markers, i.e. unique strings that identify nodes.

Each route section contains the list of identifiers of the resources a train would require when occupying the section and the minimum time required for the train to pass through the section. An optional section marker allows the link between the section requirements and the route sections.

Figure 3.3 contains a graphical representation of the route graph modeled by the "sample_scenario.json" file

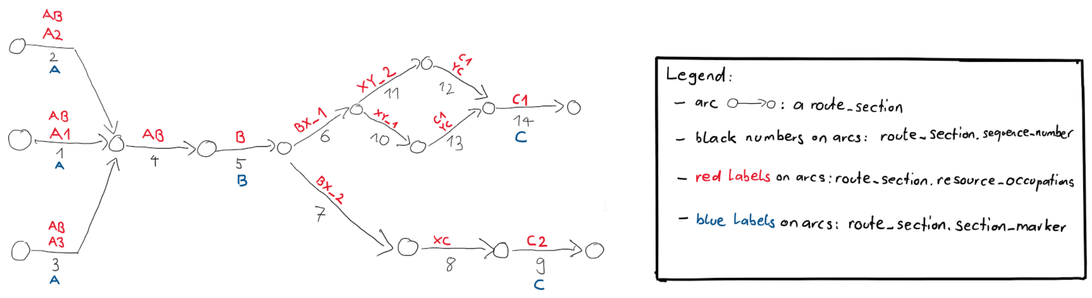

AB
A2
2
A
AB
A1
1
A
AB
A3
3
A
AB
4
B
5
B
BX_1
6
XY_2
11
XY_1
10
C1
YC
12
C1
YC
13
C1
14
C
BX_2
7
XC
8
C2
9
C
Legend:
– arc ○⟶○ : a route_section
– black numbers on arcs: route_section.sequence_number
– red labels on arcs: route_section.resource_occupations
– blue labels on arcs: route_section.section_marker

Figure 3.3: A graphical representation of the route contained in the json file

Finally, the resources are objects identified using a string that contain a release time. Some resources could allow following trains in the same direction to pass through before the release time has expired.

3.1.3 Parameters

First of all, all the information about routes and service intentions are read from input files. Routes information includes an id and a list of route sections each one with values related to:

- Minimum running time, mrt
- Minimum stopping time, mst
- Penalty in case of using the route section by a service intention, p.
- Earliest/Latest entry/out times, $EarIn$, $EarOut$, $LatIn$, $LatOut$.
- Entry/Exit delay weights, win, $wout$
- Route alternatives

With this information, all the feasible paths for each service intention are computed, taking into account all the alternatives. In order to manage service intentions, routes and sections, we have defined different sets as parameters of our model. We have a list of service intentions, a list of paths for each one and a list of resources of each path of each service intention. We have also a general list of route resources.

SI Set of Service Intentions

P Set of Paths of each si (taking into account alternative paths). Each element of this set depends on the service intention, si, and one feasible route of the si, r, i.e. (si, r).

RS Set of Route Sections of each path of each si (counting alternative paths). Each element of this set depends on the service intention, si, one feasible route of the si and one route section of this path, rs, i.e. (si, r, rs).

RE Set of Resources feasible to be used by each si. Each element of this set depends on the service intention, si, and one feasible route section to be used by it, rs, i.e. (si, rs).

RSE Set of Resources of each si at each route. Each element of this set depends on the service intention, si, one feasible route of the si, and one section of whatever path, re, i.e. (si, r, re).

RSI Set of pairs of services intentions and possible resources to be occupied by them. Each element of this set depends on the pair of service intentions, $si1$, $si2$, and one section of the list of resources, re, i.e. $(si1, si2, re)$.

RSIRE Set of pairs of services intentions, their routes and possible resources to be occupied by them. Each element of this set depends on the pair of service intentions with feasible routes, $si1$, $r1$, $si2$, $r2$, and one section of the list of resources, re, i.e. $(si1, r1, si2, r2, re)$.

In addition we have to define:

C The minimum time to wait for a coincidence between two service intentions at a common section requirement

M and ϵ for linearization purposes.

3.1.4 Variables

The following are the variables defined in the model:

$t^{in}_{si,r,rs}$ Train entrance time into a section of a route.

$t^{out}_{si,r,rs}$ Train exit time of a route section.

$\delta_{si,r}$ Binary variable equal to 1 if the service intention si, uses the route r, 0 otherwise.

$x_{si,re}$ Binary variable equal to 1 if the service intention si, uses the resource re, 0 otherwise.

$\beta_{si_1,si_2,re}$ Binary variable equal to 1 if both services intentions $si1$ and $si2$ use the same resource re, 0 otherwise.

3.1.5 The objective function

The objective of the optimization is to minimize the delay of each train and at the same time avoid to use some tracks if possible. This can be modeled by an *objective function* that uses the *Goal Programming method.*

The following formulation is taken from the *train schedule optimization challenge* [3].

$$\begin{aligned} f(x) = \frac{1}{60} \cdot \Bigg(& \sum_{SI,R,RS} win_{rs} \cdot \max(0, t^{in}_{si,r,rs} - LatIn_{si,rs}) \\ & + wout_{rs} \cdot \max(0, t^{out}_{si,r,rs} - LatOut_{si,rs}) \Bigg) \\ & + \sum_{SI,R,RS} p_{si,rs} \cdot x_{si,rs} \end{aligned} \tag{3.1}$$

In this formulation, the priority of a service is encoded into the weight of the corresponding delay: a service with higher priority will have higher weights then one with a lower priority.

3.1.6 Constraints

We have to respect several constraints to get valid solutions.

First of all, we include constraints that define the behaviour of one train along the sections of a route to ensure a confortable railway service, that means avoiding a train to depart earlier than the scheduled departure time, avoiding the stop to be short to let the passengers get on or off the train safely, or avoiding the scheduled connection between two trains to be too short for the passengers to effectively take advantage of it.

More concretely, we have included the following constraints, the meaning of which is furtherly clarified in Figure 3.4:

$$t^{out}_{si,r,rs} - t^{in}_{si,r,rs} \geq mrt + mst - M \cdot (1 - \delta_{si,r}) \quad \forall \{si, r, rs\} \in RS \tag{3.2a}$$
$$t^{in}_{si,r,rs} \leq t^{out}_{si,r,rs} \quad \forall \{si, r, rs\} \in RS \tag{3.2b}$$
$$t^{out}_{si,r,rs} \leq t^{in}_{si,r,rs+1} \quad \forall \{si, r, rs\} \in RS \tag{3.2c}$$
$$t^{in}_{si,r,rs} \geq EarIn_{si,r,rs} - M \cdot (1 - \delta_{si,r}) \quad \forall \{si, r, rs\} \in RS \tag{3.2d}$$
$$t^{out}_{si,r,rs} \geq EarOut_{si,r,rs} - M \cdot (1 - \delta_{si,r}) \quad \forall \{si, r, rs\} \in RS \tag{3.2e}$$

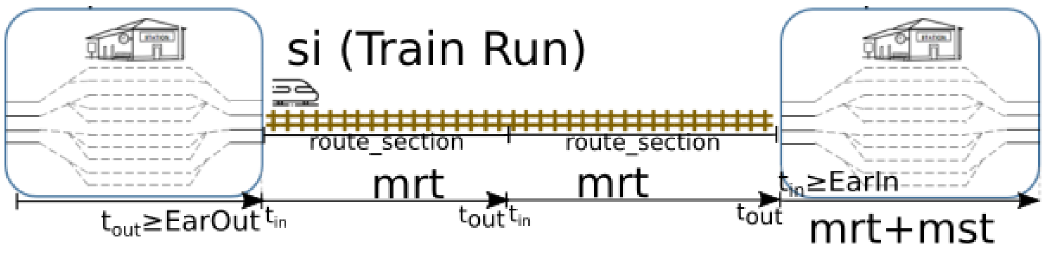


Figure 3.4: A graphical illustration of the fist set of constraints

The first set of constraints (3.2a) specify that, for the selected path of a service intention (with $\delta_{si,r} = 1$), the minimum time between entering and going out of a section is the summation of the minimum running time and the minimum stop time. In addition, the set of constraints in (3.2b) determine the sequence of times along the sections that form the route to be used by the service instance (the train first enters into the section and then goes out). Considering for example the sample scenario, the train that execute the service identified by the number 111 needs to stop for 3 minutes for the second section requirement. At the same time, it needs 32 seconds to run through that section.

Constraints in (3.2c) define that for each track section, it is occupied right after the section before is freed. In others words, after a train frees one section, it does not get lost, but goes immediately in the next section.

Constraints in (3.2d) and (3.2e) specify the earliest requirements for the selected path of each service intention (with $\delta_{si,r} = 1$). The first set of constraints (3.2d) avoids to depart earlier than the scheduled departure time, and the second (3.2e) avoids to stop too short in the station to let the passengers get on or off the train safely. The service 111 of the example has to enter the first of the section requirements after 8:20 and to leave the second not before 8:30.

In addition, we include the constraints (3.3a) to ensure that exactly one path is assigned to each service intention.

$$\sum_{r \in P} \delta_{si,r} = 1 \qquad \forall si \in SI \tag{3.3a}$$

On the other hand, constraints in (3.4a) establish the relationship between the variables $x_{si,re}$ and δ_{si_r}, i.e., if a route r is assigned to a service intention, all the sections of this path will be occupied by this service intention (train).

$$x_{si,re} \geq \delta_{si,r} \qquad \forall \{si, r, re\} \in RSE \tag{3.4a}$$

Finally, we need constraints to solve the coincidence at a common section of two service intentions.

The following constraints (3.5a) permit to fix $\beta_{si1,si2,re} = 1$ in case the two selected paths for the service intentions $si1$ and $si2$ ($r1$ and $r2$, respectively), coincide at the resource re and $si1$ enters earlier than $si2$. In the same way, constraints (3.5b) establish $\beta_{si1,si2,re} = 0$ in case the two selected paths for the service intentions $si1$ and $si2$ ($r1$ and $r2$, respectively), coincide at the resource re and $si1$ enters later than $si2$.

$$\begin{aligned} t^{in}_{si1,r1,re} - t^{in}_{si2,r2,re} \leq & M \cdot (1 - \beta_{si1,si2,re}) + \\ & M \cdot (2 - \delta_{si1,r1} - \delta_{si2,r2}) \\ & \qquad \forall \{si1, r1, si2, r2, re\} \in RSIRE \end{aligned} \tag{3.5a}$$

$$\begin{aligned} t^{in}_{si2,r2,re} - t^{in}_{si1,r1,re} + \epsilon \leq & M \cdot \beta_{si1,si2,re} + \\ & M \cdot (2 - \delta_{si1,r1} - \delta_{si2,r2}) \\ & \qquad \forall \{si1, r1, si2, r2, re\} \in RSIRE \end{aligned} \tag{3.5b}$$

The following constraints (3.6a) determine that in case of section coincidence and $\beta_{si1,si2,re} = 1$, the second train ($si2$) entry time to the section has to be delayed until the first train ($si1$) has left the resource plus an extra time C (time to wait between two trains that share a common resource). On the contrary, the constraints (3.6b) provoque the delay of the first train.

$$\begin{aligned} t^{out}_{si1,r1,re} - t^{in}_{si2,r2,re} + C \leq & M \cdot (1 - \beta_{si1,si2,re}) + \\ & M \cdot (2 - \delta_{si1,r1} - \delta_{si2,r2}) \\ & \qquad \forall \{si1, r1, si2, r2, re\} \in RSIRE \end{aligned} \tag{3.6a}$$

$$\begin{aligned} t^{out}_{si2,r2,re} - t^{in}_{si1,r1,re} + C \leq & M \cdot \beta_{si1,si2,re} + \\ & M \cdot (2 - \delta_{si1,r1} - \delta_{si2,r2}) \\ & \qquad \forall \{si1, r1, si2, r2, re\} \in RSIRE \end{aligned} \tag{3.6b}$$

The pseudo code of the collision avoidance is the following, illustrated in Figure 3.5:

```
if $re \in (r_1, r_2)$ and $(\delta_{si1,r1} = 1 \ \& \ \delta_{si2,r2} = 1)$ then
    if $\beta_{si1,si2,re} = 1$ (i.e. $t^{in}_{s1,r1,re} \leq t^{in}_{s2,r2,re}$) then
        $t^{in}_{s2,r2,re} \geq t^{out}_{s1,r1,re} + C$
    else {$\beta_{si1,si2,re} = 0$ (i.e. $t^{in}_{s1,r1,re} \geq t^{in}_{s2,r2,re}$)}
        $t^{in}_{s1,r1,re} \geq t^{out}_{s2,r2,re} + C$
    end if
end if
```

Relevant parts of the model input preparation are reported in Listing A.1. The model has been defined in the *pyomo* framework and executed using the *Gurobi* solver. In Listing A.2 it is possible to read its definition and execution.

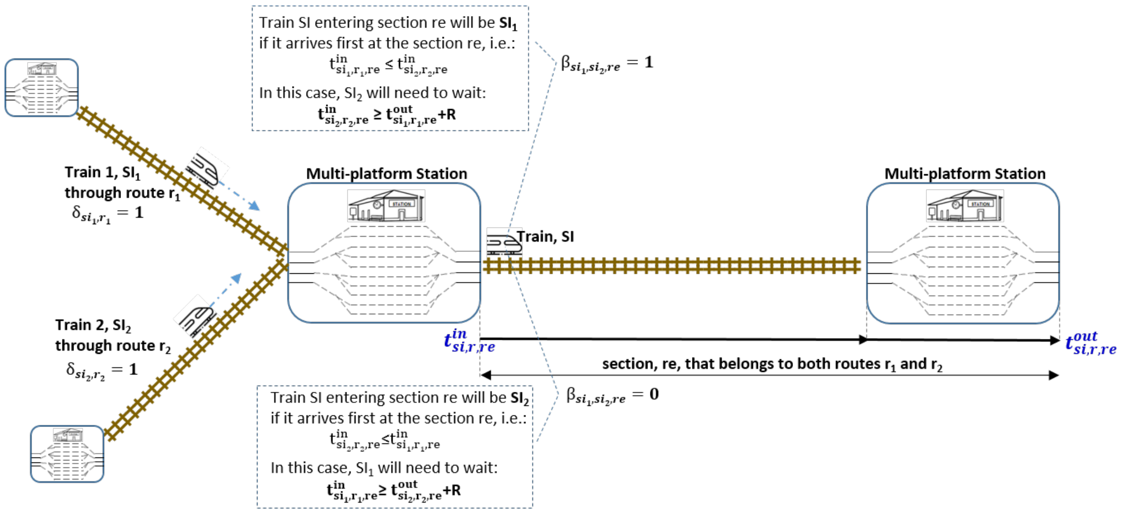

Train SI entering section re will be SI$_1$ if it arrives first at the section re, i.e.: $t^{in}_{si_1,r_1,re} \leq t^{in}_{si_2,r_2,re}$
In this case, SI$_2$ will need to wait: $t^{in}_{si_2,r_2,re} \geq t^{out}_{si_1,r_1,re}+R$
$\beta_{si_1,si_2,re} = 1$
Train 1, SI$_1$ through route r$_1$ $\delta_{si_1,r_1} = 1$
Multi-platform Station
Multi-platform Station
Train, SI
Train 2, SI$_2$ through route r$_2$ $\delta_{si_2,r_2} = 1$
$t^{in}_{si,r,re}$
$t^{out}_{si,r,re}$
section, re, that belongs to both routes r$_1$ and r$_2$
$\beta_{si_1,si_2,re} = 0$
Train SI entering section re will be SI$_2$ if it arrives first at the section re, i.e.: $t^{in}_{si_2,r_2,re} \leq t^{in}_{si_1,r_1,re}$
In this case, SI$_1$ will need to wait: $t^{in}_{si_1,r_1,re} \geq t^{out}_{si_2,r_2,re}+R$

Figure 3.5: Coincidence of two trains at a common resource

3.2 Implementation

The mathematical model of the problem is an integer optimization problem with side constraints, and it is difficult or impossible to solve it exactly in reasonable time. In order to solve this NP-hard problem, a heuristic approach is adopted that makes use of genetic algorithms to find a good solution.

As shown in Figure 3.6, the program reads the input files, generates an Instance data structure containing both the list of resources and the routes with the corresponding indication of the section requirements, and feeds the genetic algorithms framework. This will use the instance to generate a number of solutions that compose the initial population and to combine these solutions with the single point crossover technique to improve the objective value of the solution.

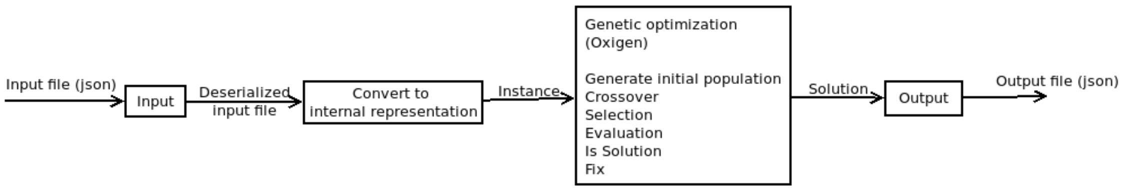


Figure 3.6: A block diagram of the heuristic solver

The full implementation of the program developed to solve the train scheduling problem will be discussed in this section, starting from the *data structures used internally*, the *resource manager* used to ensure a coherent management of the resources, the algorithm used to produce an individual of the initial population, that is a solution generated with a certain amount of randomness, to the techniques employed in the *genetic optimization* and the *output format*.

The input files have the same structure analyzed in 3.1.2 on page 14.

3.2.1 Internal representation

After the input file is deserialized it must be converted into a more suitable representation.

Each service intention and the corresponding route are merged into a unique data structure that is a graph. Each node of the graph will be an "event". It will contain an optional time, that will be left unassigned in the first place, and an optional marker that allows the connection of alternative rootpaths, if this is assigned by the "route_alternative_marker_at_entry" or "route_alternative_marker_at_exit" fields of the route section. The edges of the graph, instead will bring all the information about the route section and also to the section requirement if the current route section corresponds to a section requirement. Listing A.3 on page 42 shows the definition of the node and the edge of the graph, while Listing A.5 on page 43 the logic that builds it.

A vector will contain the source points and will be associated to the graph into the "Route" struct.

The set of the graphs obtained in this way, collected into a wrapper struct called "Route Manager" and the set of the resources taken directly from the input model will constitute the internal representation of the problem instance: a struct called "Instance".

A resource manager will be created from the list of resources every time a solution must be generated or checked.

After a service intention is scheduled, it will produce the internal representation of a "TrainRun", that will be a graph similar to the one in the instance, but with all the times assigned on the chosen path. For convenience, there is also a vector containing the sequence of nodes of the chosen path and another containing the related times. The service intention id is also kept associated with these structures. The collection containing all the "TrainRun"s is the "Solution" of the problem.

The "Route" and the "RouteManager" definitions can be found in Listing A.4 on page 42 while Figure 3.7 on the following page shows how the generated graph look like in a very simple case. The UML class diagram in Figure 3.8 illustrates the definition of the data structures.

The "Instance", the "TrainRun" and the "Solution" are instead in Listing A.7 on page 47.

This representation is optimal to compute the value of the objective function or to perform further optimizations, but it has to be converted into a more suitable one before being emitted as output. The struct also implements the "Genotype" trait providing the methods for the Genetic optimization.

3.2.2 Resource Manager

From the list of the resources a HashMap is generated that will allow direct access to a resource. The values are protected by a Mutex lock to allow resource allocation to be safely performed concurrently. In the value, a vector of intervals keeps track of the allocations. A wrapper object built around the map will provide safe management of the resources, allocating time intervals and considering the release time needed by each resource.

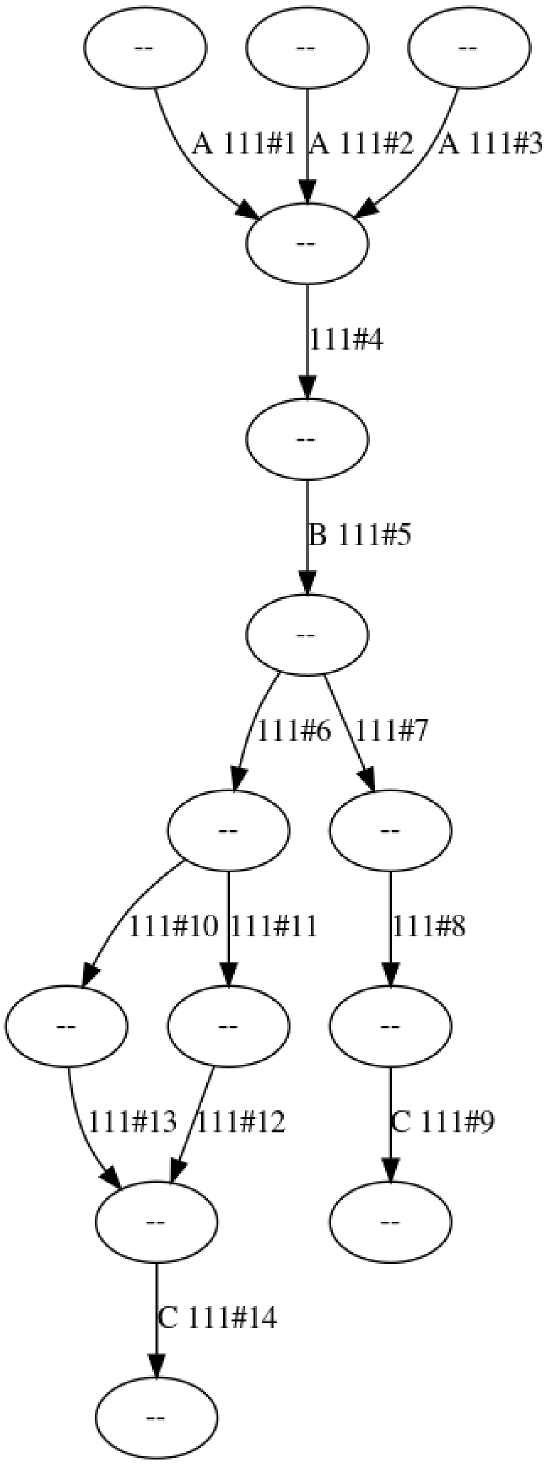


Figure 3.7: A representation of the graph contained into the Route structure. The arcs represent the route sections. The numerical label is the identifier of the section, composed by the concatenation of the route identifier with the sequence number of the route section, while the literal label, when present, identifies the section requirement related to that section. The nodes are the events in which the train pass from one route section to the next one. In this phase they are left unassigned

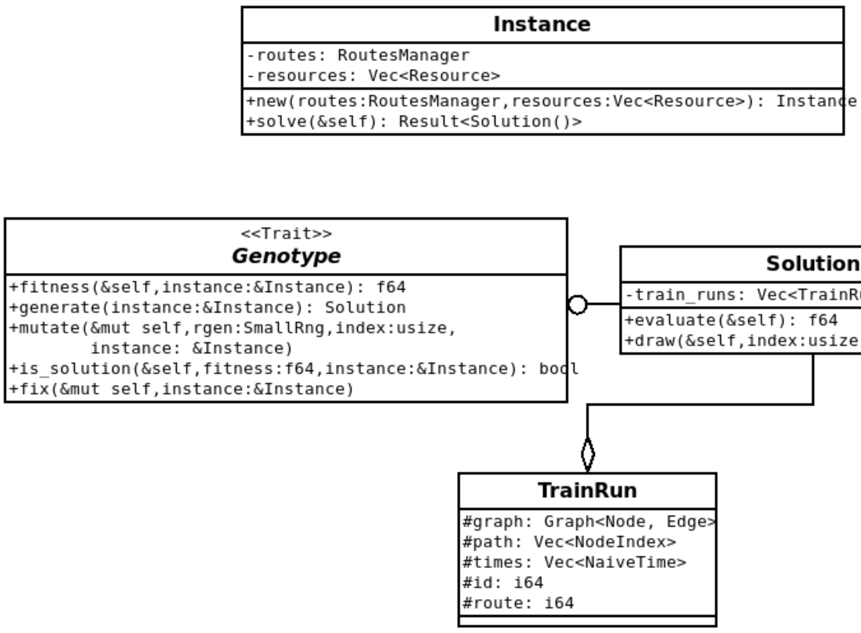

Instance
-routes: RoutesManager
-resources: Vec<Resource>
+new(routes:RoutesManager,resources:Vec<Resource>): Instance
+solve(&self): Result<Solution()>
<<Trait>>
Genotype
+fitness(&self,instance:&Instance): f64
+generate(instance:&Instance): Solution
+mutate(&mut self,rgen:SmallRng,index:usize, instance: &Instance)
+is_solution(&self,fitness:f64,instance:&Instance): bool
+fix(&mut self,instance:&Instance)
Solution
-train_runs: Vec<TrainRun>
+evaluate(&self): f64
+draw(&self,index:usize,filename:Path)
TrainRun
#graph: Graph<Node, Edge>
#path: Vec<NodeIndex>
#times: Vec<NaiveTime>
#id: i64
#route: i64

Figure 3.8: UML class diagram of the relevant classes that compose the internal representation of the system

The vector containing the allocation intervals works like an interval-tree, allowing only the insertion of non-overlapping intervals and refusing the other cases.

The "Interval" struct contains the identifier of the service intention and the index of the route section that is allocating the resource for the interval. It also contains the start and end instants.

The ordering operation ("Ord" trait) is implemented for this struct based only on the start field. This allows an easier and more efficient management of the consistency of this structure. In fact, to see if an interval has collisions, it is sufficient to perform a binary search into an ordered vector: if the search have success, we will know the exact position of the first colliding interval, otherwise the first colliding interval may be the one before the position where the new interval should have been, or the one in that position, or both. One of the two may not exist and in either case, only one comparison is needed.

The private method "search_collisions" returns the range of intervals colliding with a given interval on the given resource. The resource mutex must be locked before calling the method and the lock is passed to it. This functionality is exposed by the "collisions" public method that returns a vector with all the intervals in the range.

The "take" method, instead, uses it internally to verify that a request is acceptable. This function will accept allocations that collide with intervals scheduled by the same intention and return an error if the requested resource does not exist, if the requested interval has a negative duration (i.e. the end precedes the start), or if there is a collision, specifying the first collision with another intention. On success, instead, a token will be returned containing a reference to the resource and the interval, that allows easy deallocation of the resource.

An extensive test suite was developed for this data structure to find and fix bugs as it is the core of all the heuristic algorithm.

This structure heavily exploits cache locality, as all the intervals are stored in a continuous memory space, but an insertion requires the copy of all the intervals that are located after the insertion point. Another possibility would have been to base it on a Tree or B Tree structure, but it was not already available in the Rust ecosystem, it would have required more time to develop and its development is outside the scope of this work. Moreover it could also have had worse performance because of the impact of cache locality on the vector and the capability of modern computers to copy large portions of memory very quickly.

Listing A.6 on page 44 contains the code of the resource manager, including the definition of the data type, of the errors and the token that is returned after a successful allocation.

3.2.3 Generation of the initial population

Each individual of the initial population is generated according to the algorithm described in this section. The algorithm will produce a feasible solution that tries to be good, but not optimal. Some randomness ensures that the initial population is wide enough.

The algorithm is designed to be partially concurrent, so that it can scale out easily on more complex problems using more hardware instead of more expensive one.

Tha algorithm take as input an Instance structure and generates a Solution structure. To do this, for each route in the RouteManager, a random path is chosen, the time events assigned and the resources allocated. Figure 3.9 illustrates in a block diagram the main steps of the algorithm, that will be examined in depth in the follow-up.

The first step is to choose a path and compute a feasible timetable for this path. When there are alternative paths, one is chosen randomly, without considering that some paths will incur in penalties.

At this point, some of the events are assigned based on the section requirements that sets an entry earliest or exit earliest rule.

The other events, instead, are assigned according to the following strategy:

1. Push the node indices into the stack until the first node with an assigned time instant t_i is found;

2. At this point, pop one node from the stack and assign it the instant $t_{i-1} = t_i - minimum\_running\_time - minimum\_stopping\_time$;

3. Update $i = i - 1$ to proceed backward.

4. Go to step 2 until the stack is empty.

5. For each successive instant t_i, if it is unassigned, set

$$t_i = t_{i-1} + minimum\_running\_time + minimum\_stopping\_time,$$

otherwise $t_i =$

$$\max(t_i, t_{i-1} + minimum\_running\_time + minimum\_stopping\_time).$$

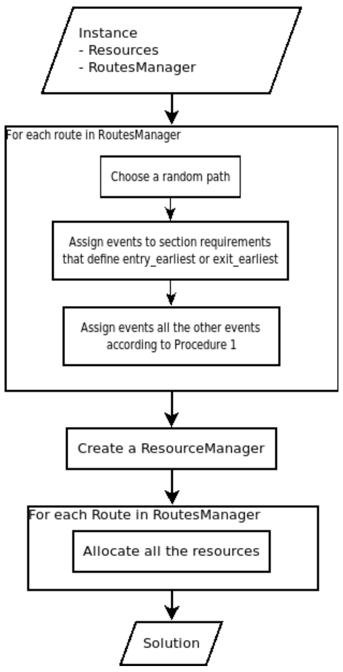

Instance
- Resources
- RoutesManager
For each route in RoutesManager
Choose a random path
Assign events to section requirements that define entry_earliest or exit_earliest
Assign events all the other events according to Procedure 1
Create a ResourceManager
For each Route in RoutesManager
Allocate all the resources
Solution

Figure 3.9: A high level block diagram of the individual generation function

This part of the algorithm is performed in parallel on the base of the service intentions: each service intention is treated as a separate job and actual concurrency use the technique of work stealing to avoid unequal division of the work between physical processing units. The code that execute this algorithm is reported in Listing A.8 on page 47.

Notice that at this point all the constraints except the resource allocation and the connections between services are enforced by construction.

The next step of the algorithm performs resource allocation. This is done in a separate method as it is also used to fix unfeasible solutions resulted by the crossover of other solutions in the genetic meta-heuristic framework.

To perform it, a new resource manager is created. Then, for each service intention, a recursive allocation function is called until it returns with success.

The recursive function:

1. Tries to allocate all the resources needed for one route section.

   If some allocation fails it will compute the first moment the unavailable resource becomes free, propagate the delay from the event of entry in the section and undo the allocations already performed (backtrack). Finally an error will be returned to the caller.

   If all the allocations succeed, it will recur onto the next section.

2. If the recursive call does return Ok, then also the current call will return Ok. Otherwise the instance of "alloc_r" that received Err will try to allocate all the needed resources for the additional time until the resources of the next section becomes available, as in step 1.

3. The call after the last section will return Ok and terminate the recursion.

The resource allocation is done in a sequential way to avoid repetitive conflicts on a resource by services that happen to occupy some common resources and to be scheduled simultaneously. The recursive function that is the core of the resource allocation is shown in Listing A.9 on page 49.

The delay propagation is performed in a way that ensures, for each couple of events i and $i+1$ that

$$t_{i+1} + delay_{i+1} \geq t_i + delay_i + mst + mrt$$

where mrt and mst are, respectively, the minimum running time for the section between the events and the minimum stopping time if the section is a requirement of halt.

This means that at each step of the propagation process, the value of the delay can be updated according to the following formula:

$$delay = t_i + delay - t_{i+1} + mst + mrt$$

in order to have the minimum possible value.

When the variable becomes negative, we probably hit a section requirement that sets an entry earliest or exit earliest rule that is adsorbing all the delay. In this case we stop the propagation and the negative delay will not be applied. The delay propagation is performed in the "propagate_delay" function, that can be found in Listing A.10 on page 50, in Appendix.

After this phase, the graph will be similar to the one illustrated in Figure 3.10 on the facing page.

3.2.4 Look Ahead

The "look ahead" method was developed as an improvement for the resource allocation algorithm. It propagates the delay to the next section requirement that sets an entry or exit latest rule and try to estimate how much the cost function would change because of that delay. The code that performs this estimate is reported in Listing A.11 on page 51.

With this information we know, in case of conflict, if it is better to delay the service that is being scheduled or the conflicting one. The first case is the same as acting without the look ahead method.

If we have to delay the other service, instead, we need to cleanup all the allocations relative to it at least since the offending route section, as done in Listing, and to push the service intention back into the queue of the intentions to be scheduled.

In case of multiple conflicts with the same allocation, the values to compare are computed as follows:

- For the scheduling service, the result of applying the look_ahead function with the delay necessary to move the allocation after the last offending one
- For the already scheduled intentions, the sum of the results obtained applying the look_ahead method to all the offending services with the delay necessary to move the first offending interval after the allocation we want to perform

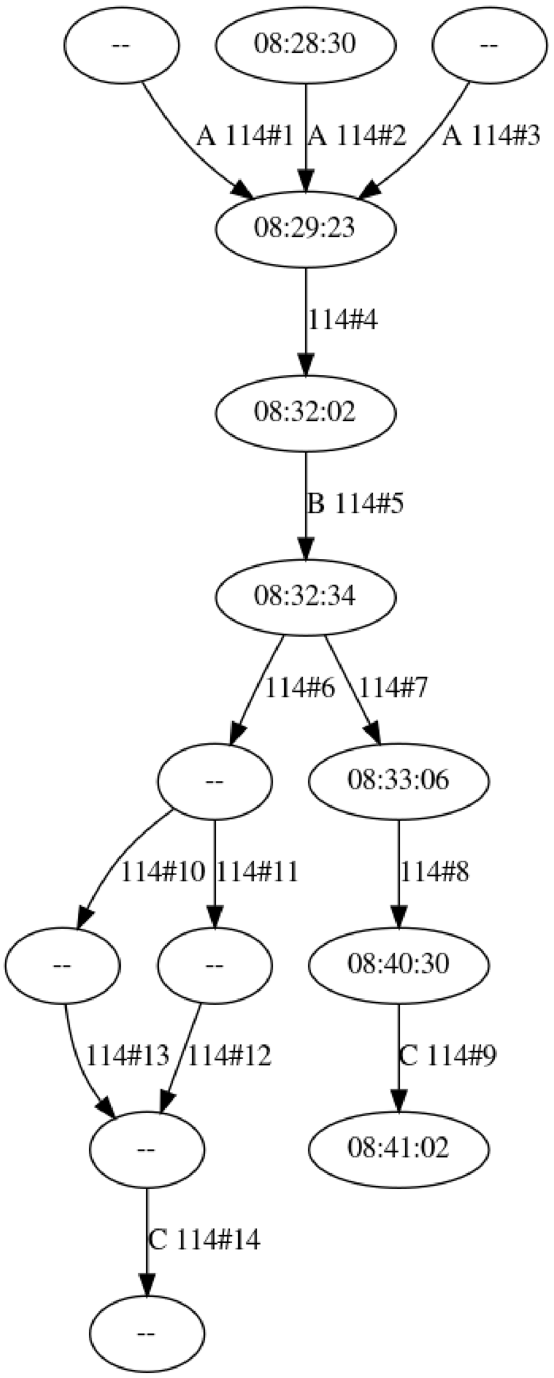


Figure 3.10: Example of graph contained in the TrainRun data structure. Note how it is similar to the one in Figure 3.7, but in this case, all the events have been assigned along the path

To avoid heavily dependent service intentions to be rescheduled an infinite number of times, a threshold is set and a counter is kept for each of the services so that, if one of the offending intentions has been rescheduled a number of times that exceed the threshold, the algorithm will move the allocation of the currently scheduling intention. With this threshold it is easy to disable the application of the look ahead method by just setting it to 1: in fact in this way all the intentions will be scheduled just once.

The value of the threshold needs to be fine tuned in order to find one that avoids the waste of computation time but at the same time does not degrade the performance.

It is also possible to use the inverse of the result of the look ahead function as a weight for a random choose in order to obtain non deterministic results and brake some patterns.

It is important to notice that it is hard to obtain a reliable estimate of the delay in which a service will occur without performing the actual resource allocation. In fact this implementation gives only a local estimate that does not keep into account, for instance, the possibility that allocations in the previous section (necessary to delay the event of entrance in the current section) fail or even that future allocations are refused by the resource manager. On the other hand it keeps into account the fact that a delay may be adsorbed into already delayed route sections.

Finally, given its local nature, this computation may bring to a decision that is not optimal in the long run. On the other end, giving a longer view to the look ahead algorithm may become too expensive while still not improving the results enough to justify the usage of that computational power. The exploration of the solution space will be done at the level of the genetic optimization.

3.2.5 Genetic Optimization

Once the initial population has been generated, this is used inside a Genetic Algorithm to obtain a better solution.

Genetic algorithms are evolutionary algorithms inspired by the mechanism of natural selection. This kind of metaheuristic has been successfully used in a wide range of search problems with a huge search space, being able to explore the space thanks to the crossover mechanism and to exploit the space around promising solutions thanks to the mechanism of mutation. The solutions with the highest quality is then selected.

In the genetic framework, we will use the single point crossover strategy, exchanging all the train runs after a certain point and the fitness function will be the negation of the value resulting by the application of the objective function explained in Section 3.1.5 on page 17.

The "Genotype" trait is implemented onto the "Solution" struct, where each train run is considered a *gene*. The individual can be streamed into the sequence of its genes and built from an iterator of this same type. The "is_solution" method of the trait checks that all the service intentions are scheduled and that there are no conflicts on any resource. This is done by issuing a new resource manager and trying to allocate every interval. As soon as one allocation fails this is interrupted and the function will return false. In this case the allocations are potentially performed in parallel on the base of the service intentions.

A fix function is provided so that the allocation of resources can be performed again in case the crossover generates collisions. No mutation strategy has been provided, so that only crossover will be employed in the research.

The genetic algorithm framework used for this work, “oxigen”, originally did not provide the possibility to access any data other than the current individual inside the functions that have to compute the fitness value, to validate a solution and to fix the individual in order to be valid. The “Genotype” trait has been modified in order to provide all the relevant methods with a reference to the Problem Instance, where it is possible to cache all the information that may be needed in problems of a certain complexity to be solved. The modified version has been published[1].

In particular, in this work, it is necessary to have access to the list of the resources inside the “fix” and “is_solution” methods in order to be able to build a new ResourceManager out of it.

The “oxigen” crate provides one more level of parallelism. In fact the generation of the population individuals and all the other operations are performed concurrently on the available threads using the same work stealing technique provided by the rayon crate.

Genetic Algorithms present high parallelization possibilities because all operations are independent, given that they operate on disjoint individuals. This is an important advantage point to consider given the increasing number of threads modern CPUs can handle simultaneously.

3.2.6 Output format

Finally, the program will output the found solution according to the output model.

The output is a json file. The main object is a Solution, that contains a label and a hash that identify the problem instance, a hash that assures integrity of the solution file and a list of train runs.

Each train run contains the identifier of the service intention and of the route into the problem instance and a list of train run sections.

A train run section indicates the time of entrance and of exit from the route section and the identifier of the route section itself, plus a sequence number to preserve the correct ordering and the section marker if the current section is a requirement.

The structures derive the Serialize trait and are automatically converted into valid json strings by the serde-json library.

3.2.7 The main function

The main of the program will orchestrate all the operations that need to be done. The parameters it takes as input on the command line are the path of the input file, containing the problem instance according to the input model, the path of the output file, where the best solution will be written to according to the output model,

[1]This version is available at `https://www.github.com/garro95/oxigen`

an optional objective value the objective function should reach and an optional number being the number of individuals in the population. The latter defaults to 16 if not provided.

If the expected value of the objective function is too low, the program could not manage to find a solution that satisfies that constraint. On the other hand, the optimization will terminate once the value is met, so a too high value may bring to solutions that are suboptimal, given that the optimization could run longer to provide better results. If no value is provided, the optimizer will use a stop criterion based on the progress of the objective function: when there is no more progress with respect to the previous generation, the optimization is terminated.

After opening and deserializing the input file from the provided path, it is converted into a problem instance. Then the "GeneticExecution" environment is initialized with the provided configuration or with the default one and it is run over the instance created before. When the optimization finishes, all the feasible solutions of the problem are evaluated and the value of the objective function is printed on the console. The best solution is saved onto the specified output file.

The code of the main function is in Listing A.12 on page 51

Chapter 4

Results

The results presented in this section are based on the problem instances provided for the "train schedule optimization challenge" proposed by SSB on the website CrowdAI[3].

First, the simplest possible scenarios are proposed as sample files. These have been modified to produce simple scenarios with collisions. The draft timetables for these scenarios are reported in Table 4.1. The case without collisions was solved by

Table 4.1: Sample cases timetables

Case	S.I.	A	B	C
1	111	EarIn: 8:20:00	EarOut: 8:30:00	LatOut: 8:50:00
	113	EarIn: 7:50:00	–	LatOut: 8:16:00
2	111	EarIn: 8:20:00	EarOut: 8:30:00	LatOut: 8:50:00
	113	EarIn: 8:20:50	EarOut: 8:30:00	LatOut: 8:48:50
3	111	EarIn: 8:20:00	EarOut: 8:30:00	LatOut: 8:50:00
	113	EarIn: 8:19:30	–	LatOut: 8:45:30

the Gurobi solver in 1.28 s. When adding the collisions, in two different formats, the model gave a result respectively in 1.25 s and 1.12 s.

In the same situations, the heuristic solver produced an exact solution in 0.009 s, probably dominated by input, output and concurrency set up. The optimal solution could be obtained in these cases using only the heuristic algorithm without further genetic optimization (i.e. using a population of only one individual).

The model could not be applied to more complicated problem instances due to the computation time becoming too high. Figure 4.1 shows the relative increase in computation time between the model solver and the heuristic.

The problem instances present examples with increasing level of difficulty.

The easiest one is the scheduling of 4 trains that do not have conflicts. Minimal routing is possible with some discouraged paths.

The second case proposes the routing of 58 trains with some conflicts and minimal routing alternatives.

The third case becomes more difficult, increasing the number of trains to 143 but still keeping minimal number of alternative paths.

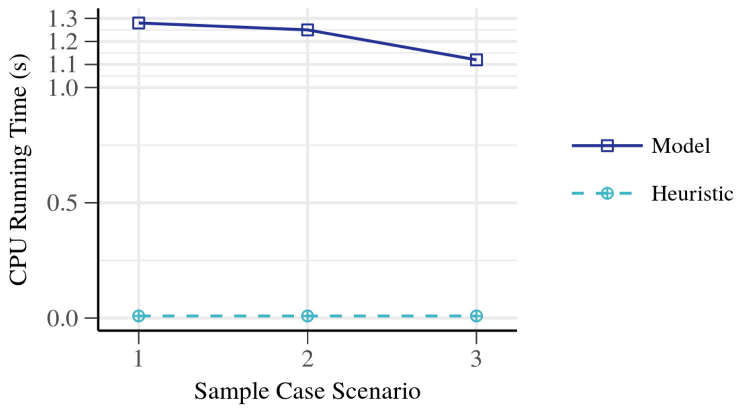


Figure 4.1: Computation times for model solver and heuristic

Table 4.2: Description of the problem instances

Problem Instance	Number of trains	Route alternatives
01_dummy	4	Minimal
02_a_little_less_dummy	58	Minimal
03_FWA_0.125	143	Minimal
04_V1.02_FWA_without_obstruction	148	Few
05_V1.02_FWA_with_obstruction	149	Few
06_V1.20_FWA	365	Some
07_V1.22_FWA	467	Some
08_V1.30_FWA	133	Lots
09_ZUE-ZG-CH_0600-1200	287	Lots

The fourth case increases the routing alternatives and features slightly more trains. The fifth case is very similar but with one more train and no optimal solution.

Sixth and seventh instances continue to increase the alternatives and add a lot more trains, while the eighth and ninth reduce again the number of trains but increase the number of paths.

For the last one it is not sure whether there is a solution with objective value 0. All the instances except number 5 can be solved optimally. Table 4.2 contains a schematic description of all the problem instances.

Note that instances from number 3 to number 9 also contain connections between services, but the logic to handle them has not been implemented in this project.

All the tests have been executed on a personal computer featuring a AMD A10 processor (4 cores, 1 thread per core), with 8 GB of RAM and a *Solid State Drive*.

The results for all the cases are reported in Table 4.3.

The easiest case can be solved with optimality also using a very small population, for example 2 individuals, in only 0.045 s.

The second case features 58 trains with minimal routing alternatives. This solver can find a feasible solution with an objective value of 2.95, that practically means, considering that all the delays are weighted 1, less then 3 min delay among

Table 4.3: Results obtained without applying the look ahead method

Problem Instance	Objective Value	Computation time (s)	Individuals
01_dummy	0.0	0.045	2
02_a_little_less_dummy	2.95	25.882	128
03_FWA_0.125	627.2	50.640	256
04_V1.02_FWA_without_obstruction	1253.33	60.295	256
05_V1.02_FWA_with_obstruction	2204	53.022	256
06_V1.20_FWA	5148.70	21.072	32
07_V1.22_FWA	21 919.95	34.394	32
08_V1.30_FWA	4770.65	10.122	64
09_ZUE-ZG-CH_0600-1200	5262.78	37.703	64

Table 4.4: Results obtained employing the look ahead method with a threshold of 100

Problem Instance	Objective Value	Computation time (s)	Individuals
01_dummy	0.0	0.116	16
02_a_little_less_dummy	142.98	33.078	128
03_FWA_0.125	1530.42	238.023	256
04_V1.02_FWA_without_obstruction	4844.83	330.661	256
05_V1.02_FWA_with_obstruction	8331.67	315.022	256
06_V1.20_FWA	-	-	32
07_V1.22_FWA	-	-	32
08_V1.30_FWA	65 041.5	177.728	64
09_ZUE-ZG-CH_0600-1200	10 267.4	37.788	32

all the section requirements of all the service intentions. This solution was found in a running time of 25.882 s using a population composed by 128 individuals.

The worst case instead was the seventh problem instance, that is the one featuring more trains. In this and also in the sixth case the solver would have been killed if run with a population of more then 32 individuals. In that case, the average delay per service intention is less then 1 h on all the schedule.

The usage of the look ahead method produces generally worse results, as it can be seen in Table 4.4. In some cases it is not able to find a feasible solution.

The computation time depends primarily on the population size. The most complex type of problem for this architecture is the one with lots of trains, because it tends to increase substantially the memory footprint of the program, that sometimes is killed for this reason.

The results in general are far from optimal. Several improvements are possible, for example:

- Implement a mutation strategy to further exploit the solution space around good solutions, for example exploring alternative paths or changing some time events in ways that are not explored in the generation of the individuals;
- Use some data structures that implement a copy on write mechanism in order to reduce the overall memory footprint and the number of allocations of the program.
- improve the look ahead method and fine tune the value of the threshold to obtain a better initial population

Chapter 5

Discussion

All the tests presented in the results section have been performed on a quite old Personal Computer. The high level of parallelism of this solution, obtained with real time concurrency and the optimal task scheduling, thanks to the work stealing technique, can easily scale on machines with higher core count.

Moreover, this approach can also be extended to wider clusters if the number of services to schedule becomes higher or a solution needs to be found in nearly real time.

Even for a medium size rail network, scheduling requires a large number of train schedulers or planners many months to complete, and makes it difficult or impossible to explore alternative schedules, plans, operating rules, objectives, ecc.

Works in this field can reduce in a sensible way the impact of timetabling on the operative costs of railways.

5.1 Environmental impact

Rail transport is recognized as an energy efficient (though capital intensive) means of transport. Moreover, each freight train can take a large number of trucks off the roads, making them safer.

Researches in this field can help to make railways more attractive to travelers. by reducing the operative cost and increasing the number of services and their punctuality.

In a world that is daily threatened by the impacts of global warming, we as society should take every possible move to reduce the usage of energy, make things more efficient and use means that can easily switch to renewable sources.

5.2 Conclusions

In this project a complex problem and its mathematical formulation is presented and its model is run against easy problem instances. More complex instances cannot be managed by a MILP solver. The proposed solver is based on genetic metaheuristic and can provide feasible solutions in seconds.

The usage of genetic algorithms allows the solver to scale well on more complex problems when more hardware is available, for example running in a computer with a high core count and a sufficient amount of main memory.

The results, far from being optimal, have delays that, in the worst case, are less then 1 hour per train on a very packed timetable draft.

Some improvements are proposed to obtain better results. They must be explored in order to make the project ready for the use in real world problems.

This research wants to make a step further in the direction of automation into the field of train scheduling on railways networks in order to increase the usage of smart and ecologic means of transport for people and freights in the society of the future and try to apply the benefits of information technologies as an innovation boost for one more field of what is meant to become the *Smart Society*.

Appendix A

Code snippets

In this appendix are reported the relevant parts of the code.

Listing A.1: Python code to read input data and preparation for the model execution

```
# Read input data
scenario = "./sample_files/sample_scenario_overlap2.json"
#sample_scenario2.json"
#sample_scenario_overlap1.json
#sample_scenario_overlap2.json

with open(scenario) as fp:
    scenario = json.load(fp)

ListOfResources = []
# Build the graph.
route_graphs = dict()
for route in scenario["routes"]:

    # set global graph settings
    G = nx.DiGraph(route_id = route["id"], name="Route-Graph for route "+str(route["id"]))

    # add edges with data contained in the preprocessed graph
    for path in route["route_paths"]:
        for (i, route_section) in enumerate(path["route_sections"]):
            sn = route_section['sequence_number']
            if sn not in ListOfResources: ListOfResources.append(sn)
            if 'section_marker' in route_section.keys() and route_section['section_marker']!=[]:
                sm = route_section['section_marker'][0]
            else:
                sm = None
            mrt = route_section['minimum_running_time']

            G.add_edge(from_node_id(path, route_section, i),
                       to_node_id(path, route_section, i),
                       sequence_number=sn,
                       section_marker=sm,
                       section_mrt = mrt,
                       section_penalty = route_section['penalty'])

    route_graphs[route["id"]] = G

# Extract all the information about routes and parameters of each train
trains = dict()
num_service_intentions = len(list(scenario["routes"]))
service_intentions = [0 for s in range(num_service_intentions)]

for si, route in enumerate(scenario["routes"]):
    # Create the graph of the route
    route_graph = route_graphs[route['id']]
    service_intentions[si] = route['id']

    for node in route_graph.nodes():
```

```
    route_graph.node[node]['label'] = node

edge_ids = {}
edge_markers = {}
edge_mrt = {}
edge_penalties = {}
for node1, node2, data in route_graph.edges(data=True):
    edge_ids[(node1, node2)] = data['sequence_number']
    edge_markers[(node1, node2)] = data['section_marker']
    edge_mrt[(node1, node2)] = data['section_mrt']
    edge_penalties[(node1, node2)] = data['section_penalty']

for edge in route_graph.edges():
    route_graph.edges[edge]['id'] = edge_ids[edge]
    route_graph.edges[edge]['marker'] = edge_markers[edge]
    route_graph.edges[edge]['mrt'] = edge_mrt[edge]

# Train service intention with all the routes and times information
trains[route['id']]={}

# Extract all the possible paths for a service instance
nodes=list(route_graph.nodes())
target = nodes[len(nodes)-1]
source_nodes=[s for s in nodes if "beginning" in s]
target_nodes=[s for s in nodes if "end" in s]
p=0
for source in source_nodes:
    for target in target_nodes:
        for path in nx.all_simple_paths(route_graph,source,target):
            trains[route['id']][p]={}
            pairs = [(path[i], path[i + 1]) for i in range(len(path) - 1)]

            rs=0
            for rs,edge in enumerate(pairs):
                re=edge_ids[edge]
                trains[route['id']][p][re]={}
                trains[route['id']][p][re]['id']=edge_ids[edge]
                trains[route['id']][p][re]['marker']=edge_markers[edge]
                mrt=GetISOTime(edge_mrt[edge])
                trains[route['id']][p][re]['mrt']=mrt
                trains[route['id']][p][re]['LatestIn']=0.0
                trains[route['id']][p][re]['LatestOut']=0.0
                trains[route['id']][p][re]['EarliestIn']=0.0
                trains[route['id']][p][re]['EarliestOut']=0.0
                trains[route['id']][p][re]['win']=0.0
                trains[route['id']][p][re]['wout']=0.0
                trains[route['id']][p][re]['mst']=0.0
                if edge_penalties[edge] is not None:
                    trains[route['id']][p][re]['p']=edge_penalties[edge]+1
                else:
                    trains[route['id']][p][re]['p']=1
                if edge_markers[edge] is not None:
                    for intention in scenario["service_intentions"]:
                        if intention['id']!=route['id']: continue
                        for sr in intention['section_requirements']:
                            if sr['section_marker'] == edge_markers[edge]:
                                if 'entry_earliest' in sr.keys():
                                    trains[route['id']][p][re]['EarliestIn']=\
                                        GetSecTime(sr['entry_earliest'])
                                if 'exit_earliest' in sr.keys():
                                    trains[route['id']][p][re]['EarliestOut']=\
                                        GetSecTime(sr['exit_earliest'])
                                if 'entry_latest' in sr.keys():
                                    trains[route['id']][p][re]['LatestIn']=\
                                        GetSecTime(sr['entry_latest'])
                                if 'exit_latest' in sr.keys():
                                    trains[route['id']][p][re]['LatestOut']=\
                                        GetSecTime(sr['exit_latest'])

                                # weights of the delay
                                if 'entry_delay_weight' in sr.keys():
                                    trains[route['id']][p][re]['win']=\
                                        sr['entry_delay_weight']
                                if 'exit_delay_weight' in sr.keys():
                                    trains[route['id']][p][re]['wout']=\
                                        sr['exit_delay_weight']

                                # minimum stop time
                                if 'min_stopping_time' in sr.keys():
                                    trains[route['id']][p][re]['mst']=\
                                        GetISOTime(sr['min_stopping_time'])

            p+=1
```

Listing A.2: Python code to Define and execute the model

```
#MODEL
from pyomo.environ import *
from pyomo.opt import SolverFactory
from pyomo.core import Var

model = ConcreteModel()

# Delete the parameters. Useful if we want to rerun this cell.
model.del_component( 'SI' )
model.del_component( 'P')
model.del_component( 'RS' )
model.del_component( 'RE' )
model.del_component( 'RSE' )
model.del_component( 'RSI' )
model.del_component( 'RSIRE' )

# Set of Service Intentions (SI)
model.SI = RangeSet(num_service_intentions)


# Paths of each SI (taking into account alternative paths)
model.P = Set(dimen=2, initialize=set((si, r) \
                                       for si in ServiceIntentions \
                                       for r in range(ServiceIntentions[si]['num_paths'])))


# Section Routes of each path of each SI (counting alternative paths)
model.RS = Set(dimen=3,
               initialize=set((si, r, rs) \
                              for si in ServiceIntentions \
                              for r in range(ServiceIntentions[si]['num_paths'])\
                              for rs in [trains[service_intentions[si]][r][i]['id'] \
                                         for i in list(trains[service_intentions[si]][r].keys())

# Set of Resources of each SI
model.RE = Set(dimen=2, initialize=set((si, re) \
                                       for si in ServiceIntentions \
                                       for re in ListOfResources))


# Set of Resources of each SI at each route
model.RSE = Set(dimen=3, initialize=set((si, r, re) \
                                       for si in ServiceIntentions \
                                       for r in range(ServiceIntentions[si]['num_paths'])\
                                       for re in ListOfResources))

# Set of pairs of services intentions and possible resources to be occupied
model.RSI = Set(dimen=3, initialize=set((si1, si2, re) \
                                       for si1 in ServiceIntentions \
                                       for si2 in ServiceIntentions\
                                       for re in ListOfResources))


# Set of pairs of services intentions, their routes and possible resources to be occupied
model.RSIRE = Set(dimen=5, initialize=set((si1, r1, si2, r2, re) \
                                       for si1 in ServiceIntentions \
                                        for r1 in range(ServiceIntentions[si1]['num_paths'])\
                                        for si2 in ServiceIntentions\
                                        for r2 in range(ServiceIntentions[si2]['num_paths'])\
                                        for re in ListOfResources))


# Minimum time to wait for a coincidence between
# two service intentions at a common section requirement
# In this case, it is considered equal for all situations,
# but it could have different values for each section.
C = 10

# For linearization purposes
M = 1000
epsilon=0.01

# Delete the variable. Useful if we want to rerun this cell.
model.del_component( 'tin' )
model.del_component( 'tout' )
model.del_component( 'delta' )
model.del_component( 'x' )
model.del_component( 'beta' )


# tin[si,r,rs] time of train entrance into a route section (si:train, r:route, rs:resource)
model.tin = Var(model.RS, within=NonNegativeReals, initialize=0)

# tout[si,r,rs] time of train exit of a route section (si:train, r:route, rs:resource)
model.tout = Var(model.RS, within=NonNegativeReals, initialize=0)

# delta[si,r]=1 if the service intention (train) si uses route r
model.delta = Var(model.P, within=Binary, initialize=0)

# x[si,re]=1 if the service intention (train) uses the resource re
model.x = Var(model.RE, within=Binary, initialize=0)

#beta[si1,si2,re]=1 if both services intentions si1 and si2 uses the same resource re
model.beta = Var(model.RSI, within=Binary, initialize=0)

# Delete the Objective function. Useful if we want to rerun this cell.
model.del_component( 'Objective' )

# Minimize weighted sum of all delays plus the sum of routing penalties
def Objective_rule(model):
    return 1/60.0*(sum(trains[service_intentions[si]][r][rs]['win'] * \
                     (model.tin[si,r,rs] - trains[service_intentions[si]][r][rs]['LatestIn'])+\
                     trains[service_intentions[si]][r][rs]['wout'] * \
                     (model.tout[si,r,rs] - trains[service_intentions[si]][r][rs]['LatestOut'])\
                     for (si,r,rs) in model.RS))\
            + sum (trains[service_intentions[si]][r][rs]['p']*\
                 model.x[si,rs] \
                 for (si,r,rs) in model.RS)

model.Objective = Objective(rule=Objective_rule, sense=minimize)

# Delete the constraint.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C1' )

# For the selected route of a SI, the difference between the
# output and the enter and exit time of the train has to
# be at least the minimum running time + minimum stop time
def C1_rule(model,si,r,rs):
    mrt = trains[service_intentions[si]][r][rs]['mrt']
    if 'mst' in trains[service_intentions[si]][r][rs]:
        mst = trains[service_intentions[si]][r][rs]['mst']
    else: mst=0
    return model.tout[si,r,rs] - model.tin[si,r,rs] >= mrt + mst- M*(1-model.delta[si,r])

model.C1 = Constraint(model.RS,rule=C1_rule)
```

```
# Delete the constraint.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C2' )

# In general, the exit time from a section has to be later than enter time.
def C2_rule(model,si,r,rs):
    return model.tin[si,r,rs] <= model.tout[si,r,rs]

model.C2 = Constraint(model.RS,rule=C2_rule)

# Delete the constraint.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C3' )

# After a train frees one section, it does not get lost, but goes immediately in the next sectio
def C3_rule(model,si,r,rs):
    pos=list(trains[service_intentions[si]][r].keys()).index(rs)
    if pos < num_route_sections[si][r]-1:
        return model.tin[si,r,list(trains[service_intentions[si]][r].keys())[pos+1]] >= model.to
    else:
        return Constraint.Skip

model.C3= Constraint(model.RS,rule=C3_rule)

# Delete the constraint.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C4' )

# For the selected route of a SI, include earliest-requirements.
# This constraint avoids to depart earlier than the scheduled departure time
def C4_rule(model,si,r,rs):
    if 'EarliestIn' in trains[service_intentions[si]][r][rs].keys():
        return model.tin[si,r,rs] >= trains[service_intentions[si]][r][rs]['EarliestIn']-\
            M*(1-model.delta[si,r])
    else:
        return Constraint.Skip

model.C4 = Constraint(model.RS,rule=C4_rule)

# Delete the constraint.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C5' )

# For the selected route of a SI, include earliest-requirements.
# This constraint avoids to stop too short in the station
# to let the passengers get on or off the train safely
def C5_rule(model,si,r,rs):
    if 'EarliestOut' in trains[service_intentions[si]][r][rs].keys():
        return model.tout[si,r,rs] >= trains[service_intentions[si]][r][rs]['EarliestOut']-\
            M*(1-model.delta[si,r])
    else:
        return Constraint.Skip

model.C5 = Constraint(model.RS,rule=C5_rule)

# Delete the constraint.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C6' )

# Each Service Instance has to get a path
def C6_rule(model,si):
    return sum(model.delta[si-1,r] for r in range(ServiceIntentions[si-1]['num_paths'])) == 1

model.C6 = Constraint(model.SI, rule=C6_rule)

# Delete the constraint.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C7' )

# Relationship between variables x and delta
def C7_rule(model,si,r,re):
    if alpha[si,r,re]==1: # if re belongs to the route r of si
        return model.x[si,re] >= model.delta[si,r]
    else:
        return Constraint.Skip

model.C7 = Constraint(model.RSE, rule=C7_rule)

# Constraints to solve the coincidence at a common section of two service intentions

# Delete the constraints.
# Useful if we want to rerun this cell without reinit the model.
model.del_component( 'C8' )
model.del_component( 'C9' )
model.del_component( 'C10' )
model.del_component( 'C11' )

# The following constraints  permit to fix   [si1,si2,re] = 1 in case the two
# selected paths for the service intentions s1 and s2 (r1 and r2, respectively), coincide at
# the section re, and si1 enters early than si2.

def C8_rule(model,si1,r1,si2,r2,re):
    if si1<si2:
        if re in list(trains[service_intentions[si1]][r1].keys())and \
           re in list(trains[service_intentions[si2]][r2].keys()):
              return  model.tin[si1,r1,re] - model.tin[si2,r2,re]  <= \
                    M*(1-model.beta[si1,si2,re]) +  \
                    M*(2-model.delta[si1,r1]-model.delta[si2,r2])
        else: return Constraint.Skip
    else: return Constraint.Skip

model.C8 = Constraint(model.RSIRE, rule=C8_rule)


# In the same way, the following constraints establish   [si1,si2,re] = 0 in case the two
# selected paths for the service intentions s1 and s2 (r1 and r2, respectively), coincide at
# the section re and si1 enters later than si2.

def C9_rule(model,si1,r1,si2,r2,re):
    if si1<si2:
        if re in list(trains[service_intentions[si1]][r1].keys())and \
           re in list(trains[service_intentions[si2]][r2].keys()):
              return  model.tin[si2,r2,re] - model.tin[si1,r1,re] + epsilon  <= \
                    M*(model.beta[si1,si2,re]) +  \
                    M*(2-model.delta[si1,r1]-model.delta[si2,r2])
        else: return Constraint.Skip
    else: return Constraint.Skip

model.C9 = Constraint(model.RSIRE, rule=C9_rule)


# The following constraints determine that in case of section coincidence
# and   [si1,si2,re] = 1, the second train (si2) entry time to the section has to be
# delayed until the first train (si1) has left the section plus an extra time C (time
# to wait between two trains that share a common section).

def C10_rule(model,si1,r1,si2,r2,re):
    if si1<si2:
        if re in list(trains[service_intentions[si1]][r1].keys())and \
           re in list(trains[service_intentions[si2]][r2].keys()):
              return  model.tout[si1,r1,re] - model.tin[si2,r2,re] + C <= \
                    M*(1-model.beta[si1,si2,re]) +  \
                    M*(2-model.delta[si1,r1]-model.delta[si2,r2])
        else: return Constraint.Skip
    else: return Constraint.Skip
model.C10 = Constraint(model.RSIRE, rule=C10_rule)


# On the contrary, the following constraints provoque the delay of the first train.

def C11_rule(model,si1,r1,si2,r2,re):
    if si1<si2:
        if re in list(trains[service_intentions[si1]][r1].keys())and \
          re in list(trains[service_intentions[si2]][r2].keys()):
              return model.tout[si2,r2,re] - model.tin[si1,r1,re] +  C <=  \
                   M*(model.beta[si1,si2,re])+ \
                   M*(2-model.delta[si1,r1]-model.delta[si2,r2])
        else: return Constraint.Skip
    else: return Constraint.Skip
model.C11 = Constraint(model.RSIRE, rule=C11_rule)

# Construct the instance of the model and pass it to an (external) solver, in this case GLPK
opt = SolverFactory('glpk')
#instance = model.create_instance()
results = opt.solve(model)
print("EXECUTION␣TIME:␣---␣%s␣seconds␣---" % (time.time() - start_time))

# Print the sequence of route sections of each service intention (train)
for si in ServiceIntentions:
    for r in range(ServiceIntentions[si]['num_paths']):
        if (model.delta[si,r].value!=0):
            print('TRAIN␣(route)',service_intentions[si])
            for rs in [trains[service_intentions[si]][r][i]['id'] \
                       for i in list(trains[service_intentions[si]][r].keys())]:
                print('␣␣␣␣␣section',trains[service_intentions[si]][r][rs]['id'])
                print('␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣in',GetTime(model.tin[si,r,rs].value))
                print('␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣out',GetTime(model.tout[si,r,rs].value))
```

Listing A.3: Rust code to define the Node and Edge in the route graph

```
#[derive(Default, Debug, Getters, Setters, Clone)]
#[get = "pub"]
pub struct Node {
    #[set = "pub"]
    time: Option<NaiveTime>,
    route_alternative_marker: Option<String>,
}

#[derive(Debug, Getters, Clone)]
#[get = "pub"]
pub struct Edge {
    route_section_id: (i64, i64),
    route_path_id: Id,
    section_marker: String,
    section_requirement: bool,
    resource_occupations: Vec<ResourceOccupation>,
    penalty: f64,
    starting_point: String,
    min_running_time: Duration,
    min_stopping_time: Duration,
    connections: Vec<Connection>,
    ending_point: String,
    entry_delay_weight: f64,
    exit_delay_weight: f64,
    entry_earliest: Option<NaiveTime>,
    entry_latest: Option<NaiveTime>,
    exit_earliest: Option<NaiveTime>,
    exit_latest: Option<NaiveTime>,
}
```

Listing A.4: Rust code that defines the route and the RouteManager

```
#[derive(Getters)]
#[get = "pub"]
pub struct Route {
    id: i64,
    intention_id: i64,
    graph: Graph<Node, Edge, Directed>,
    sources: BTreeSet<NodeIndex>,
}

pub struct RoutesManager {
    routes: Vec<Route>,
}
```

Listing A.5: Rust code to generate a route

```
/// Generate the graph relative to the possible routes given a specified service intention and the
/// route described in the problem instance. Merges the service requirements with the routes
fn generate_graph(
    route: Vec<RoutePath>,
    intention: ServiceIntention,
) -> (Graph<Node, Edge, Directed>, BTreeSet<NodeIndex>) {
    let mut g: Graph<Node, Edge, Directed> = Graph::new();
    let mut alternatives: HashMap<String, NodeIndex> = HashMap::new();
    let mut sources = BTreeSet::new();
    let requirements: IndexMap<_, _> = intention
        .section_requirements()
        .into_iter()
        .map(|sr| (sr.section_marker().clone(), sr))
        .collect();

    for mut rp in route {
        // sort to avoid any problem
        rp.route_sections
            .par_sort_unstable_by_key(|rs| *rs.sequence_number());
        let src_marker: Option<String> = rp.route_sections[0]
            .route_alternative_marker_at_entry()
            .clone()
            .and_then(|arr| arr.get(0).cloned());

        let src_node = Node {
            time: None,
            route_alternative_marker: src_marker.clone(),
        };

        let mut src = if let Some(marker) = src_marker {
            let a = alternatives
                .entry(marker.clone())
                .or_insert_with(|| g.add_node(src_node));
            sources.insert(*a);
            *a
        } else {
            let src = g.add_node(src_node);
            sources.insert(src);
            src
        };
        let rp_id = rp.id().clone();
        for rs in rp.route_sections {
            let section_marker = rs.section_marker().get(0);
            let requirement = section_marker.and_then(|marker| requirements.get(marker));

            let dst_marker: Option<String> = rs
                .route_alternative_marker_at_exit()
                .clone()
                .and_then(|arr| arr.get(0).cloned());
            let dst_node = Node {
                time: None,
                route_alternative_marker: dst_marker.clone(),
            };

            let dest = if let Some(marker) = dst_marker {
                *alternatives
                    .entry(marker.clone())
                    .or_insert_with(|| g.add_node(dst_node))
            } else {
                g.add_node(dst_node)
            };

            let rs = Edge {
                route_section_id: (*intention.id(), *rs.sequence_number()),
                route_path_id: rp_id.clone(),
                section_requirement: requirement.is_some(),
                section_marker: section_marker.cloned().unwrap_or("".to_string()),
                resource_occupations: rs.resource_occupations().clone(),
                penalty: rs.penalty().unwrap_or(0.0),
                starting_point: rs.starting_point().clone(),
                min_running_time: *rs.minimum_running_time(),
                ending_point: rs.ending_point().clone(),
                connections: requirement
                    .and_then(|r| r.connections().clone())
                    .unwrap_or_default()
                    .into_iter()
                    .filter_map(|oc| oc)
                    .collect(),
                min_stopping_time: requirement
                    .and_then(|r| *r.min_stopping_time())
                    .unwrap_or(Duration::zero()),
                entry_delay_weight: requirement
                    .and_then(|r| r.entry_delay_weight().clone())
                    .unwrap_or(0.0),
                exit_delay_weight: requirement
                    .and_then(|r| r.exit_delay_weight().clone())
                    .unwrap_or(0.0),
                entry_earliest: requirement.and_then(|r| r.entry_earliest().clone()),
                entry_latest: requirement.and_then(|r| r.entry_latest().clone()),
                exit_earliest: requirement.and_then(|r| r.exit_earliest().clone()),
                exit_latest: requirement.and_then(|r| r.exit_latest().clone()),
            };
            g.add_edge(src, dest, rs);
            src = dest;
        }
        sources = sources
            .into_iter()
            .filter(|nx| g.edges_directed(*nx, Direction::Incoming).count() == 0)
            .collect();
    }
    (g, sources)
}
```

Listing A.6: Rust code that implements the resource manager

```
/// Manage the resource allocation
#[derive(Debug)]
pub struct ResourceManager {
    resources: HashMap<String, Arc<Mutex<Resource>>>,
}

#[derive(Debug)]
pub enum AllocationError {
    Inexistent,
    Occupied(Interval),
    NegativeInterval(Interval),
}

#[derive(Debug, Getters)]
pub struct AllocationToken {
    #[get = "pub"]
    interval: Interval,
    resource: Arc<Mutex<Resource>>,
}

impl ResourceManager {
    /// Take a resource for a time interval. Returns `Ok(())` if the allocation was successful,
    /// `Err(Interval)` if the interval was occupied by another service intention.
    pub fn take(
        &self,
        resource_id: &str,
        mut interval: Interval,
    ) -> Result<AllocationToken, AllocationError> {
        if interval.start() > interval.end() {
            return Err(AllocationError::NegativeInterval(interval));
        }
        let r = self
            .resources
            .get(resource_id)
            .ok_or(AllocationError::Inexistent)?;
        let token = AllocationToken {
            interval,
            resource: r.clone(),
        };
        let mut r = r.lock().unwrap();
        let mut range = self.search_collisions(&r, interval);
        interval.end += r.release_time;
        if range.start >= range.end {
            let i = range.start;
            let mut prev_merged = false;
            let prev_i = i.overflowing_sub(1).0;
            if let Some(prev) = r.occupation_intervals.get(prev_i) {
                if prev.end == interval.start && prev.intention() == interval.intention() {
                    prev_merged = true;
                }
            }
            // check successor, if any
            if let Some(succ) = r.occupation_intervals.get(i) {
                if succ.start == interval.end && succ.intention() == interval.intention() {
                    // we can merge both
                    if prev_merged {
                        r.occupation_intervals[prev_i].end = r.occupation_intervals[i].end;
                        r.occupation_intervals.remove(i);
                    } else {
                        r.occupation_intervals[i].start = interval.start;
                    }
                    return Ok(token);
                }
            }
            if prev_merged {
                r.occupation_intervals[prev_i].end = interval.end;
            } else {
                r.occupation_intervals.insert(i, interval);
            }
            return Ok(token);
        }
        let collisions = &mut r.occupation_intervals[range.clone()];
        if let Some(int) = collisions
            .iter()
            .find(|c| interval.intention != c.intention)
        {
            if int.start > interval.end || interval.start > int.end {
                panic!("{:#?}, {:?}", r.occupation_intervals, interval)
            }
            return Err(AllocationError::Occupied(*int));
        }
        // merge all the colliding intervals into the first
        collisions.first_mut().unwrap().start =
            collisions.first().unwrap().start.min(interval.start);
        collisions.first_mut().unwrap().end = collisions.last().unwrap().end.max(interval.end);
        // check if also the successor needs to be merged
        if let Some(succ) = r.occupation_intervals.get(range.end).cloned() {
            if succ.start == r.occupation_intervals[range.clone()].first().unwrap().end
                && succ.intention == interval.intention
            {
                let collisions = &mut r.occupation_intervals[range.clone()];
                collisions.first_mut().unwrap().end = succ.end;
                r.occupation_intervals.remove(range.end);
            }
        }
        let first = r.occupation_intervals[range.clone()]
            .first()
            .unwrap()
            .clone();
        if let Some(prev) = r
            .occupation_intervals
            .get_mut(range.start.overflowing_sub(1).0)
        {
```

```
            if prev.end == first.start && prev.intention == interval.intention {
                prev.end = first.end;
                range.start -= 1;
            }
        }
        range.start += 1;
        //remove all the others
        r.occupation_intervals.drain(range.clone());
        Ok(token)
    }

    fn search_collisions(&self, r: &Resource, mut interval: Interval) -> Range<usize> {
        interval.end += r.release_time;
        match r.occupation_intervals.binary_search(&interval) {
            Ok(i) => {
                let mut j = i;
                for int in &r.occupation_intervals[i..] {
                    if int.start < interval.end {
                        j += 1;
                    } else {
                        break;
                    }
                }
                i..j
            }
            Err(mut i) => {
                // check predecessor, if any
                let prev_i = i.overflowing_sub(1).0;
                if let Some(prev) = r.occupation_intervals.get(prev_i) {
                    if prev.end > interval.start {
                        i -= 1;
                    }
                }
                let mut j = i;
                for int in &r.occupation_intervals[i..] {
                    if int.start < interval.end {
                        j += 1;
                    } else {
                        break;
                    }
                }
                i..j
            }
        }
    }

    /// Returns the vector of all the allocated intervals colliding with the given one.
    pub fn collisions(&self, resource: &str, interval: Interval) -> Vec<Interval> {
        let r = match self.resources.get(resource) {
            Some(r) => r,
            None => return Vec::new(),
        };
        let r = r.lock().unwrap();
        let range = self.search_collisions(&r, interval);
        r.occupation_intervals[range].to_vec()
    }

    pub fn free(&self, resource: &str, interval: Interval) -> Result<(), AllocationError> {
        if interval.start > interval.end {
            return Err(AllocationError::NegativeInterval(interval));
        }
        AllocationToken {
            interval,
            resource: self
                .resources
                .get(resource)
                .ok_or(AllocationError::Inexistent)?
                .clone(),
        }
        .free();
        Ok(())
    }
}
```

Listing A.7: Rust code that defines the internal Instance TrainRun and Solution

```
pub struct Instance {
    routes: RoutesManager,
    resources: Vec<crate::input_model::Resource>,
}

/// An internal representation of the intention solution.
#[derive(Debug, Getters, Clone)]
#[get = "pub"]
pub struct TrainRun {
    /// The graph with assigned events
    pub(crate) graph: Graph<crate::routes::Node, crate::routes::Edge>,
    /// The sequence of node ids traversed for the choosed path
    pub(crate) path: Vec<NodeIndex>,
    /// The times assigned to each event
    pub(crate) times: Vec<NaiveTime>,
    /// The id of the intention this solution refers to.
    pub(crate) id: i64,
    /// Identifier of the route
    pub(crate) route: i64,
}

#[derive(Clone, Debug)]
pub struct Solution {
    train_runs: Vec<TrainRun>,
}
```

Listing A.8: Rust code for the solve method

```
/// Generate a feasible solution
pub fn solve(&self) -> Result<Solution, ()> {
    let mut rng = rand::thread_rng();
    let train_runs: Vec<TrainRun> = self
        .routes
        .iter_random(&mut rng)
        .par_bridge()
        .map_init(rand::thread_rng, |mut rng, r| {
            let mut g = r.graph().clone();
            let int_id = *r.intention_id();
            let route_id = *r.id();
            let mut path = Vec::new();
            let mut stack = Vec::new();
            // choose a random path and assign time to the events
            let mut src = *r
                .sources()
                .iter()
                .choose(&mut rng)
                .expect("No starting point in the graph");
            path.push(src);

            // put the earliest times on the nodes
            while let Some((target, entry_earliest, exit_earliest)) =
                g.edges(src).choose(&mut rng).map(|e| {
                    (
                        e.target(),
                        *e.weight().entry_earliest(),
                        *e.weight().exit_earliest(),
                    )
                })
            {
                path.push(target);

                let t0 = std::cmp::max(*g[src].time(), entry_earliest);
                g[src].set_time(t0);

                let t1 = std::cmp::max(*g[target].time(), exit_earliest);
```

```rust
                g[target].set_time(t1);

                src = target;
            }
            for (src, target) in path.windows(2).map(|arr| (arr[0], arr[1])) {
                let edge_weight = g.edge_weight(g.find_edge(src, target).unwrap()).unwrap();
                let (min_run_time, min_stop_time) = (
                    *edge_weight.min_running_time(),
                    *edge_weight.min_stopping_time(),
                );

                match (g[src].time().clone(), g[target].time().clone()) {
                    (None, None) => {
                        // keep it for later
                        stack.push(src);
                    }
                    (Some(t0), None) => {
                        let t1 = t0 + min_run_time + min_stop_time;
                        g[target].set_time(Some(t1));
                    }
                    (None, Some(t1)) => {
                        let mut t0 = t1 - min_run_time - min_stop_time;
                        g[src].set_time(Some(t0));
                        let mut p = src;
                        // Fix all previous times
                        while let Some(s) = stack.pop() {
                            let e = g.edge_weight(g.find_edge(s, p).unwrap()).unwrap();
                            let min_run_time = *e.min_running_time();
                            let min_stop_time = *e.min_stopping_time();
                            let t1 = t0;
                            t0 = t1 - min_run_time - min_stop_time;
                            g[s].set_time(Some(t0));
                            p = s;
                        }
                    }
                    (Some(entry), Some(exit)) => {
                        g[target]
                            .set_time(Some(exit.max(entry + min_run_time + min_stop_time)));
                    }
                }
            }
            let times: Vec<_> = path
                .iter()
                .map(|&n| {
                    g[n].time()
                        .expect(
                            format!(
                                "FATAL ERROR! Time not assigned for intention {}. path: {:?}",
                                int_id, path
                            )
                            .as_str(),
                        )
                        .clone()
                })
                .collect();
            TrainRun {
                graph: g,
                path,
                times,
                id: int_id,
                route: route_id,
            }
        })
        .collect();
    let mut train_runs = Solution { train_runs };
    // perform resource allocation
    train_runs.fix(self);
    Ok(train_runs)
}
```

Listing A.9: Rust code for the recursive resource allocation function

```rust
fn alloc_r(
    train_runs: &HashMap<i64, TrainRun>,
    count_reschedule: &mut HashMap<i64, u64>,
    id: i64,
    path: &[NodeIndex],
    times: &mut [NaiveTime],
    graph: &Graph<routes::Node, routes::Edge>,
    deepth: usize,
    resource_manager: &ResourceManager,
    to_schedule: &mut VecDeque<i64>,
) -> Result<(), ()> {
    let int_id = id;
    let mut allocations = vec![];
    let dst_node = if let Some(dst) = path.get(1) {
        *dst
    } else {
        // EXIT CASE
        return Ok(());
    };
    let src_node = path[0];
    let mut res = Err(());
    while res.is_err() {
        let mut fail = false;
        let interval = Interval::new(int_id, deepth, times[0], times[1]);
        // try to allocate all the resources;
        let resources = graph
            .edge_weight(graph.find_edge(src_node, dst_node).unwrap())
            .unwrap()
            .resource_occupations()
            .iter();
        for r in resources {
            match resource_manager.take(r.resource(), interval) {
                Ok(allocation) => {
                    allocations.push(allocation);
                }
                Err(AllocationError::Occupied(occupation)) => {
                    let collisions = resource_manager.collisions(r.resource(), interval);
                    let delay1 = *collisions.last().unwrap().end() - times[0];
                    let delay = times[1] - *occupation.start();
                    let mut others = 0.0;
                    //look ahead the occupying schedules
                    for &occupation in &collisions {
                        if *occupation.intention() == int_id {
                            continue;
                        }
                        others += look_ahead(
                            occupation,
                            delay,
                            &train_runs[occupation.intention()].path,
                            &train_runs[occupation.intention()].graph,
                        );
                    }
                    //look ahead the current schedule
                    let current = look_ahead(interval, delay1.clone(), path, graph);
                    if others >= current
                        || collisions
                            .iter()
                            .any(|i| count_reschedule[i.intention()] >= THRESHOLD)
                    {
                        // move this
                        propagate_delay(delay1, times, path, graph);
                        fail = true;
                        break;
                    } else {
                        // move others and reschedule
                        for occupation in &collisions {
                            if occupation.intention() == &int_id {
                                continue;
                            }
                            to_schedule.push_back(*occupation.intention());
                            clean(
                                *occupation.intention(),
                                train_runs[occupation.intention()].times(),
                                train_runs[occupation.intention()].path(),
                                train_runs[occupation.intention()].graph(),
                                *occupation.index(),
                                resource_manager,
                            );
                            resource_manager.free(r.resource(), *occupation).unwrap();
                        }
                        let r = resource_manager.take(r.resource(), interval);
                        match r {
                            Ok(r) => allocations.push(r),
                            Err(e) => {
                                dbg!(e, collisions);
                            }
                        }
                    }
                }
                // Should not happen!
                Err(AllocationError::Inexistent) => panic!(),
                // should not happen!!!
                Err(AllocationError::NegativeInterval(i)) => {
                    panic!(
                        "I requested a negative interval {:?}! {}\n{:?}",
                        i, int_id, times
                    );
                }
            }
        }
        if fail {
            // backtrack
            for allocation in allocations.drain(..) {
                &allocation.interval();
                allocation.free();
            }
            return Err(());
        }
        res = alloc_r(
            train_runs,
            count_reschedule,
            id,
            &path[1..],
            &mut times[1..],
            graph,
            deepth + 1,
            resource_manager,
            to_schedule,
        )
    }
    Ok(())
}
```

Listing A.10: Rust code: function that propagates the delay

```
fn propagate_delay(
    starting_delay: Duration,
    times: &mut [NaiveTime],
    path: &[NodeIndex],
    graph: &Graph<routes::Node, routes::Edge>,
) {
    let mut delay = starting_delay;
    for (i, (src, dst)) in path.windows(2).map(|arr| (arr[0], arr[1])).enumerate() {
        times[i] += delay;
        let edge = &graph[graph.find_edge(src, dst).unwrap()];
        delay = times[i] - times[i + 1] + *edge.min_running_time() + *edge.min_stopping_time();
        if delay <= Duration::zero() {
            break;
        }
    }
    *times.last_mut().unwrap() += delay.max(Duration::zero());
}
```

Listing A.11: Rust code: look ahead function

```
/// returns how much the cost function would increase at the next station if
/// the service intention was delayed of `delay` seconds from interval
fn look_ahead(
    interval: Interval,
    mut delay: Duration,
    path: &[NodeIndex],
    g: &Graph<routes::Node, routes::Edge>,
) -> f64 {
    // propagate the delay to the next section requirement. This is optimistic.
    for (src, dst) in path
        .windows(2)
        .skip(*interval.index())
        .map(|arr| (arr[0], arr[1]))
    {
        let edge = &g[g.find_edge(src, dst).unwrap()];
        let mut res = 0.0;
        let mut end = false;
        // if the arc is a section requirement compute the increase of the cost function
        if let Some(t) = edge.entry_latest() {
            let t = edge.entry_delay_weight()
                * (((g[src].time().unwrap() + delay) - *t).num_seconds() as f64);
            if t >= 0.0 {
                res += t;
            } else {
                res += 10000.0;
            }
            end = true;
        }
        if let Some(t) = edge.exit_latest() {
            let t = edge.exit_delay_weight()
                * (((g[src].time().unwrap()
                    + delay
                    + *edge.min_running_time()
                    + *edge.min_stopping_time())
                    - *t)
                    .num_seconds() as f64);
            if t >= 0.0 {
                res += t
            } else {
                res += 10000.0;
            }
            end = true;
        }
        if end {
            return res;
        }
        // else propagate delay
        delay = (g[src].time().unwrap() - g[dst].time().unwrap()
            + delay
            + *edge.min_running_time()
            + *edge.min_stopping_time())
        .max(Duration::zero());
    }
    0.0
}
```

Listing A.12: Main function of the heuristic solver implemented in Rust

```
/// A program to optimize schedules of trains on a railways network
#[derive(StructOpt, Debug)]
struct Args {
    /// File containing the problem instance
    input_file: PathBuf,
    /// File that will contain the gerated solution
    output_file: PathBuf,
    /// Initial population of the algorithm
```

```rust
    #[structopt(default_value = "16")]
    initial_population: usize,
    /// Expected value of the objective function
    /// A value too low may produce only unfeasible solutions,
    /// a value too high could produce suboptimal solutions
    #[structopt(short = "o")]
    objective: Option<f64>,
}

fn main() -> Result<(), Box<dyn Error>> {
    let args = Args::from_args();
    let instance = read_instance_file(args.input_file)?;

    let (routes, strings) = RoutesManager::new(instance.routes, instance.service_intentions);
    // A hashmap of mutex to keep track of resources allocation.
    // routes.draw(0, "routegraph.png".into());

    let i = Instance::new(routes, instance.resources);
    // let mut sol = i.solve().unwrap();
    // println!(
    //     "{} {}",
    //     sol.is_solution(0.0, &i),
    //     sol.evaluate(),
    //     // sol.iter().map(|tr| tr.times()).collect::<Vec<_>>()
    // );
    // sol.draw(0, "train_run_graph.png".into());
    // output_model::Solution::new(instance.label, instance.hash, sol, &strings)
    //     .write(args.output_file)?;

    let mut optim: GeneticExecutionBuilder<_, Solution> = GeneticExecutionBuilder::default();
    optim
        .stop_criterion(Box::new(
            args.objective
                .map(|obj| StopCriteria::MinFitness(obj))
                .unwrap_or_else(|| StopCriteria::Progress(0.0)),
        ))
        .population_size(args.initial_population);
    let optim: GeneticExecution<_, _> = optim.into();
    let (solutions, _gens, _prog, _population) = optim.run(i);

    solutions
        .iter()
        .for_each(|s| println!("{:?}", s.evaluate()));

    if !solutions.is_empty() {
        output_model::Solution::new(
            instance.label,
            instance.hash,
            *solutions
                .into_iter()
                .min_by(|s0, s1| s0.evaluate().partial_cmp(&s1.evaluate()).unwrap())
                .unwrap(),
            &strings
        )
        .write(args.output_file)?;
    }

    Ok(())
}
```

Bibliography

[1] Carey, Malachy, Crawford, Ivan (2005). *Scheduling trains on a network of busy complex stations.* Transportation Research Part B 41 (2007) 159–178

[2] SBB (2018). *Train Schedule Optimisation Challenge; Optimizing train schedules* https://www.crowdai.org/challenges/train-schedule-optimisation-challenge

[3] Jordi, Julian (2018). *sbb-train-schedule-optimisation-challenge-starter-kit* https://github.com/crowdAI/train-schedule-optimisation-challenge-starter-kit

[4] Nedeljkovic, N.B., Norton, N.C. (1985). *Computerized Train Scheduling.*

[5] Mees, A. I. (1989). *Railway Scheduling by Network Optimization.* Math1 Compat. Modelling Vol. 15, No. 1, pp. 33-42, 1991 Pergamon Press plc

[6] Cai, X. and Goh, C. J. (1994). *A Fast Heuristic for the Train Scheduling Problem.* Computers and Operations Research , Vol 21, 499 - 510

[7] Higgins, A., Kozan, E. and Ferreira. L. (1996). *Optimal scheduling of trains on a single line track.* Transportation Research - Part B, 30B (2),147-161.

[8] Higgins, A., Kozan, E. and Ferreira. L. (1997). *Heuristic Techniques for Single Line Train Scheduling.* Journal of Heuristics, Vol 3, 43 - 62

[9] Nirmala, G., Ramprasad, D. (2014). *A Genetic Algorithm based railway scheduling model.* International Journal of Science and Research (IJSR) Volume 3 Issue 1, January 2014 11 - 14

[10] Dr. S. Sai Satyanarayana Reddy, G. S. Prasada Reddy, P. V. Hemanth, Priyadharshini Chatterjee (2017). *Train Time Scheduling using Genetic Algorithm.* International Journal of Civil Engineering and Technology, 8(12), 2017, pp. 410-413.

[11] W. E. Hartand, C. D. Laird, J. Watson, D. L. Woodruff, G. A. Hackebei, B. L. Nicholson, D. Siirola (2017). *Pyomo–optimization modeling in python*, 2nd Edition, Vol. 67, Springer Science & Business Media.

[12] W. E. Hartand, J. Watson, L. Woodruff (2011). *Pyomo: modeling and solving mathematical programs in python.* Mathematical Programming Computation 3 (3) 219–260.

[13] Blumofe, Robert D., Leiserson, Charles E. (1999). *Scheduling multithreaded computations by work stealing*. J ACM. 46 (5): 720–748.