Train scheduling and resource management in a railway station

Supervisors:
François Glineur
Pierre Schaus

Readers :
Sébastien Coppe
Jean-Charles Delvenne

Thesis submitted for the degree of
Master in Applied Mathematics Engineering

by
Salomé Mulders
Yu-Lan Scholliers

Louvain-la-Neuve
Academic year 2013-2014

Abstract

All around the world in densely populated areas, congestion in railway stations has become a major issue. Train companies hire consultants to face this rather vast dilemma, which requires not only a thorough knowledge and understanding of the problem itself, but also good optimization skills. The French organization for Operations Research and Decision Support, *ROADEF*, jointly with the European Operational Research Society, *EURO*, organized a challenge concerning this matter in collaboration with the French railway company, *SNCF*. It created an opportunity to explore this broad subject which is a real industrial problem. This thesis will present the developed approach to solve this problem and the obtained results.

We would like to express our gratitude towards our thesis supervisors Pr. Glineur and Pr. Schaus for introducing us to such an interesting problem, for supervising this thesis and helping us to conduct this work.

Our sincere thanks also goes to Mr. Coppe and Pr. Delvenne for accepting the task as reader.

Furthermore, we would like to offer special thanks to Andrew and Thibault who have been there to support us throughout the whole process and gave us insightful comments and suggestions.

Last but not least, we would like to thank our family for not only supporting us with advice and material support, but mostly for always being there for us.

Contents

Chapter 1

Introduction

Since the beginning of the existence of railway stations, in the early 18th century, a lot has changed. Railway stations have ceaselessly evolved together with the trains that use them. This growth has led to a compelling need for a sound management of trains in a station. The number of train users and the number of trains on the railways have indeed increased a lot these past few years. However, the capacities of a railway station have not increased in the same fashion because they are limited by the environment in which they are located: in the midst of urban areas.

In the past, the handling of trains was straightforward because there was enough capacity, but the optimization of train management has become an increasingly important subject nowadays. Hence, *ROADEF*[1] has decided in collaboration with the *SNCF*[2] to organize a challenge[6] which gave us not only the opportunity to tackle this interesting problem: train scheduling and resource management in a terminal station, but also to have sufficient resources at our disposal. The organization provided indeed a well-defined and documented problem statement together with test instances and a checker. Besides, the context allowed us to have benchmark solutions.

The problem consists in handling trains in the station from their arrival until their departure while taking also into account the fact that a train needs to be assigned to a departure, that maintenance can be needed, that some trains need junctions or disjunctions and that they must sometimes be stored during a certain amount of time. This challenge is formulated as an optimization problem[4] with a multi-objective function which represents different costs and needs to be minimized together with several constraints.

[1]Société française de Recherche Opérationnelle et Aide à la Décision.
[2]Société Nationale des Chemins de fer Français

The thesis committee imposed the additional constraint to implement the solution in SCALA[2].

This paper will start by a detailed explanation of the problem, followed by a small overview of the different attempts which finally led to a functional algorithm. Afterwards, an overview is given of the solutions of several given instances, with their verification through the checker[5] which is supplied by the organization of the challenge. Subsequently, the results are discussed and compared to the solutions of the groups who participated to the challenge. To conclude, some thoughts are shared on the possible further development of the algorithm.

Chapter 2

Problem statement

The aim of this challenge is to find an optimal schedule for trains between their arrivals and their possible departures in terminal stations while optimizing an objective function and respecting different constraints. The algorithm also has to adapt without any effort to different configurations of railway stations. The form of the solution has to be a schedule for each train that describes all its movements from the moment it enters the system until it exits.

The complete problem is described in [4] and will be introduced in a more concise way in order to fully comprehend the approach and methods used.

This chapter will begin by explaining the different objects and concepts used. It will then describe the constraints and objectives. To illustrate these definitions, an example of a solution will be presented.

2.1 Basic notions

2.1.1 System

The system is a railway station. They can have different configurations. Each one of them is composed of different resources with specific functionalities. Trains enter and exit this system using respectively arrival and departure sequences of given resources. These can be thought of as specific ways to enter or exit the system.

2.1.2 Time concepts

Planning horizon the planning horizon of the problem is a variable representing the number of days considered, denoted *nbDays*:

$$\text{nbDays} \in [1, ..., 14].$$

Time all actions in the problem take place at a given *time* which is given in string-format:

$$d_i\ hh : mm : ss,$$

where $d_i \in [1, nbDays]$, $hh \in [0, 23]$, $mm \in [0, 59]$ and $ss \in [0, 59]$. This implies that the time horizon is discretized and that the smallest time unit is 1 second.

Duration some actions take a given amount of time. This *duration* is also given in string-format:

$$hh : mm : ss,$$

where $hh \in \mathbb{N}$, $mm \in [0, 59]$ and $ss \in [0, 59]$.

2.1.3 Travel requirements

In order for a train to make a journey, it will need to travel a given distance during a given amount of time.

DBM the distance before maintenance refers to the travel distance and is expressed in kilometres.

TBM the time before maintenance refers to the travel time and is expressed using a duration.

2.2 Departures

Arrivals and departures are the most important events. The official document [4] defines a *Departure* event as:

> A departure is how the train leaves the system, the railway station. It is the beginning of the journey from a point of view of the passengers.

In order for a departure to occur at its departure time, one has to assign one and only one train to it. Not assigning a train to a departure is very penalizing, making it the most important objective. When no train is assigned to a departure, it does not take place. One cannot assign a train randomly to a departure, for the simple fact that each departure d, in the set of all departures $\mathcal{D}$, has some attributes that have to be taken into account:

depTime the departure time is the time at which the train must leave the platform.

depSeq the departure sequence of resources used to leave the system, e.g. by which way the train must leave the station. This sequence always starts from a platform.

prefPlat the platform on which the departure has to leave is not fixed, however, $prefPlat_d$ is the set of platforms on which the departure preferably takes place.

compCatDep the set of train categories that can be assigned to this departure.

idealDwell the ideal duration that the departure should stay on the platform.

maxDwell the maximum duration the departure may stay on the platform.

reqDBM the required distance for this departure. An assigned train must be able to cover this distance.

reqTBM the required duration for this departure. An assigned train must be able to travel during this amount of time.

jointDep a departure can be constituted of more than one train, noted a joint departure. All the departures of this joint departure have the same $jointDep_d$ variable.

2.2.1 Joint departures

If more than one departure ($n \geq 2$ where n is the number of joint trains) leaves the system linked, it is part of a joint departure. All joint departures leave from the same platform at the same departure time. Before being able to depart, the trains must be assembled in a specific order. More formally, each joint departure j, in the set of all joint departures $jDep$, is defined by an ordered list $jdList_j$. The order of this list is important: the first departure in the list has to be the leftmost (*side A*) train in the convoy. The trains are assembled from the left to the right, according to the $jdList_j$.
Every departure of a joint departure is described by its own event, however, the jointDep variable that links them indicates that they are not independent events:

idDep	depTime	depSeq	jointDep	...
Dep78	d1 13:36:00	DepSeq1	JointDep12	...
Dep79	d1 13:36:00	DepSeq1	JointDep12	...

Figure 2.1: Representation of a joint departure.

The figure 2.1 presents an example of a joint departure. The joint departure 12 is constituted of 2 departures, each having its own departure event. Some parameters in this example have been left out: $idealDwell$, $maxDwell$, $reqDBM$, $reqTBM$. For a joint departure, these parameters have the same values.

Furthermore, trains assigned to a joint departure have to be joined before they can depart if this is not already the case. For a convoy of n trains, $n - 1$ junctions are needed. Junction operations are sequential meaning two junctions can not occur at the same time.
A junction operation takes a certain amount of time $junTime$ which has to be taken into account because there has to be enough time to join trains before the departure. The trains have to be joined at least $minAsbDepTime$ before the departure which implies that a convoy cannot leave immediately after the last junction.

2.3 Trains

Having introduced the notion of trains briefly in the previous section, this section will provide a description of the properties of a train. In the problem, a train is defined as *a visit in the system of a rolling stock unit.*

> Rolling stock units are un-oriented, composed of railcars which may not be decomposed nor recombined with those of other units.

In the problem, the concept of a railcar does not exist, the concept of a *rolling stock unit* is used. This unit can be one or more railcars, but is considered as an indiscernible whole which has to be handled as such. A train is then the smallest existing rolling stock unit. This is a strong simplification, but note that today's modern rolling stock units are reversible (one can can drive them at both sides) and non-decomposable units, e.g. high-speed trains and intercity trains. However, older trains do not have these nice properties, but nevertheless, this detail will not be considered.
A train is a *visit* of a rolling stock unit and not the rolling stock unit itself. This implies that if the same rolling stock unit visits the system twice, this corresponds to two distinct trains. Reasoning about these two distinct visits can be done independently, however, a previous visit can be taken into account by altering some properties of the train.

Let's consider the set of trains denoted by $\mathcal{T}$. This set consists of two different mutually exclusive subsets:

init $\mathcal{T}_I$, initial trains are trains which are present in the system at the beginning of the horizon.

arr $\mathcal{T}_A$, arrivals are trains which are associated with arrivals in the system during the horizon.

The properties of each subset will be discussed in the following sections, but first, the concept of a *train category* will be introduced. Each train, whether belonging to $\mathcal{T}_I$ or $\mathcal{T}_A$, belongs to a specific category. These train categories have different properties. They correspond to different types of trains, which could be compared to inter-city, high-speed and local trains for example.

Let $\mathcal{C}$ be the set of all train categories. Each train category $c \in \mathcal{C}$ has the following properties:

length the length of the train is expressed in meters.

catGroup two trains are compatible if and only if their respective compatibility groups are identical. This implies that the physical and technical constraints allow two trains of different categories but with the same *catGroup* to be joined together. Trivially, trains of the same category can always be joined together.

maxDBM the maximum distance a train can travel between two maintenance operations of type "D". The notion of maintenance will be explained later.

maxTBM the maximum time a train can travel between two maintenance operations of type "T".

maintTimeT the time needed to perform a maintenance operation of type "T".

maintTimeD the time needed to perform a maintenance operation of type "D".

2.3.1 Initial trains

Some trains can already be present in the system at the beginning of the time horizon. They have the following attributes, for which an example is given on figure 2.2:

idTrain the id of the train.

category the category of the train.

resource the resource on which the train is initially placed at d_1 00 : 00 : 00.

remDBM the remaining distance the train is able to travel.

remTBM the remaining time the train is able to travel.

idTrain	category	resource	remDBM	remTBM
Train1247	Cat2	Yard2	1500	72:00:00
Train1248	Cat3	Yard1	1500	72:00:00

Figure 2.2: An example of the attributes of an initial train.

2.3.2 Arrivals

Arrivals are the end of journeys for passengers and correspond to entrances in the system. They arrive at a given, non-modifiable, time on a platform, but enter the system before that time because they use some resources (*arrival sequence*) before arriving on the platform. These arrival sequences are fixed. Unlike a departure which must not, but preferably, should be satisfied, an arrival always has to be handled. This consists in deciding on which platform it has to arrive and what to do next with the train .

Let's denote the set of arrivals during the horizon by $\mathcal{A}$. Each arrival $a \in \mathcal{A}$ has the following attributes:

arrTrain the id of the train associated to the arrival.

arrTime the arrival time represents the time at which the train has to arrive on a platform.

arrSeq the arrival sequence expresses the way the train should take to arrive on the platform.

prefPlat the set of platforms on which the train preferably should arrive.

jointArr this joint arrival parameter indicates whether this arrival is part of a joint arrival.

linkedDep the linked departure parameter indicates whether this arrivals is linked to a previous departure.

idealDwell the ideal dwell indicated the preferred duration the train should stay on the platform, in order for the passengers to have enough time to unboard.

maxDwell the maximum dwell indicates the maximum duration the train may stay on the platform.

remDBM the remaining DBM expresses the distance the train can still travel before doing any maintenance.

remTBM the remaining TBM expresses time the train can still travel before doing any maintenance.

Joint arrivals

As in the case of joint departures, an arrival can also consist of more than one train. Again, all the trains of the arrival are separate, but not independent trains. They are linked by a variable $jointArr$ which indicates that they are part of the same joint arrival. That means that as long as the trains are joint, they undergo exactly the same events and operations. Each joint arrival $j \in \mathcal{J}_{arr}$ is characterized by an ordered list $jaList_j$, where the order represents the trains from the left (side A) to the right (side B).
Joint arrivals can be assigned to joint departures of the same length, but they also can be disjoint. If a joint arrival has n trains, $n-1$ disjunction operations are needed to disjoin all of them. Each disjunction takes a certain amount of time $disjTime$ and must be performed sequentially.

Arrivals with linked departures

As mentioned previously, the same rolling stock unit can leave the system through means of a departure and re-enter after a certain amount of time as an arrival. This will be treated as two different trains. However, in certain cases one would like to update the information of the incoming train with the information of the previously departed train. This can be done using the variable $linkedDep_a$ associated to an arrival a. This variable contains the identifier of the previous departure which has the same rolling stock unit as the arrival a. Whenever an arrival a has a linked departure that has been assigned in the past, that is, a train t has been assigned to the linked departure d, the following information will be updated for this arrival:

$$\begin{aligned} remDBM_a &= remDBM_t - reqDBM_d \\ remTBM_a &= remTBM_t - reqTBM_d \\ cat_a &= cat_t \end{aligned}$$

Whenever the linked departure d has not been assigned yet, the information for a is not updated and keeps its initial values for its attributes. Otherwise, the train a that arrives in the system will inherit the category of the train t assigned to its linked departure d. The remaining distance and time of the arrival a will be altered by the information about the previous train t and the departure d. In the following example, some attributes of the arrivals and departures have been left out:

idArr	arrTrain	arrTime	linkedDep	cat	remDBM	remTBM
Arr2	Train2	d1 17:30:00		Cat6	870	65:30:00
...	...	...	...	...	...	...
Arr9	Train9	d1 20:35:00	Dep14	**Cat1**	**450**	**67:30:00**

idDep	depTime	reqDBM	reqTBM
Dep14	d1 18:05:00	120	02:20:00

Figure 2.3: Example of a linked departure schedule.

Example of a linked departure In figure 2.3, suppose that the train 2 arriving with arrival 2 is assigned to depart with departure 14. Given that in this case, Dep14 is covered, this will alter the category, remDBM and remTBM for the train 9 that arrives with arrival 9:

$$\begin{aligned} cat_{Train_9} &= Cat6 \\ remDBM_{Train_9} &= remDBM_{Train_2} - reqDBM_{Dep_{14}} \\ &= 750 \\ remTBM_{Train_9} &= remTBM_{Train_2} - reqTBM_{Dep_{14}} \\ &= 63:10:00 \end{aligned}$$

Suppose now that the departure 14 isn't covered. In that case, train 9 will keep its initial values, as indicated in bold on figure 2.3.

2.4 Maintenance

Trains need to be maintained regularly for different reasons. Maintenance resets the DBM or TBM of trains. There are two different types of maintenance:

Maintenance type T this type of maintenance has an influence on the TBM of a train. It is mostly done for comfort: cleaning, small reparations,... In order to undergo a maintenance of type T, a train has to go to a Facility of type T that is compatible: the facility must be long enough and must be able to receive a train of category *cat*. Whenever a train undergoes maintenance of type T, its *remTBM* is re-initialized to $maxTBM$,this maximum value depends on the category of the train.

Maintenance type D this type of maintenance has an influence on the DBM of a train. It is mostly done for security reasons: check-up, technical reparations,... Maintenance of type D is done on a Facility of type D that is compatible. Afterwards, the *remDBM* is re-initialized to its maximal value $maxDBM$, also depending on the train's category.

Depending on the category of the train, the maintenance takes $maintTimeT_{cat}$ or $maintTimeD_{cat}$. The number of overall maintenances of type T and D is limited per day, expressed by the parameter $maxMaint$.

2.5 Infrastructure resources

Each railway station is a complex network and has been simplified in connected blocks of resources. In this section, the properties and compatibilities of each type of resource will be explained in detail. Furthermore, how trains can move between the different resources will be explained at the end of this section. A representation of a system is depicted on figure 2.4.

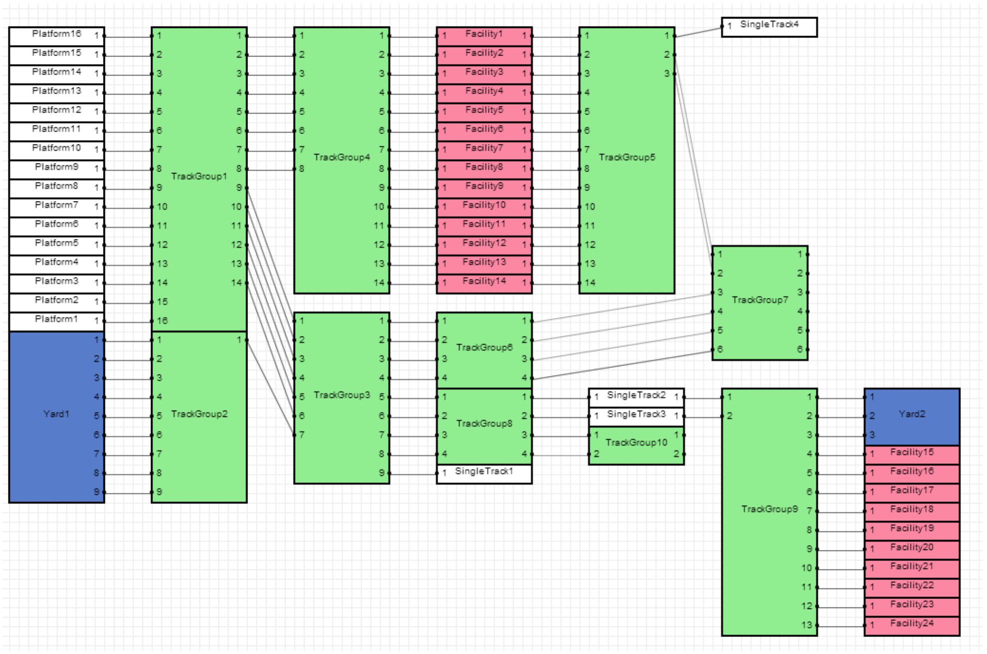

Figure 2.4: A global view of an example of a system. This image can be found in a bigger format in the appendix A.

In reality, trains can be on two resources at a time, for example when the train is long and when it is moving from one resource to another. However, for the sake of simplicity, the hypothesis is made that trains move from one resource to another instantly.

For every resource, there is a list of compatible train categories. This implies that some resources cannot be used by trains for which the category is not in its compatibility list.

2.5.1 Single tracks

A single track is a track where passengers cannot board or unboard. Each single track s has the following properties:

length the length of the single track. The total length of all the trains on this track can never exceed it.

capa the capacity of the single track in number of trains. This number can never be exceeded.

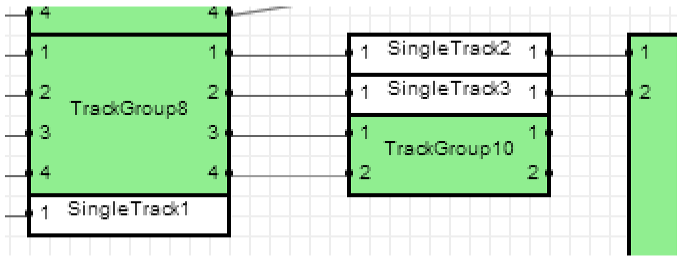

Figure 2.5: Example of a disposition of some single tracks.

An example of a representation of a single track can be found on figure 2.5. A single track can be either a double-ended queue (entrance/exit points at both sides) or a stack (only one entrance/exit point). This is a physical limitation and because trains cannot fly over one another, the order in which trains are put on a single track defines the order in which the trains can be retrieved from it. Single tracks can be used as passing-through or storage unit.

2.5.2 Platforms

A platform is very similar to a single track, with 2 main differences:

- Passengers can **board** and **unboard** on a platform. Hence, only platforms can be used for arrivals and departures.

- There is no capacity limit, only a maximum length.

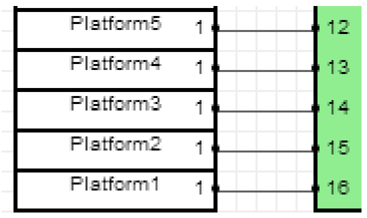

Figure 2.6: Example of a disposition of some platforms

An example of a representation of a platform can be found on figure 2.6. Platforms can be used as passing-through, arrivals, departures and limited storage.

2.5.3 Maintenance facilities

Maintenance facilities are tracks inside maintenance workshops. A facility f has the following properties:

type the type of a facility can be "D" or "T". Only operations of type T can be done on facilities of type T and likewise for the facilities of type D.

length the total length of the facility, which can not be exceeded by the sum of the lengths of all the trains on it.

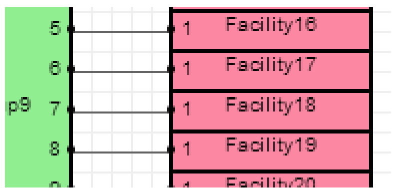

Figure 2.7: Example of a disposition of some facilities

An example of a representation of a facility can be found on figure 2.7. As with a single track, a facility can be a double-ended queue or a stack. The order of the trains on it cannot change. Facilities can be used for passing-through, maintenance and storage.

2.5.4 Yards

A yard is a collection of tracks and switches, mainly used for parking. The order of the trains on a yard is off no importance except when a train is (dis)joint on a yard. A train can stay on a yard during an unlimited amount of time. A yard y has one attribute:

capa the capacity of the yard is a not the real maximum number of trains able to go in the yard, for more physical and technical reasons. Therefore, it is possible to exceed this number.

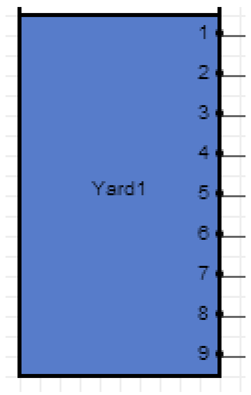

Figure 2.8: Example of a disposition of a yard.

An example of a representation of a yard can be found on figure 2.8. Yards can be used for passing-through and storage.

2.5.5 Track groups

A track group is a set of tracks used by a train to move throughout the system. It is a simplified representation of a physical configuration with a lot of different tracks and switches. A track group always has two sides (A and B) and has gates on both sides where trains can enter or exit. In the track group, all possible ways from one gate on side A to one gate on side B are considered. If there are g_1 A gates and g_2 B gates, the track group contains $g_1 \cdot g_2$ possible ways a train can take.
In each track group k, a train needs $trTime_k$ to go from one side to another. A train cannot change direction in a track group: if it enters one side, it has to leave at the other.

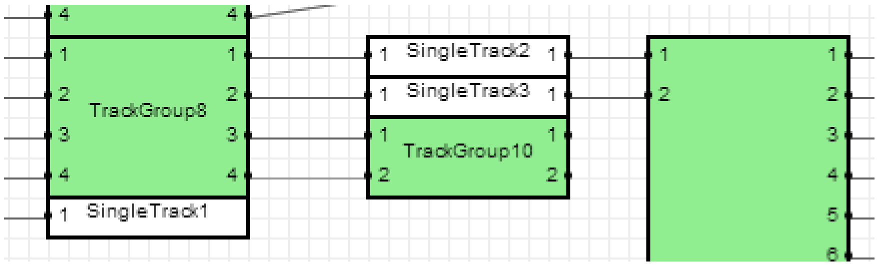

Figure 2.9: Example of a disposition of several track groups.

An example of a representation of a track group can be found on figure 2.9. The track group 10 on this figure has two left and two right gates. The right gates allow to enter or exit the system as they are not linked to other resources. No storage is possible.
Ideally, there should be a security time margin $hwTime_k$ between two trains that intersect or that have at least one gate in common when passing through the track group, but this is not mandatory.
Trains cannot intersect on a track group, physically this implies that one will have to stop to leave a security distance. In this problem however, two trains will be able to intersect, but it will result in a penalty cost.
When several trains enter a track group and do not intersect, they can easily go on without any intersection cost.

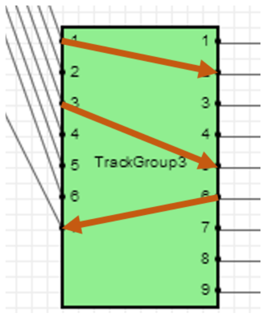

Figure 2.10: This figure shows how several trains can use a track group during overlapping time-intervals. As long as the arrows don't intersect one another, there is no intersection cost.

Enter and exit the system The only way for a train to enter or exit the system, is by means of some specific track groups. These track groups do not have a neighbour resource and are the entry and exit points of the system.

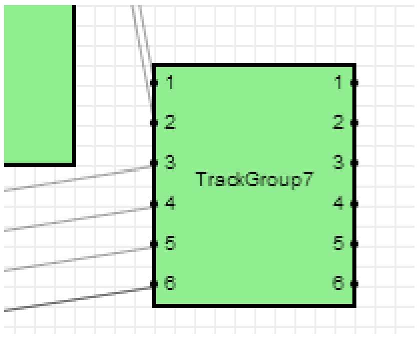

Figure 2.11: Illustration of a track group providing entry or exit points for a train. It has no neighbour resources at the right side.

2.5.6 Transition between resources

Each resource can have gates on one or two sides. A gate is a physical track linking this resource to a neighbouring resource. The gate is undirected, so if it goes from resource a to b, it can go from b to a. Only yards and track groups can have more than one gate per side, which means that there are different ways to enter the same resource. For the other resources, there is at most one gate per side.

2.6 Constraints

A solution, e.g. a schedule for every train during the horizon, is only feasible when it satisfies a set of constraints that will be detailed in this section.

2.6.1 Schedule properties

The schedule representation has to satisfy a few constraints.

– Each schedule of any train must always begin with a `EnterSystem` event and end with a `ExitSystem` event.

– When a train goes to a resource, its schedule must always begin with a `EnterResource` event. If it leaves this resource, for example to go to another resource or to exit the system, this must always begin with a `ExitResource` event.

Between resources, several operations can take place:

– The resource can be used to go from one resource to another. In that case, the time between the EnterResource and ExitResource event is *trTime* if the resource is a track group. If not, there are two possibilities:

  Traversal The train goes from one side to another, and the minimum duration is *minResTime*.

  Reversal The trains changes direction on the resource. The minimum duration is *revTime*. The reverse time is the time the driver needs to get out of the train and walk to the other side.

– The resource is a facility and is used for maintenance. If the train undergoes maintenance, the minimum time between the EnterResource and ExitResource event is *maintTimeT* or *maintTimeD*, in function of the type of maintenance. The maintenance operation is always preceded by a `BegMaintenance` and followed by a `EndMaintenance` event which occurs after *maintTime*. The maintenance can begin from the moment the train enters the resource, but it can also happen after it enters. After the maintenance, the train can leave immediately by exiting the resource or can stay on the facility.

– The resource is a platform and is used for an arrival or a departure. In that case, an `Arrival` respectively `Departure` event occurs at the moment the train enters respectively leaves the resource. The time between the EnterResource and ExitResource events can vary, but is bounded by the *maxDwell* time for the given arrival or departure. The minimum duration the train can stay on the platform is *minResTime* if the train goes from one side to another and *revTime* if the train leaves the resource by the same side it entered.

– The resource is a facility, platform, single track or yard and two or more trains enter the resource for a (dis)junction operation. The time between the EnterResource and ExitResource event depends on whether or not the given train has to wait for another train before being (dis)joint, for the (dis)junction time and for the remaining time the train stays on the resource. Each (dis)junction event is preceded by a `BegJunction` or `BegDisjunction` event, followed by a `EndJunction` or `EndDisjunction` which occurs exactly after *junTime* respectively *disjTime*. If a train is being joint twice (for example, a train at his left and at his right), there will be two BegJunction/EndJunction events in its schedule.

Example of a schedule

Train	Time	Event type	Resource	Gate	Complement
Train2	d1 06:01:50	EnterSystem	TrackGroup10		
Train2	d1 06:01:50	EnterResource	TrackGroup10	B2	
Train2	d1 06:02:50	ExitResource	TrackGroup10	A2	
Train2	d1 06:02:50	EnterResource	TrackGroup8	B4	
Train2	d1 06:05:50	ExitResource	TrackGroup8	A4	
Train2	d1 06:05:50	EnterResource	TrackGroup3	B8	
Train2	d1 06:06:20	ExitResource	TrackGroup3	A6	
Train2	d1 06:06:20	EnterResource	TrackGroup1	B14	
Train2	d1 06:07:00	ExitResource	TrackGroup1	A14	
Train2	d1 06:07:00	EnterResource	Platform3	B1	
Train2	d1 06:07:00	Arrival	Platform3		Arr2
Train2	d1 06:35:00	Departure	Platform3		Dep16
Train2	d1 06:35:00	ExitResource	Platform3	B1	
Train2	d1 06:35:00	EnterResource	TrackGroup1	A14	
Train2	d1 06:35:40	ExitResource	TrackGroup1	B14	
Train2	d1 06:35:40	EnterResource	TrackGroup3	A6	
Train2	d1 06:36:10	ExitResource	TrackGroup3	B8	
Train2	d1 06:36:10	EnterResource	TrackGroup8	A4	
Train2	d1 06:39:10	ExitResource	TrackGroup8	B4	
Train2	d1 06:39:10	EnterResource	TrackGroup10	A2	
Train2	d1 06:40:10	ExitResource	TrackGroup10	B2	
Train2	d1 06:40:10	ExitSystem	TrackGroup10		

Figure 2.12: The complete schedule of Train 2 which arrives and immediately departs after.

Train	Time	Event type	Resource	Gate	Complement
Train22	d7 22:19:40	EnterResource	Yard2	A3	
Train22	d7 22:19:40	BegDisjunction	Yard2		Train21+Train22
Train22	d7 22:24:40	EndDisjunction	Yard2		Train22
Train22	d7 23:59:59	ExitResource	Yard2	A1	
Train22	d7 23:59:59	ExitSystem	Yard2		

Figure 2.13: A part of the schedule of Train 22 which is part of a joint arrival. After the arrival, the trains go to the yard and are disjoint. The train 22 is not assigned and it exits the system at the end of the horizon.

2.6.2 Departure assignments

For an assigned departure to be valid, different constraints have to be respected:

- Clearly, at most one train can be assigned to a departure.
- The remaining DBM and TBM, with or without resetting by maintenance, from the train t have to be sufficient for the departure d:
  - $remDBM_t \geq reqDBM_d$,
  - $remTBM_t \geq reqTBM_d$.
- The category of the train t has to be compatible with the departure d: $cat_t \in compCatDep_d$, each departure has a list of compatible categories.
- For a joint departure, not only do the trains have to be compatible with the departures, but the trains must also be compatible with one another. For each (t, t') assigned: $catGroup_t = catGroup_{t'}$.

2.6.3 Resource usage

Trains can use resources under certain conditions:

Compatibility for each train t that uses a resource r, the resource has to be compatible with the trains category: $cat_t \in compCatRes_r$ where $compCatRes_r$ is a list of categories which are compatible with the resource.

Length the *track length* that is defined for each single track, maintenance facility and platform may not be exceeded by the length of the trains on it. Let's define $\mathcal{T}_{r,h}$ the set of trains using the resource r at time h, this constraint can be written as:

$$\sum_{t \in \mathcal{T}_{r,h}} length_t \leq length_r.$$

Capacity each single track s has a maximum capacity $capa_s$, expressed in number of trains. At any given time, the number of trains on s may not exceed $capa_s$:

$$\sum_{t \in \mathcal{T}_{s,h}} 1 \leq capa_s.$$

maxDwell whenever a platform is not used for a arrival or departure, it can be used for at most *maxDwellTime*. When it is used for an arrival a or a departure d, this is limited by $maxDwell_a$ respectively $maxDwell_d$. When an arrival is directly followed by a departure (e.g. the train of the arrival is assigned to the departure), the maximum dwell time is $\max(maxDwell_a, maxDwell_d)$.

minDwell the minimum duration for a resource, except for a track group is *minResTime*.

Imposed Consumptions in railway stations, trains other than passenger trains regularly pass through. They will be considered in this problem as *Imposed Consumptions*. For each imposed consumption $i \in \mathcal{I}$ on a resource res_i during $[beg_i, end_i[$, there can be no trains using the same resource.

maxMaint the number of maintenances per day, whether of type T or D, is limited by *maxMaint*.

Order the train order on individual tracks is defined by the order they arrive on it. A train cannot move if it is blocked by another one on one or both of its sides.

(Dis)junction order the relative positions of trains on a yard are only important for (dis)junction operations. This implies that if a yard only has gates at side B, in order to join *Train1+Train2*, Train1 has to arrive before Train2. In order to join *Train2+Train1*, Train2 has to arrive earlier.

Joint schedule during the period trains are assembled, they must have exactly the same schedule.

2.7 Objectives

The objective function contains three distinct functions to minimize:

$$\min\{f_1, f_2, f_3\}$$

where:

1. f_1 : number of uncovered departures,
2. f_2 : number of conflicts on track groups and yard overload,
3. f_3 : performance costs.

The objective is f_1 is more important than f_2, which is more important than f_3.

2.7.1 Objective 1: Uncovered departures

A departure is considered as *uncovered* when there is no train assigned to it. This implies that the departure cannot take place. When a joint departure of length n is partially covered by m trains, with $m < n$, then this results in $n - m$ uncovered departures.

$$f_1 = \text{number of uncovered departures}$$

2.7.2 Objective 2: Conflicts on track groups and yard overload

Conflicts on track groups

Trains going in the **same direction** on intersecting paths can create costs whenever the buffer time between them at any moment on the track group is insufficient. First of all, let's define what is an intersecting path. For that, the concept of a path in a track group must be defined:

> A path in a track group is the path a train takes to go from one side of the track group to another one. This path is defined by the side where the train originates ($side_o$) and the side to where the train is destined ($side_d$). A side can be A (left) or B (right). Whenever $side_o$ is A, then $side_d$ is B and reciprocally. A track group also has several gates at both sides. The gate from which the train originates is defined by a number indicating its position ind_o, as goes the same for the destination gate ind_d.

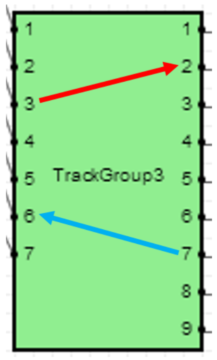

Figure 2.14: Representation of two paths going in opposite directions.

On figure 2.14, two paths are represented. The red one is defined by

$$side_{o_r} = A, side_{d_r} = B, ind_{o_r} = 3, ind_{d_r} = 2$$

while the blue one is defined by

$$side_{o_b} = B, side_{d_b} = A, ind_{o_b} = 7, ind_{d_b} = 6.$$

Two trains intersect when their paths' origin or destination gate is the same, or when the lines connecting their origin and destination gates intersect. More formally, two trains who are going in the same direction ($side_{o_1} = side_{o_2}$) intersect when

$$(ind_{o_1} - ind_{o_2}) \cdot (ind_{d_1} - ind_{d_2}) \leq 0.$$

At any time h_1, h_2 the trains are on a track group k, if $|h_1 - h_2| \leq hwTime_k$, this represents a conflict cost. Trains going in **opposite directions** can also create costs. Two trains who are going in opposite directions ($side_{o_1} \neq side_{o_2}$) intersect when

$$(ind_{o_1} - ind_{d_2}) \cdot (ind_{d_1} - ind_{o_2}) \leq 0.$$

At any time h_1, h_2 the trains are on a track group k, if $|h_1 - h_2| \leq trTime_k + hwTime_k$, this represents a conflict cost. Indeed, when trains go in opposite directions on an intersecting path, one train has to wait until the other one has left the resource.

Let's define $\mathcal{K}$ the set of all track groups and $\mathcal{P}$ the set of all paths.

$$\begin{aligned} nbConflicts \;=\; & |(p_1, p_2, k) \in \mathcal{K} \times \mathcal{K} \times \mathcal{P} \text{ s.t. } side_{o_1} = side_{o_2} \wedge \\ & (ind_{o_1} - ind_{o_2}) \cdot (ind_{d_1} - ind_{d_2}) \leq 0 \wedge |h_1 - h_2| \leq hwTime_k| \\ + \; & |(p_1, p_2, k) \in \mathcal{K} \times \mathcal{K} \times \mathcal{P} \text{ s.t. } side_{o_1} \neq side_{o_2} \wedge \\ & (ind_{o_1} - ind_{d_2}) \cdot (ind_{d_1} - ind_{o_2}) \leq 0 \wedge |h_1 - h_2| \leq trTime_k + hwTime_k| \end{aligned}$$

Yard overload

Every yard $y \in \mathcal{Y}$ has a capacity $capa_y$ in number of trains. This capacity can be exceeded, but it represents a cost. For any event $e \in \mathcal{E}$ in the train schedule of t_e, if:

$r_e = y$ the resource on which the event takes place is a yard,

h_e that occurs at a given time,

$s_e = EnterResource$ for which the description of the event is a `EnterResource`,

a cost applies when $capa_y$ is exceeded. The same counts for an imposed consumption $i \in \mathcal{I}$ that begins on yard y.

$$\begin{aligned} nbYardOverload &= |(e,y) \in \mathcal{E} \times \mathcal{Y} \text{ s.t. } s_e = \texttt{EnterResource} \wedge \\ &\quad r_e = y \wedge nbTrain_{y,h_e} > capa_y| \\ &+ |(i,y) \in \mathcal{I} \times \mathcal{Y} \text{ s.t. } res_i = y \wedge \\ &\quad nbTrain_{y,beg_i} > capa_y| \end{aligned}$$

Finally, the second objective function can be written as:

$$f_2 = nbConflicts + nbYardOverload$$

2.7.3 Objective 3: Performance costs

The third objective is a sum of individual costs.

$$f_3 = f_3^{maint} + f_3^{jun} + f_3^{plat} + f_3^{pref} + f_3^{reuse}$$

Over-maintenance cost: f_3^{maint} is the cost related to the maintenance and is designed in a way that applying maintenance to a train with still a high amount of remaining TBM or DBM is very costly.

$$\begin{aligned} f_3^{maint} &= \sum_{e \in \mathcal{E}, s_e = \texttt{BegMaintenance}, \text{“}T\text{”}} remTCost \cdot remTBM_{t_e} \\ &+ \sum_{e \in \mathcal{E}, s_e = \texttt{BegMaintenance}, \text{“}D\text{”}} remDCost \cdot remDBM_{t_e}. \end{aligned}$$

Train (dis)junction operation cost: f_3^{jun} is the cost of doing a (dis)junction.

$$f_3^{jun} = \sum_{e \in \mathcal{E}, s_e = \texttt{BegJunction}} junCost + \sum_{e \in \mathcal{E}, s_e = \texttt{BegDisJunction}} disjCost.$$

Platform usage costs: f_3^{plat} is a penalty cost for staying too long or not long enough on a platform. Indeed, staying too long prohibits other trains from using this platforms and not staying long enough can be uncomfortable for the passengers when (un)boarding. A platform is used in three cases:

Arrival arrival a which has an ideal dwell time $idealDwell_a$,

Departure departure d which has an ideal dwell time $idealDwell_d$,

Arrival+Departure arrival a is followed immediately by a departure d. Let $z = (a, d)$, the ideal dwell time is:

$$idealDwell_z = idealDwell_a + idealDwell_d.$$

Let $\mathcal{A}^*$ be the set of arrivals without immediate departures, $\mathcal{D}^*$ be the set of departures that is not an immediate departure after an arrival and $\mathcal{Z}$ the set of arrivals followed by an immediate departure.

$$\begin{aligned} f_3^{plat} &= \sum_{a \in \mathcal{A}^*} dwellCost \cdot |dwell_a - idealDwell_a| \\ &+ \sum_{d \in \mathcal{D}^*} dwellCost \cdot |dwell_d - idealDwell_d| \\ &+ \sum_{z \in \mathcal{Z}} dwellCost \cdot |dwell_z - idealDwell_z| \end{aligned}$$

Non-satisfied preferred platform assignment cost: f_3^{pref} is a cost that is applied when an arrival or departure is not assigned to one of its preferred platforms, represented by $prefPlat_a$ and $prefPlat_d$ respectively.

$$f_3^{pref} = \sum_{a \in \mathcal{A}, plat_a \notin prefPlat_a} platAsgCost + \sum_{d \in \mathcal{D}, plat_d \notin prefPlat_d} platAsgCost$$

Non-satisfied train reuse cost: f_3^{reuse}. Some matchings between a train and a departure are given, they are called *reuses*. If a reuse $u \in \mathcal{U}$ is not satisfied, this induces a cost.

$$f_3^{reuse} = \sum_{u \in \mathcal{U}, train_{dep_u} \neq train_{arr_a}} reuseCost$$

Chapter 3

Different approaches

As explained in the introduction, the problem is part of a challenge[6]. The groups who wanted to participate had to respect some requirements about the solver they would develop. The main requirement is a time limit of 10 minutes. Even if participating is not an objective, all the constraints are nevertheless respected in order to have benchmarks to compare the obtained solution with. This constraint is a really restraining one because it does not give much time to handle all the data and solve the problem.
Several approaches were tested and some of them were rapidly discarded because they take too much computation time.

The first considered approach consists in modelling the problem as a matching problem. The matching is modelled as a bipartite graph,- with the trains on the one side and the departures on the other.
An other considered approach restrains the possibilities to a few configurations and uses the constraint programming solver OscaR[3].
As all the previous trials led to problems with space and time complexity, a third approach which consists in solving the problem in a greedy manner is considered.

Those three approaches are quite global and require some parsing and pre-processing methods to solve the entire problem. Reading and parsing the data were working properly for the first approach and are re-used for the other ones. The more complex required method to solve the problem is the one which computes the ways between the different resources, taking into account the specific constraints (resource compatibility, no reversal in a track group nor between resources, no conflicts outside track groups and yards: the order of arrivals implies the order of departures).

To compute the ways, the first approach is to use an existing mixed integer programming (MIP) solver LPSOLVE using the OscaR[3] interface to compute the shortest way in the graph which represents the infrastructure of the problem. Two different representations of the graph are considered to express the no-reversal constraints. The no-reversal limitation is first added as a constraint in the model to optimize. The second representation splits each resource into two nodes: one for the arrival from the left side, one for the arrival from the right side.

The obtained results are not as good as expected in terms of time complexity as the MIP solver is not designed for this kind of problem. Besides, computing the ways between all the resources is not relevant as some resources can be grouped. For example, all the platforms in the visualization in figure 3.5 can be grouped in the ways computation. Furthermore, all the track groups within a way can be seen as a black box. A train can indeed never stop on a track group. The only ways that need to be computed are those between the resources that are not track groups. The size of the considered graph can then be reduced by considering groups of all resources but track groups as nodes and groups of track groups as edges. This graph is considerably smaller and easier to deal with. The computation of a path is done using the Dijkstra algorithm[1]. This new approach allows more flexibility and reduces significantly the time and space complexity.

3.1 General

These approaches, their respective strengths and weaknesses will be discussed here in more detail.

3.1.1 Matching

The first approach considers the problem as a maximal matching problem. The first lexicographic objective indeed maximizes the number of covered departures. Let's reconsider the problem from the matching point of view while neglecting the other lexicographic objectives.
There are two types of trains: those initially in the system and those that arrive in the system during the time horizon. These trains have several characteristics: their category and the remaining time/distance before they have to undergo a maintenance. The system is composed of different resources: platforms, yards, maintenance facilities and single tracks which are linked by track groups.
A maximum number of departures should be covered by a train and a train can cover at most one departure. Each departure has several train specifications:

the compatible categories and a minimum remaining time/distance.
For this approach, the problem is considered step by step. First, trains are assigned to departures and then the resources are dealt with.
This problem is represented as a bipartite graph with the trains (initials trains and arrivals) on the first side and the departures on the other side, as shown on figure 3.1.

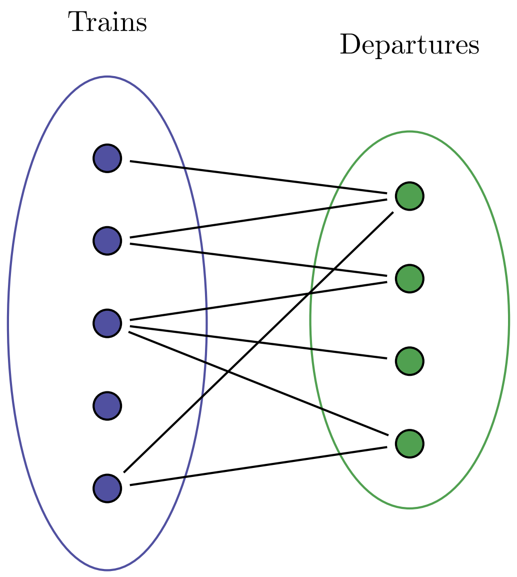

Figure 3.1: Illustration of a bipartite graph representing the trains and the departures.

The objective is to achieve a minimal cost maximal matching. The edges representing the trains which are not compatible or not available at the departure time have an infinite cost.
The Hungarian algorithm[9] is used to compute the maximal matching. A java implementation is available in [7].
This approach first seems suitable, however they are some feasibility issues. Some arrivals are indeed linked to departures. In other words, if those departures are covered, the characteristics of the linked arrivals are modified. Some assignments thus influence the feasibility of other ones. To solve this issue, a penalty cost is introduced and the matching is relaunched, but it is not efficient.
After this matching is done, each arrival and departure needs to be assigned to a platform, trains that do not have a departures following their arrival need

to go to a compatible storage unit and trains needing maintenance need to go on an available and compatible facility. This assignment is implemented using the constraint programming solver OscaR[3].

3.1.2 Constraint programming

The constraint programming paradigm is used in two different situations. The first one is as a part of the matching problem. It is used to schedule the platforms for each arrival/departure after matching the trains to the departures and also to assign the maintenance facilities and storage units. After discarding this matching approach, an attempt was made to solve the entire problem using constraint programming.

Scheduling As explained in the previous section, the constraint programming paradigm is used to assign platforms, yards and maintenance facilities to trains between arrivals and departure sequences. The matching fixes the assignment of trains to departures and maintenances. The resources' utilisation is assigned sequentially:

1. assign platform for arrivals and departures,
2. assign maintenance facilities according to trains needing maintenance,
3. assign yards according to the trains' needs.

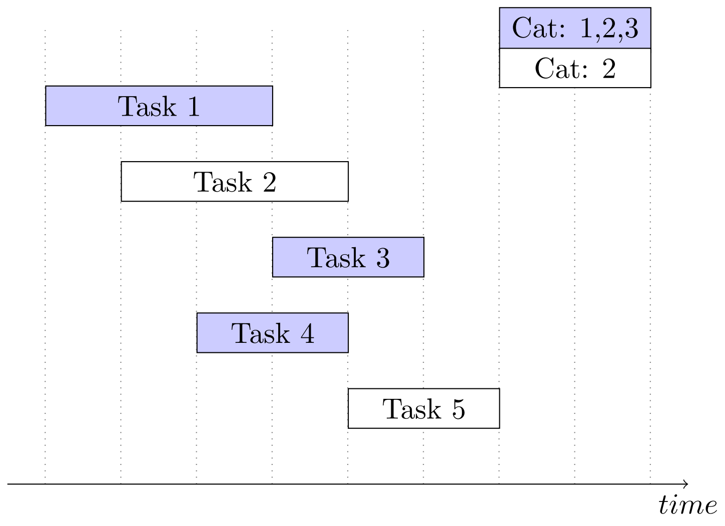

Figure 3.2: Example of five tasks that needs a resource: a platform, facility or yard. The colors indicate which resources are compatible with the train.

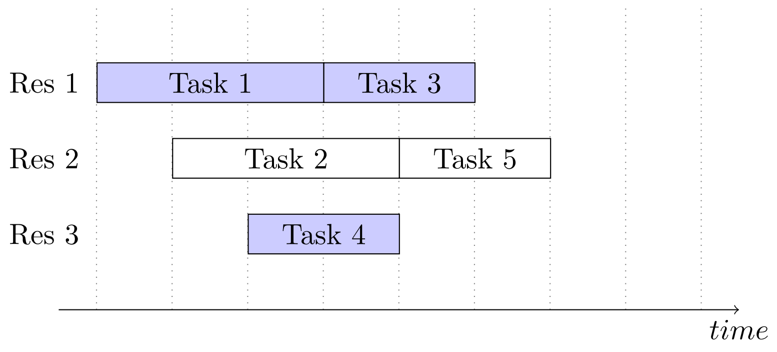

Figure 3.3: Compatible assignment.

An example of an assignment problem is shown on figure 3.2. The instances solved concern a larger horizon with more trains. However, this depicts a typical example. A feasible schedule is given on figure 3.3.

A first issue encountered is linked to the infrastructure. Resources of the same type are indeed not always neighbours and this lead to feasibility issues because the way to reach them are not the same and do not take the same amount of time. This problem is fixed by taking the ways into account in the model, but it complexify it and lead to a space complexity overload.

Solving As the matching do not take into account the relations between the assignments and the fact that an assignment can influence an other one, an attempt is made to solve the problem in a more global way. Given the previous time and space complexity issues, some restrictions are needed as this approach was even more complex. The simplification states that train storage can only occur in yards, as they can be overloaded. Limiting the overload of yards is indeed a second lexicographic objective and thus less important. Besides, if needed, a maintenance will always occurs just before an assigned departure.
The problem is unfortunately too complex to be optimized. The huge number of variables and constraints does not allow the constraint programming solver to find a solution within the time requirement.
Constraint programming could be efficient to find one solution, but the search space was too big to find the best one within the time requirement. There are indeed a lot of equivalent solutions considering all departure assignments permutations, but also all ways and gates permutations.

3.1.3 Greedy

As time complexity is the major issue for the previous approaches, designing a simpler approach is therefore considered. This approach consists in splitting the horizon in independent days and making greedy assignments.
Each day, there are initial trains and arrivals. The considered approach consists in using a divide-and-conquer method and proceeds step by step.

1. Assign simple arrivals to simple departures:
   (a) without maintenance nor storage,
   (b) with maintenance and/or storage.
2. Assign joint trains to joint departures:
   (a) without maintenance nor storage,
   (b) with maintenance and/or storage.
3. Assign remaining departures with junction and/or disjunction.

These steps are depicted in the block diagram on figure 3.4.

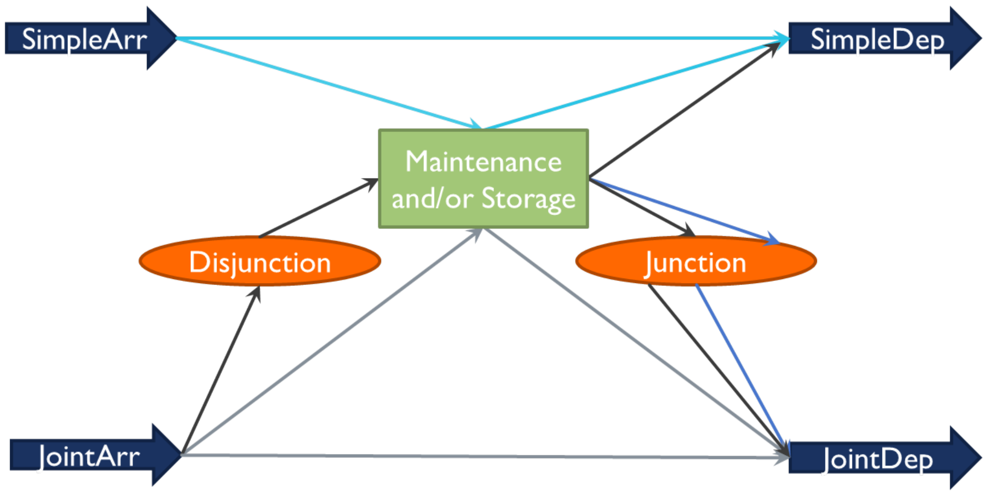

Figure 3.4: Illustration of the divide-and-conquer greedy approach.

This approach will be further detailed later as it is the chosen approach to solve the problem.

3.2 Ways

To visualize and understand the model, the specific infrastructure of the $A1$ instance shown on figure 3.5 will be used. Of course, the implementation is working for all instances of the problem and thus all infrastructures.

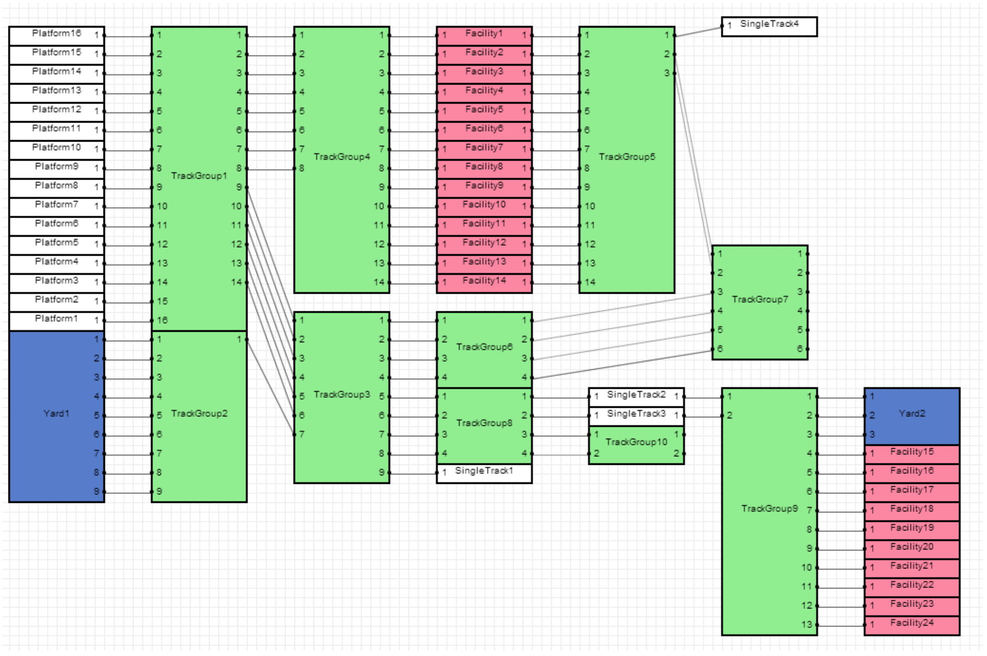

Figure 3.5: Visualization of the infrastructure for the A1 instance.

3.2.1 Simple graph

This first approach is the exact representation of the infrastructure as given in the data. Each gate of each resource is a node in our graph. Edges represent the link between two resources or between two gates of the same resource. Some nodes are internal edges which links two opposite gates of a resource while other one are external nodes linking two resources. The graph is undirected and constraints are added to ensure that a train cannot reverse on a track group nor between two resources. A graphical representation of a part of the infrastructure can be seen in figure 3.6.
The graph is quite huge as there are 55 resources, 103 external edges and 603 internal edges. Therefore, the model is not straightforward as forbidding the reversals within track groups and between resources is still needed to ensure the model consistency.

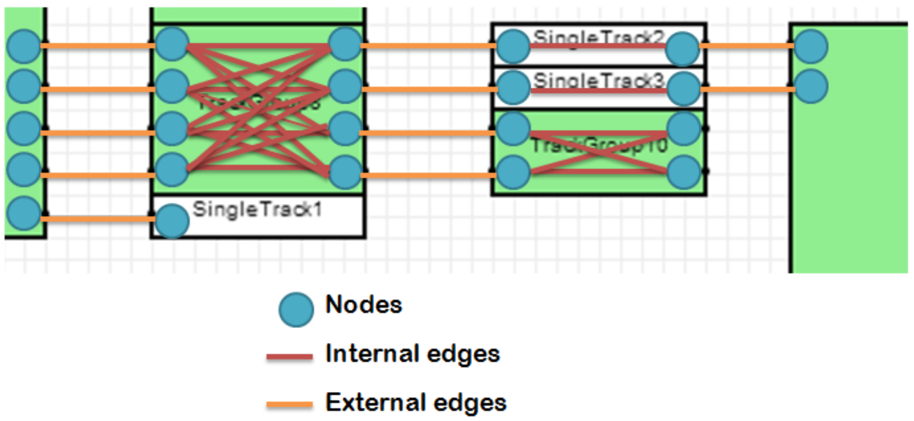

Figure 3.6: Simple graph for a part of the A1 instance.

Computing ways in this way is way too slow. There are indeed a lot of gates permutations which are not relevant. The solver often needs to compute a way between two resources and computing it each time is too costly. Other approaches are then considered to solve this problem.

3.2.2 Split graph

The second approach is an improvement of the first one. As explained, it is not relevant to consider gates permutations in a resource. The only resources with multiple gates on one side are yards and track groups. Choosing a gate instead of an other one which is linked to the same resource could only imply a conflict which is penalized in the second lexicographic objective which is neglected in this approach.
Besides, sometimes, only the time to go from one resource to another is needed and dealing with conflicts and choice of gates is not really relevant. Therefore, the number of nodes and the number of edges in the graph can be drastically reduced. Only one node per side of a resource is indeed needed and it could even be reduced to one node per resource because the two nodes in the same resource are linked. However, all the resources are split into two nodes to simplify the constraints: the first one representing this resource when entered by the right side, the second one representing it when entered by the left side. For all the resources except the track groups, the two nodes of the same resource are linked. You can see in figure 3.7 the nodes and the edges for two

single tracks and a track group.

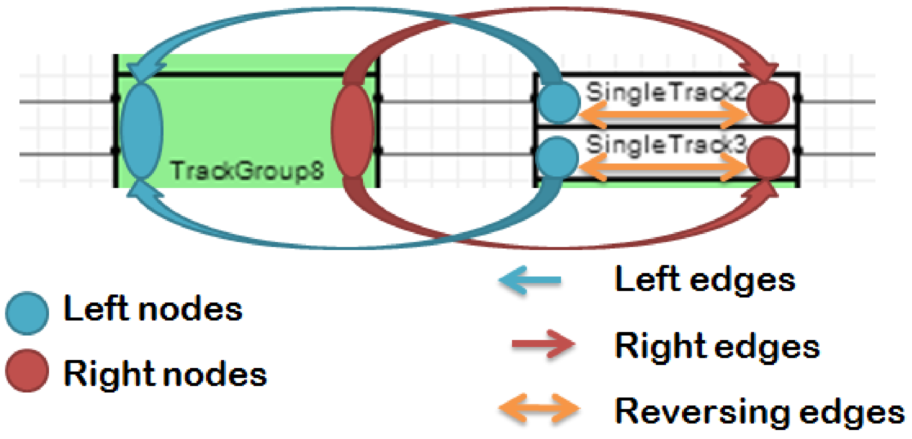

Figure 3.7: Representation of the split graph for two single tracks and a track group.

This approach is more efficient than the previous one because there are no more permutations between gates. It reduces the number of optimal solutions. However, this approach is still not efficient enough to face the computation time limit.

3.2.3 Grouped graph

Two non track group resources will always be linked by at least one track group. Those are only considered as a way with constant travelling time as a train cannot stop in a track group. Track groups can then be considered as edges as they will never be a starting nor an arrival resource.
The only nodes to be considered are the resources that are not track groups and the only edges are shrinkages of track groups.
Furthermore, the graph representation could be simplified by considering resources which have the same neighbour(s) as one node. Figure 3.8 shows that all the platforms of the A1 instance have the same neighbour, the track group 1. They could therefore be grouped in one single node. The approach can then be simplified: instead of computing the way between two resources, the way between their two groups can be computed. This modification allows to reduce the number of nodes and the number of edges in the graph considerably.

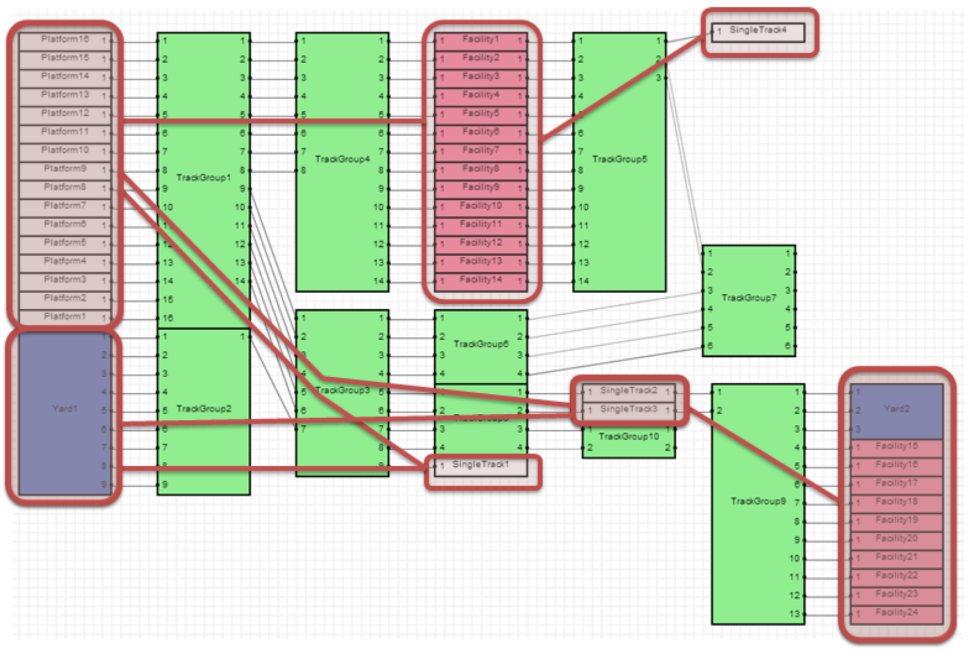

Figure 3.8: Grouping non track group resources for the instance A1.

Figure 3.8 shows that grouping the resources and using track groups as edges considerably simplifies the problem. Indeed, this grouped representation has only 7 nodes and 7 edges to consider while the simple graph has 55 resources and 706 edges.
As the new graph is small, there are two possible options:

- Compute all the shortest ways between two groups when launching the program to avoid further computations.
- Compute the ways when queried.

As the required time to compute a way is quite negligible, both approaches are used because the second one allows to have more complex requests. As a matter of fact, the shortest way cannot always be used because it uses a resource to do a reversal or traversal which can be non compatible with the train or which is not available at the time. Computing all ways in advance does not cover all the possible ways that might be needed.

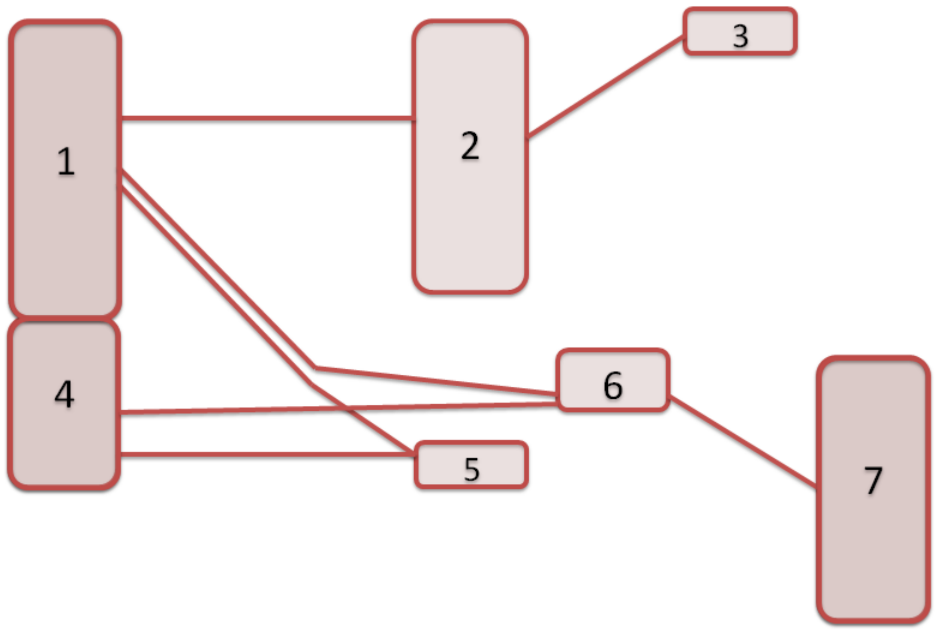

Figure 3.9: Representation of the group graph for instance A1.

For example, if the shortest way between the group 1 and group 4 includes a reversal in group 5, the reversal can also be done in group 6. Considering all possible ways can be useful when dealing with the compatibility but also the availability of the resources.

This last approach is the one used for the final algorithm as it gives the best results, both in time and space complexity. It is also more adapted because it takes into account the availability and compatibility of the resources when computing a way for a given train.

Chapter 4

Solution

After several attempts, the final solution consists of 4 distinct parts:

1. The parsing of the data into **objects**. All these objects have distinct properties and functionalities.

2. An **Oracle** which verifies whether or not certain operations (e.g. maintenance) can take place and is able to tell not only whether or not an operation is possible, but also where and when it is possible.

3. A ways computer **WaysPC** that computes compatible ways for a set of given trains throughout the system.

4. An algorithm **Solver** that implements a greedy approach to solve the problem and that uses the WaysPC and the Oracle.

The information related to the system (the railway station) is stored in a object *System*. The object *System* is created once to give as an argument to *Solver*, which on it's turn gives it as an argument to *Oracle* and *waysPC*.
As explained previously, this solution only attempts to minimize the most important objective f_1, without taking into account f_2 and f_3.
The problem of finding matchings throughout the horizon has been simplified: each day is treated independently and sequentially.
This chapter will end by a study of the complexity of the algorithm.

4.1 Objects

4.1.1 Time

Every event that takes place in the system during the horizon occurs at a certain *time*, whether it is in the schedule of a train or providing from an imposed consumption. Every event has a time-object that stocks the time at which it occurs.

$$d_i\ hh : mm : ss$$

where i is the day, hh is the hour, mm the minutes and ss the seconds. In order to ease the computation, the object contains an integer value representing the time in seconds.

4.1.2 Duration

Some events happen during a certain amount of time, a *duration*.

$$hh : mm : ss$$

where hh are the number of hours, mm the number of minutes and ss the number of seconds the event takes. Again, the object contains an integer value representing the duration in seconds.

4.1.3 Train category

Each train has a specific *train category*. Each train category is represented by an object with the following attributes:

id the identifier of the train category,

len the length of the train,

catGroup the category group of the train,

maxDBM the value of the DBM of the train after resetting it by doing a maintenance of type D,

maintD the duration of a maintenance operation of type D,

maxTBM the value of the TBM of the train after resetting it by doing a maintenance of type T,

maintT the duration of a maintenance operation of type T.

4.1.4 Train

The train object is used to check whether or not a train can be assigned to a departure. It contains the following information:

id the identifier of the train,

inTime the time when the train enters the system. If it is an initial train, it enters at d_1 $00:00:00$ otherwise it is the time the train arrives on the platform,

remDBM the remaining DBM,

remTBM the remaining TBM,

isInit a boolean value indicating whether or not the train is an initial train or an arrival,

isDep a boolean value indicating whether or not the train is assigned to a departure,

arrival an Arrival object that is empty when the train is an initial train, otherwise it contains an arrival,

res a list of the resources the train has visited and the last resource the train is on, if it has not exited the system,

outputFile a list of Strings, containing the information that has to be printed in the train's schedule.

4.1.5 Arrival

Whenever a train enters the system through an arrival, the Train object will contain an Arrival object with the following information:

arrSeq the identifier of the arrival sequence of the arrival,

jointArr an integer indicating if the arrival is part of a joint arrival. If so, the integer is the id of the joint arrival, otherwise it is -1,

linkedDep an integer indicating if the arrival has a linked departure. If so, it is the id of the departure, otherwise it is -1,

idealDwell the ideal dwell time of the arrival,

maxDwell the maximum dwell time,

prefPlat the array containing the id's of the preferred platforms.

4.1.6 Departure

A departure object has different parameters which have to be taken into account when trying to assign a train:

id the identifier of the departure,

depTime the departure time at which the train must leave the platform,

depSeq the departure sequence which the train needs to follow to exit the system,

jointDep an integer which indicates the id of the joint departure, if the departure is part of it otherwise its value is -1,

idealDwell the ideal dwell time the departure should stay on the platform,

maxDwell the maximum dwell time the departure may stay on the platform,

reqTBM the required TBM for the journey,

reqDBM the required DBM for the journey,

prefPlat a list of identifiers of the platforms on which the departure preferably should take place,

trainId the id of the assigned train, if there is one otherwise its value is -1.

4.1.7 Resources

There are different objects representing the different types of resources, but they all inherit the following properties from their superclass *Res*:

compCatRes a list of compatible train categories,

gates a list of gates, indicating to where we can go from this resource. The gates are represented by a List containing tuples of the form

$$(Side, Int, Res, Int)$$

where Side can be A or B, the first integer indicates the gate index on the resource, while the resource Res is the neighbouring resource with the corresponding gate index. Obviously, if Side is A, then the Side of the neighbouring resource is B and reciprocally.

The different objects are:

Yard a yard has an *identifier* and an integer expressing the *capacity* of the yard.

SingleTrack a single track has an *identifier*, a specific *length* and a *capacity*.

Platform a platform has an *identifier* and a specific *length*.

FacilityD a facility D has an *identifier* and a specific *length*.

FacilityT a facility T has an *identifier* and a specific *length*.

TrackGroup a track group has an *identifier*, a *trTime* and a *hwTime*. The *trTime* is the duration for passing through the track group. The *hwTime* is the security duration between two trains which should be respected for safety reasons. If it is not respected, it is penalized by a cost in the second objective function.

4.2 Oracle

The oracle is the object which will keep track of the utilization of each resource. According to these uses, an available resource at a given time can be asked for. The variables are the uses of non track group resources. Conflicts on track groups are indeed not a problem as they are only penalized by a cost in the second objective function. Those uses are stored as a couple of integers representing the enter time and the exit time of a train on the resource. Except for yards where the order of entrance is negligible and overload is allowed, a simplification imposes that only one train can use a resource at the same time to avoid conflicts.
The oracle is divided in four main parts:

1. General purpose methods
2. Platforms assignment methods
3. Storage usage methods
4. Maintenance methods

The following sections will explain in detail each part.

4.2.1 General purpose methods

There are five general methods:

initOracle this method initializes the oracle. It will allow to re-initialize the oracle each day. Each day is indeed treated separately.
The complexity is:

$$\mathcal{O}\left(|sys.resources| - |sys.trackGroups|\right)$$

as usages of each non track group resource have to be initialized.

updateRes this function allows to provide a list of uses to add to the oracle. This method is used at the beginning of each day after the initialization step. There indeed are some remaining trains from the previous day in the system and some imposed consumptions every day. This general function is also used to update the oracle after fixing a train schedule so that its movements in the system can be taken into account when trying to establish the next train's schedule. As there is no risk of conflict between a train and itself, the overall schedule of each train can

be fixed and the oracle can be updated at the end.
The complexity is:
$$\mathcal{O}\left(|usesToAdd|\right)$$
as each new use has to be prepended to a list.

removeUse this method allows to remove some previously entered uses.
The complexity in the worst case is:
$$\mathcal{O}\left(|usesToRemove| \cdot |prevUses|\right)$$
as each element has to be removed from a list. However, this element will often be the first in the list which correspond to the last use added.

isDisp this private function takes a list of uses and a time slot and checks if there is no conflict.
The complexity is
$$\mathcal{O}\left(|prevUses|\right)$$
as all the previous uses have to be checked to avoid conflicts.

chooseBest this private function represents an heuristic which will choose the best available resource.
The considered heuristic simply consists in choosing the first compatible and available resource. This may seem naive, but in fact is quite efficient. It is the commonly used heuristic for the bin packing problem called first-fit algorithm[8].
The complexity is constant as the function will just return the first index.

4.2.2 Platforms assignment methods

Platforms can be used for arrivals, for departure or for traversal/reversal as part of a way. There are four public methods in this part of the oracle dedicated to the platforms.

updatePlat this method updates the oracle with a new use on the platform. The complexity is constant as the new use has to be prepended to the list of uses. The platform compatibility and availability is assumed to be previously checked.

removePlat this method removes a platform use from the oracle.
The worst complexity is
$$\mathcal{O}\left(|prevUses|\right)$$

as the element has to be removed from the list. However, this element will often be the first in the list (the first element indeed corresponds to the last added use) and the complexity will then be constant.

isFreePlat this method takes a platform and a time slot and returns a boolean value stating if the platform is available.
The complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as all the previous uses have to be checked to avoid conflicts.

platArrDep this method is the main platform method. It returns the best compatible platform among the available ones.
The complexity is

$$\mathcal{O}\left(|platforms| \cdot (|compCat| + |gates| + |prevUses|)\right)$$

as the compatibility, the adjacency to arrival/departures sequence, the availability and all the previous uses have to be checked for all platforms to avoid conflicts.

In the *updatePlat* method, the resource availability, the length and category of the train are assumed compatible. However, platforms have compatible train categories and limited length. Furthermore, arrivals and departures have fixed sequences which implies the group of the platforms they can use. To summarize, the compatibility, the length and the adjacency of the platform used with regards to the train specifications and the arrival/departure sequence have to be checked. Once the possible platforms are known, their availability is checked and among those the "best" one is selected according to a heuristic.

4.2.3 Storage usage methods

Single track In the chosen approach, single tracks are only used as a part of a way for train traversal or reversal. Only three methods are then needed to deal with single tracks:

updateSingleTrack this method updates the oracle with the new single track use.
The complexity is constant as the new use just has to be prepended to the list.

removeSingleTrack this method removes a single track use from the oracle.
The worst complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as the element has to be removed from a list. However, this element will often be the first in the list (the first element indeed corresponds to the last added use) and the complexity will then be constant.

isFreeSingleTrack this method checks the availability of a given single track at a given time.
The complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as all previous uses have to be checked to avoid conflicts.

Yard As the second lexicographic objective is neglected, the capacity of the yard is a negligible constraint. There are only two constraints that need to be verified:

– The compatibility of the train category,

– The time the train exits the previous resource and the time it enters the next resource after the yard define the reachable yards. Some trains will indeed be stored during a short amount of time in a yard between its arrival and departure. The time between arrival and departure can be too long to let the train stay on a platform, but too short to go to a yard and come back. In this case, yards which are compatible to go, come back and stay the minimal reversal time within the given time slot have to be verified.

Even if the second objective is not considered here, the load of the yard is computed and the compatible yard with the minimal load is assigned. There are three public methods for yards:

updateYard this method updates the oracle with a yard use.
The complexity is constant as the new use just has to be prepended to two lists.

removeYard this method removes a yard use from the oracle.
The worst complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as the element has to be removed from a list. However, this element will often be the first in the list (the first element indeed corresponds to the last added use) and the complexity will then be constant.

getFreeYard this method is the main yard method. It checks which yard is compatible, reachable in the given time slot and among those feasible yards, it returns the one with minimal load.
The complexity is

$$\mathcal{O}\left(|yards| \cdot \left(|compCat| + \mathcal{O}\left(timeWays\right) + |prevUses|\right)\right)$$

as we check the compatibility, the time and the load which implies all the previous uses.

Private methods allowing to compute the load of a given yard have also been implemented.

4.2.4 Maintenance methods

As for the other kind of resources, there are three basic access methods for the different types of maintenance facilities:

updateMaintT this method updates the oracle with a new facilityT use.
The complexity is constant as the new use just has to be prepended to the list.

removeMaintT this method removes a facilityT use from the oracle.
The worst complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as the element has to be removed from a list. However, this element will often be the first in the list (the first element indeed corresponds to the last added use) and the complexity will then be constant.

isFreeMaintT this method checks if a given facilityT is available in a given time slot.
The complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as all the previous uses have to be checked to avoid conflicts.

updateMaintD this method updates the oracle with a new facilityD use.
The complexity is constant as the new use just has to be prepended to the list.

removeMaintD this method removes a facilityD use from the oracle.
The worst complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as the element has to be removed from a list. However, this element will often be the first in the list (the first element indeed corresponds to the last added use) and the complexity will then be constant.

isFreeMaintD this method checks if a given facilityD is available in a given time slot.
The complexity is

$$\mathcal{O}\left(|prevUses|\right)$$

as all the previous uses have to be checked to avoid conflicts.

On top of these methods, there are three main methods:

maintT this method chooses the best compatible and available facilityT.
The complexity is

$$\mathcal{O}\left(|facT| \cdot \left(|compCat| + |prevUses| + \mathcal{O}\left(timeWays\right)\right)\right)$$

as for all *facilityT* the compatibility, the availability (which means checking all the previous uses to avoid conflicts) and the time needed from the previous resource and to the next resource have to be checked.

maintD this method chooses the best compatible and available facilityD.
The complexity is

$$\mathcal{O}\left(|facD| \cdot \left(|compCat| + |prevUses| + \mathcal{O}\left(timeWays\right)\right)\right)$$

as for all *facilityD* the compatibility, the availability (which means checking all the previous uses to avoid conflicts) and the time needed from the previous resource and to the next resource have to be checked.

maintTD this method will return the best compatible and available facilities and the order of the maintenances (first T or D).
The complexity is

$$\mathcal{O}\left(\left(|facT| + |facD|\right) \cdot \left(|compCat| + |prevUses| + \mathcal{O}\left(timeWays\right)\right)\right)$$

as a first order (T, D) is tested, followed by (D, T) is the first order is not feasible.

Two private heuristics (one for each type of facility) have been implemented to choose the best facility. Rather than choosing the first available facility, the first which has the smallest length and is closest to the previous resource and the next resource is chosen. It allows to reduce the travelling time of the ways and therefore reduce the number of conflicts during the ways.

4.3 Ways

As explained in the different approaches presented in chapter 3, only ways between two non track group resources are queried. Then, grouping resources that have the same neighbours is efficient and relevant.
There are two pre-processing parts:

1. Computing groups of resources.
2. Computing edges composed of track groups.

Afterwards, computing a way between two non track group resources is done in three successive parts:

1. Determining the group of the first and the last resource.
2. Finding a way between those two groups using edges composed of track groups.
3. Rewriting the computed way to output a list of compatible resources that will be successively visited.

4.3.1 Computing groups

The neighbourhood of resources is stored for each resource in a list of gates:

$$gates: \ List(Side, Int, Res, Int)$$

where the integers represent the index of each gate. A group of resources is characterized by the fact that all its resources have the same list of neighbour resources at the same side.

To facilitate the algorithm, a resource identifier is associated to each non track group resource. Those resource identifiers are integers from 0 to $n-1$, where:

$$n = nbYards + nbPlatforms + nbFacT + nbFacD + nbSingTracks.$$

This allows to store the groups as an array of integers of size n where the entries represent the identifier of the group of the corresponding resource. For example, the representation of the instance $A1$ on figure 3.5, page 38 shows that

$$\forall i \in [1 : nbPlatforms] : \ groups(platform_i) = 1$$

Two methods to link resources to their identifier are then implemented:

resToInt takes a resource as a parameter and returns the corresponding identifier.

resIntToRes takes the identifier as a parameter and returns the corresponding resource.

This approach allows to handle all the non track group resources in the same way.
The algorithm which computes the groups of non track group resources is detailed in Algorithm 1.

Algorithm 1 Computing groups

```
 1: function ComputeGroups
 2:     $res \leftarrow nonTrackGroupRes$
 3:     $n \leftarrow |res|$
 4:     $groups \leftarrow Array\,[Integer]\,(n)$
 5:     $neighbourOfGroup$
 6:     for $i \leftarrow 1 : n$ do
 7:         $neighbours : List(Side, Res) \leftarrow gates(res(i))$.distinct
 8:         $isInGroup \leftarrow false$
 9:         for $k \leftarrow 1 : |neighbourOfGroup|$ do
10:             if $neighbourOfGroup(k) = neighbours$ then
11:                 $isInGroup \leftarrow true$
12:                 $groups(i) \leftarrow k$
13:                 $break$
14:             end if
15:         end for
16:         if $!isInGroup$ then
17:             $neighbourOfGroup = neighbourOfGroup + neighbours$
18:             $groups(i) \leftarrow |neighbourOfGroup|$
19:         end if
20:     end for
21:     return $(groups, |neighbourOfGroup|)$
22: end function
```

The complexity of this algorithm is

$$\mathcal{O}\left(|nonTGRes| \cdot |gates| \cdot |groups|\right)$$

as for each non track group resource checking if the neighbours are identical among all the existing groups is needed.

4.3.2 Computing edges

In the chosen representation of the infrastructure, groups of non track group resources are the nodes and groups of track groups are the edges of the graph which models the system. The format of the information needed to compute these edges is the list of gates $gates = List(Side, Int, Res, Int)$ for each track group, where the integers represent the gate indexes.
Reversals within a track group and between two track groups are forbidden. The entering sides of two successive track groups have then to be the same. As the gates are always bi-directional, computing only the edges that leave resources by the A side allows to avoid computing all the edges twice.

The algorithm recursively iterates through each track group which is a neighbour by the B side to a group of non track group resources until we reach another group of non track group resources or a dead end (a gate which enters/exits the station). The algorithm that computes the edges is shown in algorithm 2.

Algorithm 2 Computing edges

```
1:  function computeEdges
2:      $edges \leftarrow \emptyset$
3:      for $i \leftarrow TG$ do                                    ▷ $TG$: track group
4:          for $g \leftarrow nonTGGatesB(i)$ do
5:              $edges = edges +$ recursEdges($i, EmptyList, group(g)$)
6:          end for
7:      end for
8:      return $edges$
9:  end function
10: function recursEdges($currentTG, prevTG, inGroup$)
11:     $edges \leftarrow \emptyset$
12:     for $i \leftarrow currentTGGatesSideA$ do
13:         $newPrev \leftarrow currentTG + prevTG$
14:         if $neighbourRes(i) isTG$ then
15:             $edges = edges +$ recursEdges($i, newPrev, inGroup$)
16:         else
17:             $edges = edges + (inGroup, group(i), newPrev)$
18:         end if
19:     end for
20:     return $edges$
21: end function
```

The complexity of this algorithm is

$$\mathcal{O}\left(|trackGroup|^2 \cdot |gates|\right)$$

as the algorithm has to check, for each track group which has a node on its B side, through each reachable track group which has a node on its side A. This means that the algorithm checks all neighbour groups for each track group. Afterwards, it iterates through all the reachable track groups and checks the neighbour groups every time.

4.3.3 Computing ways

Determining the minimal time needed to go from one resource to another is a common query. So it makes sense to compute all the shortest ways between each pair of groups as part of the initialization. It allows to have directly access to the time needed to reach a resource starting from another or more globally, the minimal time to reach a kind of resource (e.g. *FacilityD*) to an other one (e.g. *FacilityT*).
But in some cases, the list of successive resources used for a particular passage through the system is queried. In that case, the way has to take into account the compatibility of the resources, but also their availability. To deal with those specifications, a method which computes a way between two resources for a given train at a given time is designed.

Unfortunately, there are situations when a resource is not available to do a traversal or a reversal. The way then becomes infeasible at this time. To solve this problem, an other method which computes a way between two resources for a given train in a given timeslot is designed.
All the ways computations were computed using the well-known Dijkstra algorithm[1].

Basic shortest ways

As previously explained, during the initialization, the shortest way between each pair of groups is computed. This allows to have a lower bound on the time needed to link these two resources.
To compute those ways, the Dijkstra algorithm is used to compute the shortest way from each resource to all the other ones. This algorithm is defined in Algorithms 3 and 4 on page 59 and 60.

Algorithm 3 Dijkstra

```
function DijkstraAllWays(init, edges)
    $cur \leftarrow init$
    $toVisit \leftarrow List(cur)$
    $Visited \leftarrow \emptyset$
    $shortest \leftarrow Array[List[Edge], Side, Time](n)$
    $shortest(cur) = (List(), noSide, 0)$
    while $!toVisit$.empty do
        $adjEdges \leftarrow edges(cur, groups \notin Visited)$
        $toVisit \leftarrow toVisit - cur$
        for $k \leftarrow adjEdges$ do
            $adj \leftarrow group(k) \neq cur$
            $travRevTime \leftarrow 0$
            if $shortest(cur).side = noSide$ then
                $travRevTime \leftarrow 0$
            else if $shortest(cur).side = k.side$ then
                $travRevTime \leftarrow traversalTime$
            else if $shortest(cur).side \neq k.side$ then
                $travRevTime \leftarrow reversalTime$
            end if
            $time \leftarrow shortest(cur).time + k.time + travRevTime$
                                                    ▷ Update if shortest way
            if $shortest(adj).time > time$ then
                $shortest(adj) \leftarrow (k + shortest(curr).edges, k.side, time)$
            end if                                  ▷ add to toVisit
            $toVisit \leftarrow toVisit + adj$
        end for                        ▷ Update Visited and new current
        $Visited \leftarrow Visited + cur$
        $cur \leftarrow \min_{i \in toVisit} shortest(i).time$
    end while
    return $shortest(List, time)$
end function
```

Algorithm 4 Computing all ways

```
1: function ComputeAllWays(groups, edges)
2:     $n \leftarrow |groups|$
3:     $ways \leftarrow Array[List[Edge], Time](n, n)$
4:     for $i \leftarrow 1 : n$ do
5:         $ways(i) \leftarrow \textsc{DijkstraAllWays}(i, edges)$
6:         $(edge_m, time_m) \leftarrow \min_{j \neq i} ways(i, j)$
7:         $ways(i,i) \leftarrow (edge_m + edge_m.\text{reverse}, revTime + 2 \cdot time_m)$
8:     end for
9:     return $ways$
10: end function
```

The complexity of this algorithm is

$$\mathcal{O}\left(|groups|^2 \times |edges|\right)$$

as the shortest path from each group to every other group is computed.
This computation could be improved by using the previously computed shortest paths. But as the size of the graph is small thanks to the groups, it is not an issue and the algorithm can be withheld for further usage.

Compatible ways

When assigning a way for a given train between two non track groups resources is queried, the assigned way has to be compatible and available at the given time. Moreover, when such a way is found in terms of groups of non track group resources and edges composed of track groups, it has to be expressed in terms of resources to encode the train schedule.

The Dijkstra algorithm is used in getCompWay to compute the compatible and available ways between the two resources. Afterwards, it is expressed in terms of resources and while translating it from the grouped graph representation to a list of successive resources, the compatibility and availability of the resources is double checked.
The complexity of this algorithm is

$$\mathcal{O}\left(|groups| \cdot |edges| \cdot \mathcal{O}\left(compatible\right) \cdot \mathcal{O}\left(available\right)\right)$$

as one shortest path from one group to another group is computed, checking the compatibility and availability. The complexity to check the availability is

$$\mathcal{O}\left(available\right) = \mathcal{O}\left(|prevUses|\right)$$

while the complexity to check the compatibility is

$$\mathcal{O}\left(compatible\right) = \mathcal{O}\left(|compCat|\right)$$

Compatible ways within a time slot

Sometimes, there is no available way at a given time to link two resources. In this case, it may be interesting to compute when a way will be available. Sometimes a query is made to find a way starting as soon as possible or to find a way which arrives as late as possible on the target resource, within the timeslot.

For example, when a train arrives, it goes to a platform and it is stored into a yard after the arrival. In this case, the train is expected to leave the platform as soon as possible to release it while the yard, having an "infinite" capacity, is always free. Conversely, when a departure is covered by a train which was stored in a yard, it is expected to arrive on the platform as late as possible during a given timeslot.
To cover those two possibilities, two methods are implemented to compute the way and the minimal timeslot needed to link two resources. They are respectively called GETCOMPSLOTWAY and GETCOMSLOTR2WAY.
The complexity of this algorithm is

$$\mathcal{O}\left(|groups| \cdot |edges| \cdot |compCat| \cdot \mathcal{O}\left(available\right) \cdot |slot|\right)$$

as one shortest path from one group to another is computed, checking the compatibility, availability and timeslot needed. The complexity to check the availability is

$$\mathcal{O}\left(available\right) = \mathcal{O}\left(|prevUses|\right).$$

Bounds on complexities The parameters of the previous expressions are not always known in advance. They can be bounded by the parameters of the *System* which allows to compute the general complexity of the algorithm.

Groups The number of groups is bound by the number of non-track group resources:

$$\begin{aligned} |groups| &\leq |Yards| + |Platforms| + |FacT| + |FacD| + |SingTracks| \\ &\leq |sys.res|. \end{aligned}$$

Edges The number of edges in a graph is bound by the square of the number of node:

$$|edges| \leq |sys.res|^2.$$

CompCat The number of compatible categories per resource is bound by the total number of train categories:

$$|compcat| \leq |sys.trainCat|.$$

Previous uses The number *prevUses* of a given resource is bound by 5 times the number of trains in the system. A train which arrives can indeed go to the maintenance at most two times and can optionally go to a yard. The maximum number of visits per resource per train is then bound by 5:

$$|prevUses| \leq |sys.init| + |sys.arrivals|.$$

Slot The duration of the slot is fixed independently of the problem and its complexity can thus considered as a constant factor.

$$\begin{aligned} \mathcal{O}(getCompWay) &= \mathcal{O}(|sys.res|^3 \cdot |sys.trainCat| \cdot (|sys.init| + |sys.arrivals|)) \\ \mathcal{O}(getCompSlotWay) &= \mathcal{O}(|sys.res|^3 \cdot |sys.trainCat| \cdot (|sys.init| + |sys.arrivals|)) \end{aligned}$$

These complexities are polynomial. The size of $|res|$ is considerably reduced due to the grouping of the resources. The number of arrivals $|sys.arrivals|$ is also reduced because the problem is considered per day.

4.4 Solver

The Solver is the algorithm that will solve a given system. It does not return anything, but will modify the information that is needed for the schedule in the Train objects. At the end of the algorithm, an other function will iterate through all trains and will print the information is a csv-file. In this section, a brief summary of the algorithm will be given followed by a more detailed explanation. Algorithm 5 shows the structure of the Solver.

Algorithm 5 Solver

```
1: $oracle \leftarrow$ new Oracle(sys)                                  ▷ Initialize
2: $init \leftarrow$ sys.initialTrains
3: putInitOnYard($init$)
4: $arrs \leftarrow$ sys.arrivals
5: $deps \leftarrow$ sys.departures
6: $nbDays \leftarrow$ sys.param.nbDays
7:
8: $filter \leftarrow$ (sm1, jA, jB, sm2, sm3)                          ▷ Filters
9: $match \leftarrow$ (simpleMatch, jointMatchA,                ▷ Matching functions
10:             jointMatchB, simpleMatch, simpleMatch)
11: $mLen \leftarrow filter.length$
12:
13: for $i \leftarrow 1 : nbDays$ do
14:     initOracleDay($oracle, init, i$)
15:     $(arrDay, depDay) \leftarrow$ filterToday($arrs, deps, i$)
16:     $trainsDay \leftarrow arrDay + init$
17:
18:     for $k \leftarrow 1 : mLen$ do
19:         $(T_{ass}, D_{ass}) \leftarrow match_k(filter_k(trainsDay, depDay)))$
20:         $(trainsDay, depDay) \leftarrow$ updateSets($T_{ass}, D_{ass}$)
21:     end for
22:     $init \leftarrow$ process($trainsDay, i$)
23: end for
24: printOutAllTrains
```

A visual representation of how the algorithm works for one day can be seen on Figure 4.1, page 66. This clearly shows the sequential structure of the algorithm: sub-sets of trains and departures are computed sequentially.

A summary of how the algorithm works will now be presented:

1. Before calling the main loop, the set of initial trains at *d1 00:00:00* is processed. Some of them are on yards, some are not. These trains are being placed on a yard. This step can be seen as superfluous, however, it simplifies the algorithm because putting all the trains that are idle in the system on a yard makes it easier to retrieve them.

2. A loop is called *nbDays* times. When it is called on day i, the set of initial trains contains only the trains that are in the system at $d_i\ 00:00:00$. The trains that arrive during a given day and that are not assigned at the end of the day, become the initial trains for the next day. Trains still in the system at the end of the last day ($i = nbDays$) will be processed to exit the system.

3. The initial trains and the trains arriving on day i, noted *TrainsDay*, are those that the algorithm will consider to assign to the departures on day i, noted *DepDay*. The matching will only consider trains, resource allocations and departures on the given day. It reduces considerably the space complexity.

4. All the trains are filtered to obtain those that are not part of a joint arrival. All the departures are filtered to obtain those that do not belong to a joint departure. They are called *simple* trains and *simple* departures. The matching procedure SimpleMatch will use these two sets to find a matching between them. All the trains and departures that have been matched are removed from *TrainsDay* and *DepDay* respectively. They are indeed no longer possible candidates for a future matching.

5. The second matching procedure Joint Match A will consider only *joint arrivals* and *joint departures* that do not have to be joint nor disjoint. This procedure is very similar to SimpleMatch because the trains stay together. However, they differ in the fact that the resources have to be handled differently because some of them are not compatible with the number of the joint trains or their length. Again, the matchings are removed afterwards.

6. The third matching procedure Joint Match B will consider all *simple trains* for all *joint departures*. Therefore, the filtering of the trains will apply a disjunction disjunJoints to all joint arrivals that have not been assigned in the previous matching. This disjunction procedure will first assign a platform for the joint arrival to arrive on and it will then send it to a yard where it will be disjoint. After this operation, there are only simple trains left in the yards. This set is used to find a matching. Afterwards, the remaining set is again updated.

7. The fourth iteration re-uses SimpleMatch with as input all remaining trains. The filtering of the departures will return all simple departures, and all first trains of a non-assigned joint departure. If a joint-departure is not entirely assigned, assigning at least one train will indeed decrease the objective f_1. Again, the remaining set is updated.

8. The last iteration will do almost the same job as the previous one, but with one difference: if the first train of a joint-departure hasn't been assigned in the previous matching, this matching will consider the last train of the joint departure and all remaining single trains.

9. At the end of the day, all non-assigned departures will increase the objective f_1. The non-assigned trains are stored in a yard, possibly after an arrival.

10. After an iteration of *nbDays* times of the main loop, all the schedules of each train are printed out in a file.

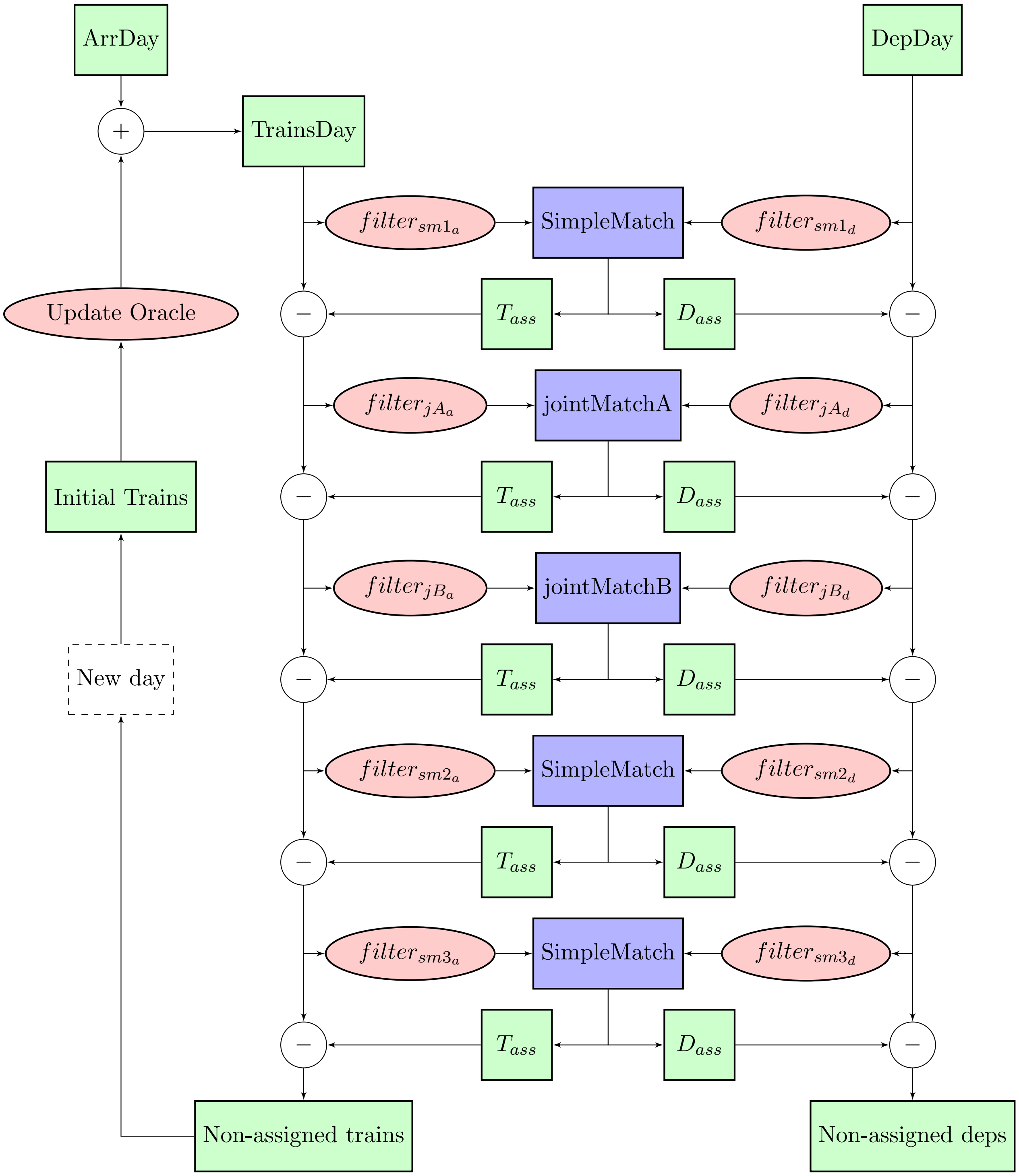

Figure 4.1: A visual representation of how *Solver* works.

4.4.1 Pre-processing

The pre-processing consists of several steps:

Creating Solver The configuration of the railway station is given as an argument *System* when the Solver is called. Some parameters are stored locally in the *Solver* in order to have an easy access to them.

Creating ways and oracle A new ways computer *groupedWays* is created along with a new *Oracle*.

Treating initial trains The initial trains on day 1 follow a special procedure PUTINITONYARD. Given that the hypothesis was made that all trains are stored in a Yard and that some initial trains are on other resources, this procedure will put the initial trains that are not on a Yard, on it. This functioning can be seen in algorithm 6.

Algorithm 6 putInitOnYard

```
function PUTINITONYARD(initialTrains)
    $stockInfo$
    $prevTime \leftarrow d_1\ 00:00:00$
    for $t \leftarrow initialTrains$ do
        enterSystem($t$)
        if $t.res != Yard$ then
            $y \leftarrow$ COMPATIBLEYARD
            $(minOut, maxOut) \leftarrow$ computeInterval($t, prevTime, margin$)
            $info \leftarrow t$.addCompatibleWay($t, t.res, y, minOut, maxOut$)
            updateInfo($t, y, info$)
            $stockInfo \leftarrow (y, info.inYardTime)$
            $prevTime \leftarrow info.outResTime$
        end if
    end for
    return $stockInfo$
end function
```

For every initial train that is not initially on a yard, the following operations occur:

1. A compatible yard is computed to put the train in.
2. An interval is computed during which the train must leave its initial resource. This interval is computed using the time that the previous

initial train had to leave a non-yard resource. These intervals do not overlap. The conflicts can then be minimized on other resources that the train will have to use to go from its initial resource to a yard. This is possible because at that given time of the day, there is not a lot of traffic in the system.

3. A compatible way is computed for the train to leave its initial resource during the given interval and go to the yard.
4. The information is updated in the train object: when it leaves his actual resource, which way it uses to go to the yard and when it enters the yard.
5. The information about the usage of the yards is stored in a variable *stockInfo*. This list will be used when initializing the oracle, so that it knows the usage of its resources.
6. The *prevTime* is updated with the time the trains leaves its initial resource.

4.4.2 Daily initializations

The main loop in SOLVER is repeated *nbDays* times. Every day the following steps occur:

MaintDay the number of maintenances already completed is re-initialized to 0. This number is updated after every matching and is used to know whether a maintenance can be performed or if the daily limit *maxMaint* has been reached.

Init oracle the oracle is initialized, e.g. the memory is erased. Every day is indeed treated independently. After resetting the oracle, it is updated with the information about all initial trains.

Filtering working set the set of unassigned arrivals and departures is computed through *fDay*. This function will take all arrivals and departures throughout the horizon and will check if their arrival or departure time takes place during the given day.

4.4.3 Matching: simple trains and simple departures

Before trying to match simple trains and simple departures, a filter *fSM1* will filter the unassigned trains and departures so that only the simple ones remain.

Algorithm 7 SimpleMatch

```
function SIMPLEMATCH($sTrains, sDeps$)
    $stockTrains \leftarrow \emptyset$
    $stockDeps \leftarrow \emptyset$
    for $dep \leftarrow sDeps$ do
        $match \leftarrow \emptyset$                                    ▷ Minimum cost match
        $compatT \leftarrow$ compatTrains($sTrains \setminus stockTrains, dep$)
        ($noMaints, maints$) $\leftarrow$ split($compatT$)
        if !$noMaints.empty$ then
            $costs \leftarrow$ computeCost($noMaints$)
            if !$costs.empty$ then
                $match \leftarrow costs.min$
            end if
        else if !$maints.empty$ && $match.empty$ then
            $costs \leftarrow$ computeCost($maints$)
            if !$costs.empty$ then
                $match \leftarrow costs.min$
            end if
        end if

        if !$match.empty$ then                                 ▷ Post-processing
            ($canDo, info$) $\leftarrow$ postProcess($match$)
            if $canDo$ then
                updateInfo($match, info$)
                $stockTrains$.add($match.t$)
                $stockDeps$.add($match.dep$)
            end if
        end if
    end for
    return ($stockTrains, stockDeps$)
end function
```

The SimpleMatch procedure is described in algorithm 7. It takes all non-assigned trains of the day, and all non-assigned simple departures of the day. It then tries to assign a train to each departure and returns the set of assigned trains and departures. For every non-assigned departure, the following steps are taken:

1. It computes the set of compatible trains that could be assigned to the departure:

Time compatibility the train has to be in the system at least *revTime* before the departure time and if the arrival needs to be stored in a yard between its entrance time and departure time, there must be enough time to go to a yard and back, with or without undergoing maintenance in between.

Category compatibility the category of the train has to be compatible with the departure.

Required TBM and DBM the train needs, possibly after a maintenance, enough DBM and TBM for the departure.

$$remDBM \geq reqDBM$$

$$remTBM \geq reqTBM.$$

Feasible maintenance if maintenance is needed, it must be possible:

$$maintToday \leq maxMaint$$

and there must be enough time *maintTime* to attend it and to go back and forth between the facility and other resources.

2. The set of compatible trains is split in a set that does not needs maintenance *noMaints* and that does, *maints*.

3. If *noMaints* is not empty, the costs for every match with a train in this set is computed. If there exists a minimum cost match, this *match* is used. If *noMaints* does not have a feasible match or if the set is empty and if *maints* is not empty, all costs are computed for *maints* and a minimum cost *match* is returned, if it exists.

   The reason for this split is that historically, the COMPUTECOST procedure was very slow, hence, in order to speed up the algorithm, less computations needed to be made. Afterwards, the algorithm was made faster and this distinction was not really needed anymore. However, the results were worse and the decision was made to keep it that way.

4. If there exists a minimum cost *match*, this match is post-processed. The post-processing consists in verifying whether a train can cover all the ways it needs to take, whether it can undergo maintenance if needed and whether there is an available platform at the time of arrival, if it is an arrival and at the time of the departure.

5. The post-processing can return *false* if it does not succeed in assigning a train. In that case, the information is not updated and the algorithm does not return a match for the particular departure.

6. If the post-processing is possible, the following information is updated:

   – The number of maintenances of the day are updated.
   – The path of the train throughout the system is added to its schedule.
   – The arrival and departure objects are flagged to indicate they are a match.
   – The *Oracle* is updated with the information about all the resources used during a given timeslot by the train.
   – If the assigned departure is linked to a future arrival, the information of the corresponding train is updated.

Post-processing

The reasoning behind the post-processing is visualized on figure 4.2 on page 72. The post-processing distinguishes if a train is already in the system or not, if it needs maintenance and if it needs to be stored between its arrival and departure.

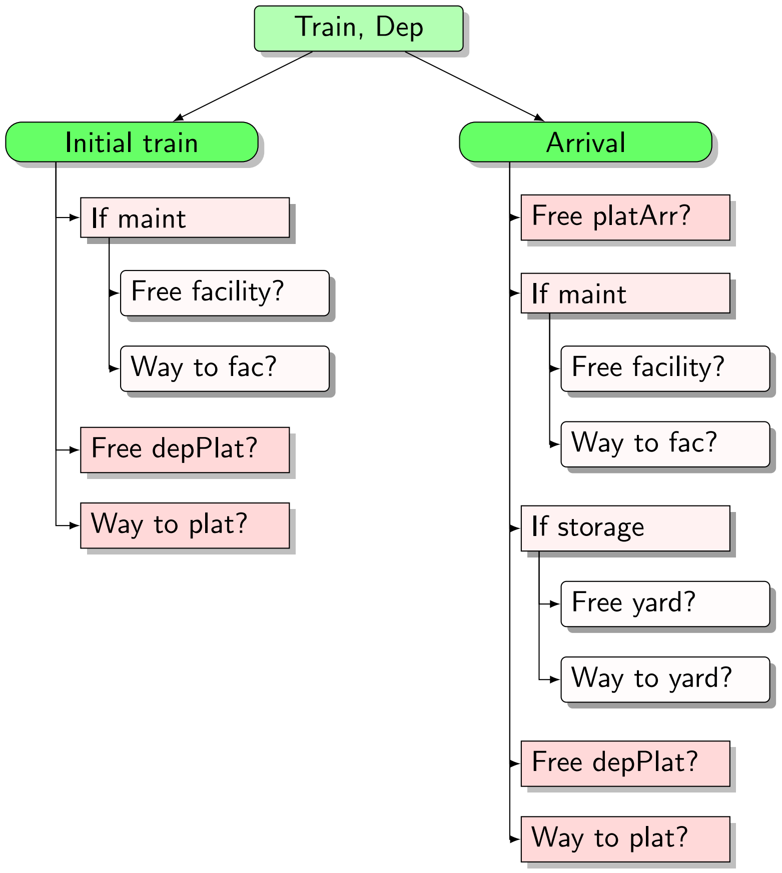

Figure 4.2: Visual representation of how a matched train is post-processed.

Initial trains are already in the system and are post-processed differently than arrivals. Several situations are considered:

No maintenance If the train t is assigned to a departure d and does not need maintenance, the post-processing will compute a timeslot during which the train has to arrive on the platform for departure.

This interval has an upper bound:

$$dIn_{max} = depTime - revTime.$$

On all the given configurations platforms are indeed stacks and the minimum duration a train can stay on it is *revtime*. The lower bound dIn_{min} is bounded by the duration *dep.maxDwell*. In general, a shorter interval is used, e.g. $2 \cdot revTime$, in order to ensure enough available platforms.

If such a platform exists and if the train can arrive on it during the interval, the post-processing will output *true*, together with the information about the used resources and the way the train takes to go from the yard to the platform.

Maintenance If the train needs maintenance, it undergoes it right before its departure. The same procedure is followed for the departure platform, but some supplementary steps are taken:

1. In order to obtain a lower bound on the time needed to reach the platform from the facility, a compatible maintenance facility is computed, just by traversing the list of facilities. This facility will just be used to estimate the time from a compatible maintenance facility to the departure platform:

   $$time_{fac,plat} = waysPC.getTimeWay(fac, platDep).$$

2. An interval is computed during which the train should leave the maintenance facility to go to the platform:

   $$minOutFac = dIn_{max} - time_{fac,plat} - margin$$

   $$maxOutFac = dIn_{max} - time_{fac,plat}$$

   Once this interval is fixed, a facility is queried from the oracle during the interval

   $$(minInFac, minOutFac)$$

   where $minInFac = minOutFac - time_{maintenance}$.

3. If such a facility exists, a compatible way is computed that has to leave the facility during

   $$(minOutFac, maxOutFac)$$

   and has to arrive on the platform during

   $$(dIn_{min}, dIn_{max})$$

   using GETCOMPSLOTR2WAY.

4. If a compatible way exists, the time *outFac* when the train leaves the facility is fixed. This implies that the last time at which the train has to be in the facility is:

   $$maxInFac = outFac - time_{maintenance}$$

   where the maintenance duration depends on the type (T or D) and of the category of the train.

5. The last step consists in computing when and how the train should leave the yard to go to the maintenance facility. Again, getCompSlotR2Way is used and the train can arrive on the facility during the interval
$$(minInFac, maxInFac).$$
If this final step is possible, the train can then be assigned and the post-processing will return the boolean value *true* and update the corresponding information.

Arrivals are trains that enter the system during the day via a platform. They are handled differently than initial trains. Several situations are possible:

Immediate departure An arrival can enter on a given platform and immediately leave after a small amount time via the same platform. This is only possible if:
$$depTime_d - inTime_t \leq dwell$$
where the dwell time is a fixed duration limited by the maximum dwell time. In that case, a platform is requested during the timeslot:
$$(inTime_t, depTime_d).$$
If a platform is available, then the matching is possible.

Storage An arrival can need storage, if it can not stay long enough on the platform before departure. First of all, a platform is queried for the arrival of the train during the timeslot:
$$(inTime_t, aOut_{max})$$
where $aOut_{max} = inTime_t + 2 \cdot revTime$.

Afterwards, a free yard is queried so that a compatible way can be computed from the arrival platform to the yard. By using getCompSlotWay, the train can leave the platform between
$$(aOut_{min}, aOut_{max})$$
where $aOut_{min} = inTime_t + revTime$. That fixes the time the train will arrive in the yard, noted $inYard$. This implies that the train may leave the yard at the earliest at
$$minOutYard = inYard + revTime.$$

Afterwards, getCompSlotR2Way is used to compute a compatible way for the train which may leave the yard at earliest at $minOutYard$ and must arrive at the departure platform between

$$(dIn_{min}, dIn_{max}).$$

If such a way exists, the match is possible.

Maintenance An arrival can also need a maintenance of type T or D. A maintenance is always carried out immediately after the arrival, so before any storage if there is one. First of all, a platform is queried for the train to arrive on during the interval:

$$(aOut_{min}, aOut_{max}).$$

Afterwards, a compatible maintenance facility is computed to estimate the travel time. This facility is not queried from the *Oracle*, as it is just needed to obtain an approximation of the travel time.

$$time_{plat,fac} = waysPC.getTimeWay(platArr, fac).$$

This gives us an interval during which a compatible maintenance facility can be queried from the *Oracle*:

$$(minInFac, maxOutFac)$$

where

$$\begin{aligned} minInFac &= aOut_{min} + time_{plat,fac} \\ maxOutFac &= minInFac + time_{maintenance} + margin. \end{aligned}$$

Afterwards, getCompSlotWay is used to compute a compatible way for the train to leave the platform during the timeslot $(aOut_{min}, aOut_{max})$ and to go to the maintenance facility.

After the maintenance, a train can immediately go to the departure platform during (dIn_{min}, dIn_{max}) or it can go to a yard for storage. If the train needs storage, it must leave the facility after the maintenance during the timeslot $(minOutFac, maxOutFac)$ to go to the yard. $minOutFac$ is the time the maintenance has finished and $maxOutFac$ is that time plus a fixed margin.

Cost function

The cost function for the SIMPLEMATCH procedure takes into account two different costs and is described in the algorithm 8:

reuseCost the cost of not reusing a recommended pair,

maintCost the cost of undergoing maintenance. The cost of a maintenance for train t of type T is $remTBM_t \cdot tCost$ and for a maintenance of type D is $remDBM \cdot dCost$. In that way, maintenance on trains that still have a lot of TBM or DBM left is more costly and thus discouraged.

Furthermore, the cost function checks the feasibility of a pair:

- If needed, is there a compatible facility for maintenance and can it go there from his current resource?
- If needed, is there a compatible yard for storage and can it go there from his current resource?
- Can it go from its current resource to a platform for departure?

If at least one of these points is not feasible, the cost is evaluated at its maximum value.
The assumption was made that an arrival that needs maintenance, always goes to a facility immediately after its arrival, before going to a yard for storage or to the platform for departure.
Sometimes, a departure immediately occurs after an arrival. In that case, the cost function will only check if there is a compatible platform for the pair.

Algorithm 8 Cost Simple Match

```
function COMPUTECOST($t, dep, maint$)
    $cost$
    if $(t, dep) \notin reuses$ then
        $cost \leftarrow cost + reuseCost$
    end if
    if $!maint$ then
        if $t.isInit$ then
            $canDo \leftarrow$ checkCompatibleWays(t,dep)
        else                                                  ▷ t.isArrival
            $storage \leftarrow$ ifStorageNeeded($t, dep$)
            $canDo \leftarrow$ checkCompatibleWays(t,dep, storage)
        end if
    else
        $cost \leftarrow cost + maintCost$
        $fac \leftarrow$ getCompatFac($t, dep$)
        if $t.isInit$ then
            $canDo \leftarrow$ checkCompatibleWays(t,fac, dep)
        else                                                  ▷ t.isArrival
            $storage \leftarrow$ ifStorageNeeded($t, fac, dep$)
            $canDo \leftarrow$ checkCompatibleWays(t,dep, fac,storage)
        end if
    end if
    if $!canDo$ then
        return $noSol$
    end if
    return $(cost, storage, maint)$
end function
```

4.4.4 Matching: joint trains and joint departures

The trains needed for JOINTMATCHA are filtered by *fJA*. This filter returns only the joint arrivals and joint departures.
The functioning of the algorithm is exactly the same as with the simple match, except for the fact that $|trains| > 1$. However, given that the trains stay joint, the only thing that changes are the queries to the *Oracle* and the *WaysPC*: they must take into account the total length of the joint trains and of their respective categories to ensure compatibility and feasibility.

The computation of feasible trains is the same as for the simple match, with the slight difference that every train of a joint arrival must be compatible with one another.

Algorithm 9 Joint Match A

```
function JOINTMATCHA($jTrains, jDeps$)
    $stockTrains$
    $stockDeps$
    $n \leftarrow jdep.length$
    for $jdep \leftarrow jDeps$ do
        for $i \leftarrow 0 : n$ do
            $compatT(i) \leftarrow$ compatTrains($jTrains \setminus stockTrains, jdep(i)$)
        end for
        $(noMaints, maints) \leftarrow$ split($compatT$)
        if !$noMaints.empty$ then
            $costs \leftarrow$ computeCost($noMaints$)
            if !$costs.empty$ then
                $match \leftarrow costs.min$
            end if
        else if !$maints.empty$ && $match.empty$ then
            $costs \leftarrow$ computeCost($maints$)
            if !$costs.empty$ then
                $match \leftarrow costs.min$
            end if
        end if

        if !$match.empty$ then
            $(canDo, info) \leftarrow postProcess(match)$
            if $canDo$ then
                updateInfo($match, info$)
                $stockTrains$.add($match.trains$)
                $stockDeps$.add($match.deps$)
            end if
        end if
    end for
    return $(stockTrains, stockDeps)$
end function
```

The algorithm 9 operates in the same way as the previous one. However, if one train of the convoy undergoes maintenance, the other ones do too because they are joint and undergo exactly the same operations.
The updates done afterwards are the same as for a simple match, but they

are done for each train. Each train of a joint match has therefore exactly the same schedule.

Cost function

This cost function 10 is very similar to the cost function of the simple match. The only differences are:

- The pair has to get a compatibility check: their arrival respectively departure sequences have to be the same,
- JOINT MATCH A only consider arrivals, no initial trains,
- If one train of the convoy needs maintenance, they all undergo maintenance and the cost is:

$$\sum_{t \in trains} remDBM \cdot dCost \text{ or } \sum_{t \in trains} remTBM \cdot tCost.$$

Algorithm 10 Joint Match A

```
function computeJointCost($jTrain, jDep, maint, storage$)
    $cost$
    $n \leftarrow jDep.length$
    for $i \leftarrow 1 : n$ do
        if $(jTrain(i), jDep(i)) \notin reuses$ then
            $cost \leftarrow cost + reuseCost$
        end if
    end for
    if compatSeq($jTrain, jDep$) then
        return $noSol$
    end if
    if !$maint$ then
        if $storage$ then
            $canDo \leftarrow$ checkCompatWays(jTrain,storage,jDep)
        end if
    else
        $cost \leftarrow cost + maintCost$
        $fac \leftarrow$ getCompatFac($t, dep$)
        if $storage$ then
            $canDo \leftarrow$ checkCompatWays(jTrain, fac, storage, jDep)
        else
            $canDo \leftarrow$ checkCompatWays(jTrain, fac, jDep)
        end if
    end if
    if !$canDo$ then
        return $noSol$
    end if
    return $cost$
end function
```

4.4.5 Matching: simple trains and joint departures

After Joint Match A, there are no more joint arrivals that are potential candidates for a joint departure. To fully utilize all trains, all remaining joint arrivals are being disassembled. Afterwards, a matching is performed with all simple trains and the remaining unassigned joint departures by assembling simple trains before departure on the departure platform. The filtering *fJB* will perform this disjunction task. It then returns all unassigned trains and unassigned joint departures.

Algorithm 11 Joint Match B

```
function JOINTMATCHB($sTrains, jDeps$)
    $stockTrains$
    $stockDeps$
    $n \leftarrow jdep.length$
    for $jdep \leftarrow jDeps$ do
        for $i \leftarrow 0$ until $n$ do
            $compatT(i) \leftarrow$ compatTrains($sTrains \setminus stockTrains, jdep(i)$)
        end for
    end for
    $compatT \leftarrow$ compatTuples($compatT$)
    $costs \leftarrow$ computeCost($compatT$)

    if !$costs.empty$ then
        $match \leftarrow costs.min$
        $(canDo, info) \leftarrow$ postProcess($match$)
        if $canDo$ then
            for $j \leftarrow 0 : n$ do
                updateMatch($match(j), jdep(j), info(j)$)
                $stockTrains$.add($match$)
                $stockDeps$.add($dep$)
            end for
        end if
    end if
    return ($stockTrains, stockDeps$)
end function
```

The JOINTMATCHB algorithm 11 follows the same reasoning as JOINTMATCHA, but is different in the sense that separate trains which arrive all separately on the platform are considered, before being joint for departure.

Compatible trains The computation of the compatible trains is done under the hypothesis that only trains that do not need maintenance are considered. Per departure of a joint departure, a set of feasible trains that respects the following conditions is computed:

$$\begin{aligned} inTime_t + junTime &\leq depTime_d - minAsbTime \\ cat_t &\in compCatDep_d \\ remDBM_t &\geq reqDBM_d \\ remTBM_t &\geq reqTBM_d \end{aligned}$$

Given the constraint that a joint departure must be joint at least *minAsbTime* before leaving, a train must have enough time to undergo one or more junctions. If the train is an arrival, it must arrive immediately on the platform and cannot stay too long on it either.

Every departure of the joint departure will have a list of compatible trains, but possibly with repetition, e.g. the same train is compatible for more than one departure. The function *compatTuples* will take all these lists and return only distinct and feasible tuples without repetition, such that each element in a tuple has the same category group.

Post-processing

The post-processing of the Joint Match B procedure is similar to the post-processing of Joint Match A, but the time when an initial train arrives on a platform has to be fixed.
That time decision is influenced by whether the left and right neighbours of the initial train are arrivals or not. Arrivals fix a lower or upper bound for the time the train can arrive on the platform. Otherwise, a fixed margin is used. These times are stores in a variable called *atPlatTimes*.

Once the *atPlatTimes* are computed, a platform must be queried for all the trains to arrive on before departure during the following timeslot:

$$(atPlatTimes(0) - margin_0, depTime_d).$$

An arrival will immediately arrive on this platform to wait to be joined, while an initial train must arrive on the platform during a given interval:

$$(minAtPlat(i), atPlatTimes(i))$$

where i is the position (from left to right) of the train in the joint departure and

$$\begin{aligned} minAtPlat(i) &= \quad atPlatTimes(0) - margin_0 \quad \text{if } (i = 0) \\ &= atPlatTimes(i-1) + margin_1 \quad \text{if } (i > 0) \end{aligned}$$

Once this intervals are fixed for the initial trains, getCompSlotR2Way is used to compute their exact arrival time on the platform. Whenever at least 2 trains are on the platform, they can begin their junction immediately.

If a compatible way exists for every initial train to arrive on the platform, the post-processing will return *true* and all trains are updated with the necessary information, including the information about their junction.

Disjunction of joint arrivals

For every unassigned joint arrival, the following steps are made, as described in the algorithm 12:

Arrival a compatible platform is queried for the trains to arrive on,

Two trains if there are two trains in the joint arrival, a compatible yard is queried, and all the trains are sent to the yard, still joint. Once on the yard, the disjunction takes place. Afterwards, one of the two trains is moved, otherwise it will block the other train. Therefore, a function will return the train that is closest to the entrance/exit side of the yard, which is a stack. That train will then leave the yard through the last gate, make a u-turn and come back through the first gate of the same yard.

More than two trains if there are more than two trains in the joint arrival, this creates errors on some resources which do not have enough capacity. In that case, the right most train is disjoint on the arrival platform and send to a yard. Afterwards, the same procedure is applied to the remaining joint trains.

Algorithm 12 Disjunction joint arrivals

```
function DISJUNJOINTS($jArrs$)
    $lastTime \leftarrow d_{nbDays}$ 23 : 59 : 59
    for $jarr \leftarrow jArrs$ do
        $n \leftarrow jarr.length$
        addArrival($jarr$)

        if $n = 2$ then
            $y \leftarrow compatYard(jarr)$
            $info \leftarrow$ getCompatWay($jarr, y$)
            for $i \leftarrow\ 0 : n$ do
                updateInfo($jarr(i), y, info(i)$)
                disJunction($i, jarr, y$)
            end for
        else if $n > 2$ then
            $y \leftarrow$ compatYard($jarr$)
            disJunction($n, jarr, platArr$)
            $info_1 \leftarrow$ getCompatWay($jarr.last, y$)
            updateInfo($jarr.last, y, info_1$)

            $jarr \leftarrow jarr(1 : n - 1)$
            $info_2 \leftarrow getCompatibleWay(jarr, y)$
            for $i \leftarrow\ 0 : n - 1$ do
                updateInfo($jarr(i), y, info_2(i)$)
                disJunction($i, jarr, y$)
            end for
        end if
        $t_m \leftarrow jarr$.getTrainAtOpenSide            ▷ Blocks other train, move.
        $info_m \leftarrow$ getCompatWay($t_m, y, changeGate$)
        updateInfo($t_m, y_m, info_m$)
    end for
end function
```

Cost function

For the cost function 13 of the JOINT MATCH B procedure, only the reuse cost is taken into account, because the simplifying assumption was made that only trains with enough remaining DBM and TBM are considered. The following feasibility checks have to be done:

Compatible arrival time if the tuple consists of arrivals, the arrival times have to be in increasing order.

Sufficient time if there are initial trains, they have to be able to arrive at the right moment on the platform and still have sufficient time to be joint, e.g. the trains must be joint at least $minAsbTime$ before the departure.

Maximum dwell if the train is an arrival, it must arrive during a certain interval, because it cannot stay longer than $maxDwell_{dep}$. But it must still have enough time to be joint.

Compatible arrival sequence all arrivals in the tuple have to follow the same arrival sequence.

Algorithm 13 Joint Match B

```
function COMPUTEJOINTCOST($jDep, trains$)
    $cost$
    $n \leftarrow jDep.length$
    $nJun \leftarrow n - 1$
    $minDepTime \leftarrow depTime - minAsbTime$
    if $trains.head.isArr$ then
        $atPlat \leftarrow inTime$
    else
        $aAtPlat \leftarrow minDepTime - totJunTime - margin$
    end if
    $canPlat \leftarrow \text{compatPlat}(jDep, trains, atPlat, depTime)$
    if $!canPlat$ then
        return $noSol$
    end if

    for $i \leftarrow 0 : n$ do
        if $(trains(i), jDep(i)) \notin reuses$ then
            $cost \leftarrow cost + reuseCost$
        end if

        $t \leftarrow trains(i)$
        $atPlat_{min} \leftarrow depTime_d - maxDwell_d$
        $atPlat_{max} \leftarrow minDepTime - junTime$
        if $t.isArr$ then
            $canDo(i) \leftarrow t.inTime.\text{in}(atPlat_{min}, atPlat_{max})$
        else
            $toPlat \leftarrow waysPC.getTimeWay(initRes_t, compatPlat)$
            $canDo(i) \leftarrow (atPlat_{max} \geq initTime_t + toPlat + margin)$
        end if
    end for
    if $!canDo$ then
        return $noSol$
    end if
    return $cost$
end function
```

4.4.6 Matching: simple trains and remaining departures

The last step consists in re-using the SIMPLE MATCH functions twice to assign the remaining simple departures. Because assigning one train of a joint departure also decreases the first objective function, an attempt is made to match them:

1. The first filtering *fSM2* takes all remaining simple trains and considers the first (leftmost) train of a joint departure as a simple departure.

2. The second filter *fSM3* takes all remaining simple trains and considers the last (rightmost) train of the joint departures of which the first departure has not been assigned.

4.5 Improvement of Solver

After having produced a functional algorithm, it became clear that this solution could be improved without changing drastically the algorithm. In SOLVER, some trains that are not suited for a next matching step are indeed filtered out, because they are linked to a departure that had already been assigned. For example, suppose that during the first SIMPLE MATCH, the following departure is assigned:

Departure	depTime	reqDBM	reqTBM
Dep21	d1 21:45:00	500	04:00:00

Arrival	Train	arrTime	remDBM	remTBM
Arr18	Train18	d1 21:25:00	1500	15:00:00

The third procedure is the JOINT MATCH B where simple trains are used to match with joint departures. Let's have a look at the following joint departure:

Departure	depTime	jointDep	reqDBM	reqTBM
Dep14	d1 12:20:00	jointDep8	1100	11:00:00
Dep15	d1 12:20:00	jointDep8	1100	11:00:00

Suppose that a simple train that has minimum cost can be assigned to $Dep14$ with the following properties:

Arrival	Train	arrTime	linkedDep	remDBM	remTBM
Arr7	Train7	d1 11:50:00	linkedDep21	1500	15:00:00

This is a feasible match. However, the properties of $Train18$ will be altered if this match is assigned because this train will now inherit the properties of $Train7$, due to the linked departure:

Arrival	Train	arrTime	remDBM	remTBM
Arr18	Train18	d1 21:25:00	100	04:00:00

However, when assigning $Train18$ to $Dep21$ first, this properties were not altered and the match was perfectly possible. But due to assigning $Train7$ this match becomes impossible.

So in order to prevent this type of situations to happen, the trains that have a linked departure that is already assigned are filtered out. This illustrates the fact that the order in which the matching is carried out is important.

By implementing a new solver Sequential Solver, the order of the matching has been changed as can be seen in the pseudo-code on Figure 14 on page 90. This solver will iterate through the departures, check whether if it is a simple or a joint departure and try a matching in function of its length. If a joint match does not produce a matching with a joint arrival, a matching is tried with all the simple trains in the system. However, no disjunction is made at this stage because this would imply a major modification of the algorithm. After iterating through simple and joint departures, the remaining trains are matched following the usual procedure.

Algorithm 14 Sequential Solver

```
1:  $oracle \leftarrow$ new Oracle(sys)                                   ▷ Initialize
2:  $init \leftarrow$ sys.initialTrains
3:  putInitOnYard($init$)
4:  $arrs \leftarrow$ sys.arrivals
5:  $deps \leftarrow$ sys.departures
6:  $nbDays \leftarrow$ sys.param.nbDays
7:  **for** $i \leftarrow 1 : nbDays$ **do**
8:      initOracleDay($oracle, init, i$)
9:      $(arrDay, depDay) \leftarrow$ filterToday($arrs, deps, i$)
10:     $trainsDay \leftarrow arrDay + init$
11:     $dDay \leftarrow$ copy($depDay$)
12:     $dLen \leftarrow dDay.length$
13:     $ctr \leftarrow 0$
14:     **while** $ctr < dLen$ **do**
15:         $d \leftarrow$ getDep($dDay, ctr$)
16:         **if** $d.simple$ **then**
17:             $sTrains \leftarrow$ fSM1($trainsDay$)
18:             $T_{ass} \leftarrow$ simpleMatch($sTrains, d$)
19:         **else**
20:             $jArrsA \leftarrow$ fJA($trainsDay$)
21:             $T_{ass} \leftarrow$ jointMatchA($jArrsA, d$)
22:             **if** $T_{ass}.empty$ **then**
23:                 $sTrainsA \leftarrow$ fSM1($trainsDay$)
24:                 $T_{ass} \leftarrow$ jointMatchB($sTrainsA, d$)
25:             **end if**
26:         **end if**
27:         $(trainsDay, depDay) \leftarrow$ updateSets($T_{ass}, d$)
28:         $ctr \leftarrow ctr + d.length$
29:     **end while**
30:     $match \leftarrow (jointMatchB, simpleMatch, simpleMatch)$
31:     $filter \leftarrow (fJB, fSM2, fSM3)$
32:     $mLen \leftarrow filter.length$
33:     **for** $k\ gets 1 : mLen$ **do**
34:         $(T_{ass}, D_{ass}) \leftarrow match_k(filter_k(trainsDay, depDay))$
35:         $(trainsDay, depDay) \leftarrow$ updateSets($T_{ass}, D_{ass}$)
36:     **end for**
37:
38:     $init \leftarrow$ process($trainsDay, i$)
39: **end for**
40: printOutAllTrains
```

4.6 Complexity of the algorithm

Studying the complexity of the whole algorithm comes down to studying the main loop of the SOLVER procedure. All the other pre and post computations are linear in function of the input files. They just parse every line of text and process it to several objects. Furthermore, creating a new object **WaysPC** takes a negligible amount of time, as does creating a new object *Oracle*.

$$\mathcal{O}(\textsc{Solver}) = \mathcal{O}(\textsc{Sequential Solver}) = |deps_{day}| \cdot |allTrains_{day}|^2 \cdot |res|^3$$

where $|d|$ is the total number of departures per day, $|allTrains|$ is the number of arrivals and initial trains per day and $|res|$ is the total number of resources in the system. This expression is obtained using the individual complexities of the functions. These complexities can be found in appendix B on page 110. It does not take into account the number of days, because it is bound by:

$$0 < nbDays \leq 14.$$

The other parameters are bound too:

$$\begin{array}{rcl} 50 \leq & |deps_{day}| & \leq 500 \\ 50 \leq & |arr_{day}| & \leq 500 \\ 10 \leq & |init| & \leq 100 \\ 21 \leq & |res| & \leq 175 \end{array}$$

The $|allTrains_{day}|$ is composed of the $|arr_{day}|$ and the remaining trains of the previous day and the initial trains. The size of this set depends on the number of previously assigned departures.

Chapter 5

Validation

This part will introduce the instances we worked on, followed by the results of SOLVER and SEQUENTIAL SOLVER. Afterwards, the results will be discussed and benchmarked to results obtained by other teams that worked on the same problem.

5.1 Instances

Only 6 of the 12 instances provided are considered. They are all quite similar.

Configurations The configurations of the railway stations are all the same, except for instance A4. The graphical representations are available in the appendix A. The difference with instance A4 is that instead of having a large yard, a part of the yard is replaced by several single tracks.
In the given instances, all platforms and yards are stacks. All the other resources, such as facilities and single tracks can be a double-ended queue or a stack.

Data The tableau 5.1 shows the most important properties of each instance. All the instances have a horizon of 7 days.

Instance	Departures	Initial Trains	Arrivals	maxMaint
1	1235	37	1235	30
2	1235	37	1235	30
3	1235	37	1235	60
4	1235	37	1235	30
5	1499	59	1499	40
6	1235	37	1235	30

Figure 5.1: Properties of the considered instances.

5.2 Results

The following results have been obtained on a computer with an Intel Core i7 3537U, with a clock speed of 2 GHz, 2 cores and 8 GB RAM with 4 GB assigned to Eclipse.
A checker[5] is provided by the organization of the challenge and is used to check the feasibility and the objective values of the obtained solutions.

i	Solver			Sequential Solver		
	# Match	Ass deps[%]	Time[s]	# Match	Ass deps[%]	Time[s]
1	1067	86.4	26	1098	88.9	52
2	1064	86.2	28	1089	88.2	54
3	1070	86.6	26	1105	89.5	55
4	1063	86.1	21	1079	87.4	56
5	1317	87.9	40	1348	89.9	78
6	1068	86.5	26	1086	87.9	52
Avg		86.6	28		88.6	58

Figure 5.2: The obtained results when running Solver and Sequential Solver on instances A1 to A6 with the number of matchings made, the percentage of assigned departures and the computation time per instance. The computation times are an average of 10 runs.

i	Solver			Sequential Solver		
	f_1	f_2	f_3	f_1	f_2	f_3
1	168	16918	891392.9	137	17338	897493.6
2	171	16688	839267.3	146	16706	836657.0
3	165	17301	844093.0	130	17214	852404.0
4	172	16617	891026.9	156	17059	887858.8
5	182	26116	1082761.2	151	25801	1087251.8
6	167	16893	889097.4	149	16964	887934.9

Figure 5.3: The values of the objective functions for SOLVER and SEQUENTIAL SOLVER as returned by the checker.

5.3 Discussion

5.3.1 Computation time

The results are computed in less than one minute. The algorithm is solved in polynomial time:

$$\mathcal{O}(\textsc{Solver}) = \mathcal{O}(\textsc{Sequential Solver}) = |deps_{day}| \cdot |allTrains_{day}|^2 \cdot |res|^3.$$

This expression reflects the consistency of the choice of splitting the horizon in separate days. This indeed reduces the scope of the matching procedures but more importantly, reduces the sizes of the sets of departures and trains considered for the matchings. An other important observation is that the factor $|res|$ is due to computing the ways. Grouping these resources reduces this factor tremendously.

The difference in computation time between SOLVER and SEQUENTIAL SOLVER is due to the order in which the departures are assigned. The complexity's dominant factor is related to the JOINTMATCH procedure. In the SOLVER, this procedure is called after assigning all simple matches, which reduces considerably the set of unassigned trains considered when calling the JOINTMATCH procedure. In the SEQUENTIAL SOLVER, the size of the set of unassigned trains considered for a joint departure is larger. This increases the number of assigned departures for SEQUENTIAL SOLVER.

5.3.2 Assigned departures

The results previously introduced do not analyse the evolution of the assignments over time. It is indeed interesting to understand which departures are not assigned and why. The following two graphs 5.4 and 5.5 depict the percentage of assigned departures for both solvers, per day.

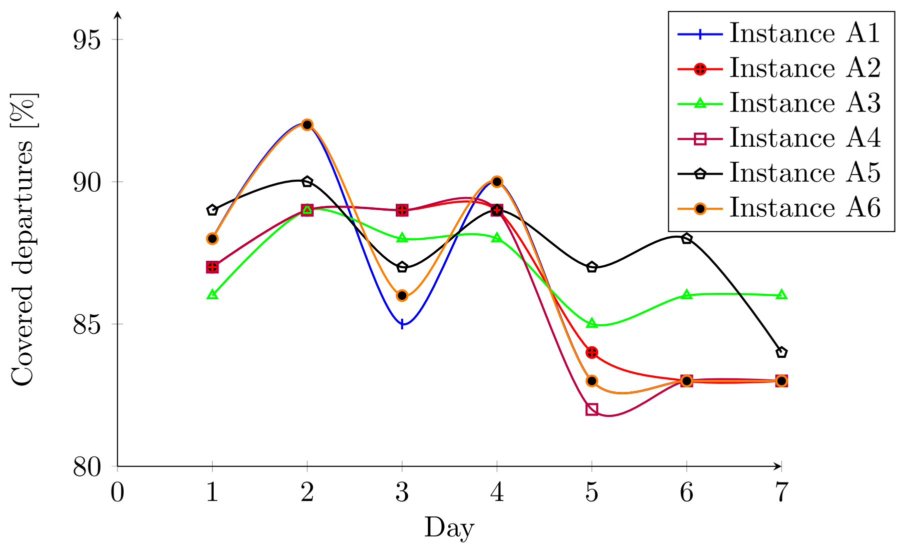

Figure 5.4: Illustration of the percentage of assigned departures per day for each instance by the Solver.

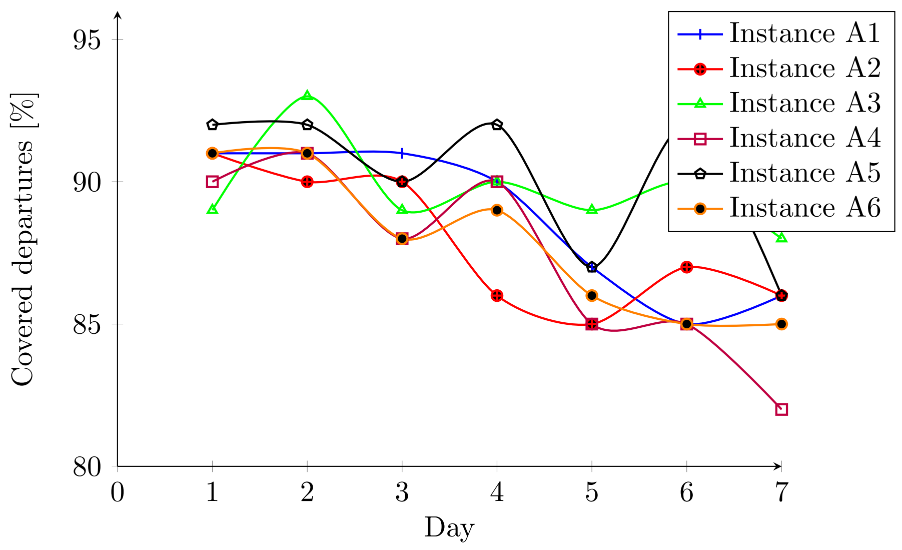

Figure 5.5: Illustration of the percentage of assigned departures per day for each instance by the Sequential Solver.

These two graphs show that the percentage of assigned trains decreases in time, while the number of available trains increases every day. The number of departures each day is indeed close to the number of arrivals which means that a train will remain in the system for each unassigned departure.
These trends are less strong for instances $A3$ and $A5$.
These two observations indicate two opposite trends. It may be interesting to try to understand this phenomenon.

Several causes can be invoked. First, the bound on the number of maintenances can prevent some trains to be assigned to a departure. Second, the hypotheses used to develop the algorithm state that all the trains of a joint departure are only assigned if they arrive assembled or if neither of them needs a maintenance. These phenomena will be further explored in the next sections.

5.3.3 Maintenances

As previously explained, the number of maintenances per day is bounded. The algorithm penalizes the use of maintenance to avoid reaching the limit unnecessarily early. The figures 5.6 and 5.7 illustrate the ratio between the number of maintenances accomplished each day and the daily upper bound.

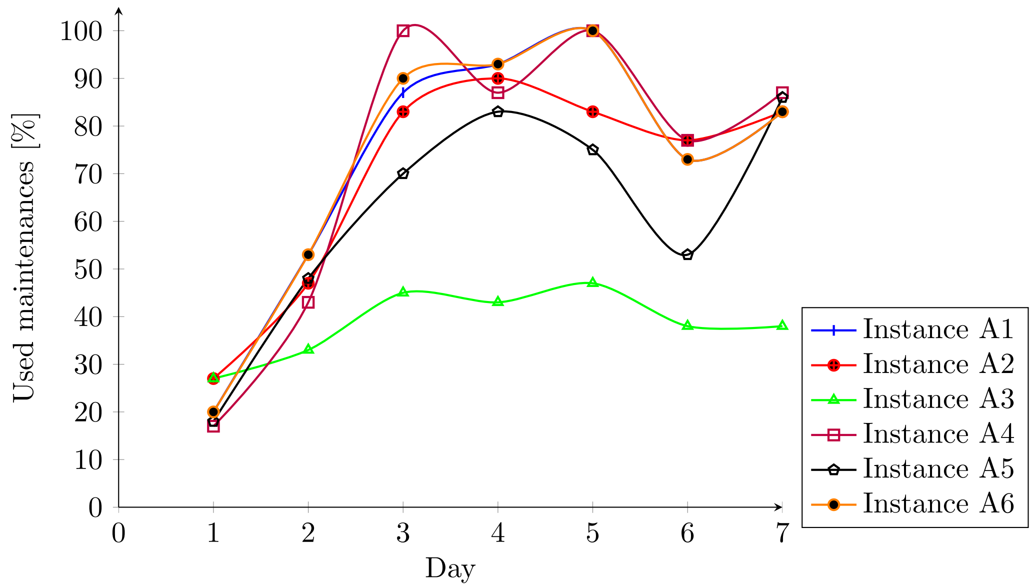

Figure 5.6: Illustration of the percentage of used maintenances per day for each instance by the SOLVER.

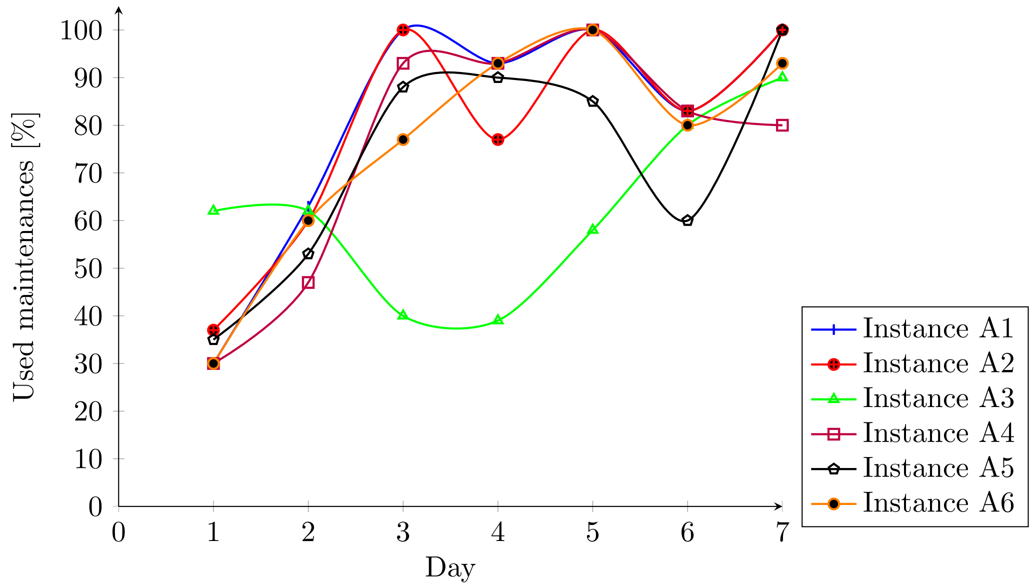

Figure 5.7: Illustration of the percentage of used maintenances per day for each instance by the SEQUENTIAL SOLVER.

These maintenance curves are increasing, in contrast to the observed trends of the percentage of assigned departures. The limit is reached after several days, except for the instances $A3$ and $A5$.
Whilst for the instances $A3$ and $A5$, the trends on the assigned departures and maintenance graph are less sloped. This could indicate that the two phenomena are linked.
This also could imply that allowing trains to undergo maintenance earlier could improve the number of assigned trains. This observation indeed suggests that some departures are not covered because the maintenance cannot be carried out on the same day as the departure. This limitation could be overcome by programming some maintenances earlier in the horizon. This would involve considering several days when trying to assign a departure.

5.3.4 Unassigned joint departures

As established earlier, the algorithm limits the possible junctions for a joint departure to trains that do not need maintenance. The figures 5.8 and 5.9 illustrate the ratio of joint departures in the set of unassigned departures per day.

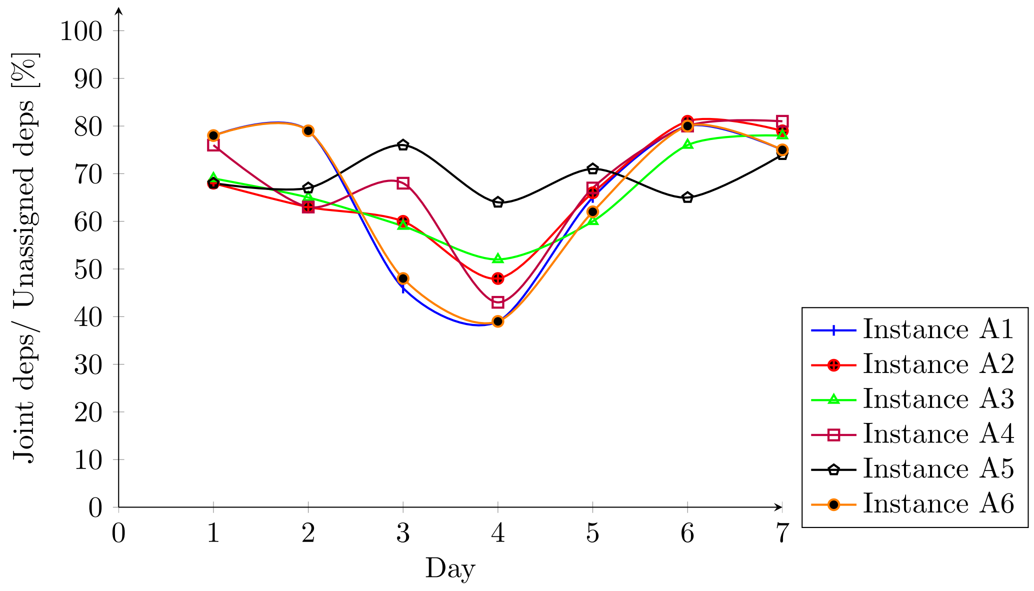

Figure 5.8: Illustration of the percentage of joint departures among unassigned departures per day for each instance by the Solver.

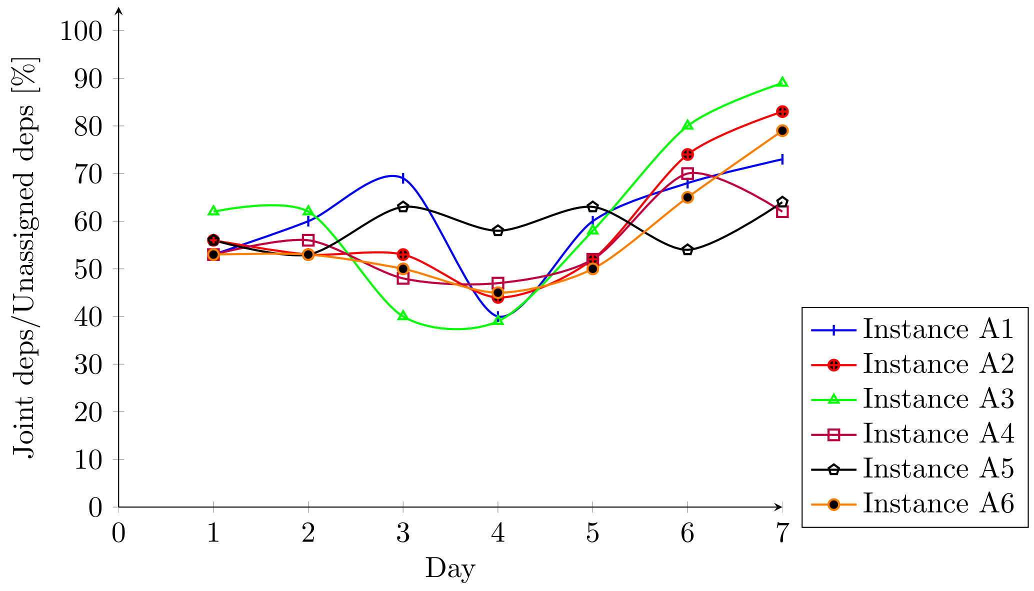

Figure 5.9: Illustration of the percentage of joint departures among unassigned departures per day for each instance by the Sequential Solver.

The curves exhibit a U-shape, less noticeable for the Sequential Solver. The increase at the end of the horizon might be due to the number of maintenances needed. The fact that more trains need maintenance to be assigned at the end of the horizon and that maintenances are not allowed for trains needing a junction to be assigned to a joint departure, could indeed explain the increase at the end.

The difference between the Solver and the Sequential Solver curve could be justified by the fact that the improvement brought by the Sequential Solver diminishes the effect of the order in which the matching procedures are called. It indeed considers all the assignments in chronological order instead of assigning all the simple matches before considering the joint departures. Therefore, it can assign more departures at the beginning, but the percentage of joint departures among all unassigned departures will similarly show an increase at the end of the horizon.

The number of trains in the system increases as does the number of trains needing maintenance during the horizon. This can explain the U-curve as a lot of trains are available in storage in the middle of the time horizon.

5.4 Benchmarking

As the problem was introduced as an international challenge, the obtained results can be compared to those obtained by the qualified teams.
Unfortunately, the details of the results are not provided. Two distinct informations are at our disposal:

1. the best results obtained for the 12 instances.

   For reminder, the algorithm developed and explained in this report only covers the 6 first instances.

2. the sums of the objectives functions for the first 12 instances for the 13 qualified teams.

The first data can be compared to the previously introduced results. The best results obtained for the 6 first instances are compared to the results obtained by the developed algorithm on figure 5.10. These best solutions provide schedules with approximately four times less uncovered departures.

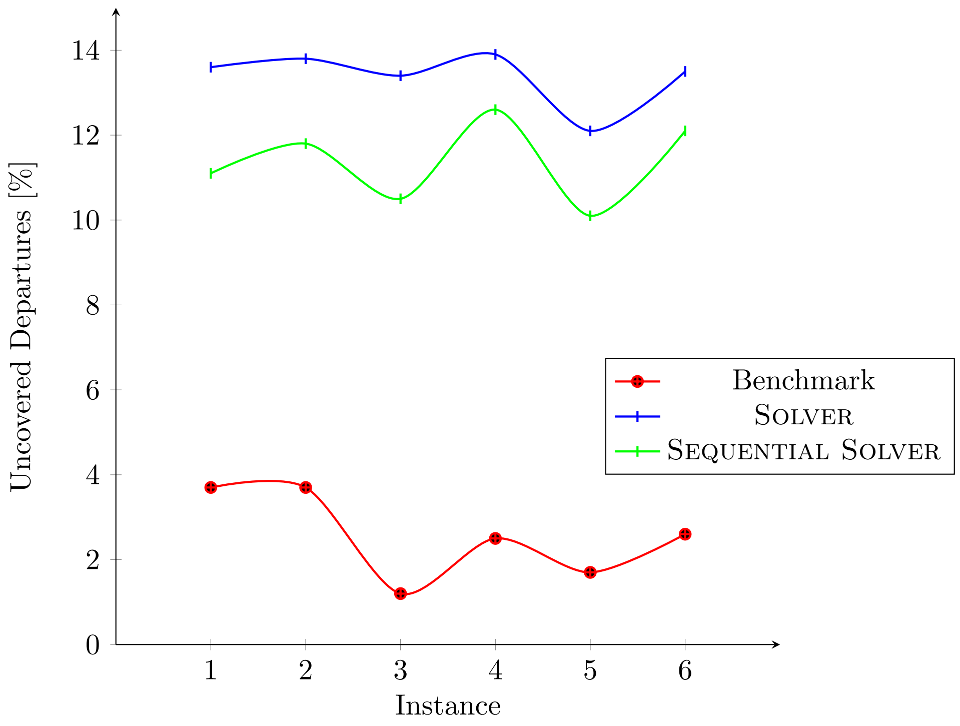

Figure 5.10: Comparison of the obtained results with the results obtained by the best participating team.

This states that the algorithm can be improved. However, the results obtained by the other qualified teams are quite spread.

The participants were divided into two groups:

Juniors team composed entirely of students (no PhD).

Seniors no restriction on the composition of the team.

Therefore, it might be relevant to compare the results to those of the other qualified teams. However, as the algorithm only covers the first 6 instances, comparing results is not straightforward.
The best benchmark results for the first objective function are shown on figure 5.11. They indicate that the instances $A1 - A6$ are not comparable to the instances $A7 - A12$. The graph indeed shows that the number of unassigned departures for the first set of instances are almost three times bigger than those for the last instances. The exact ratio is 2.79.

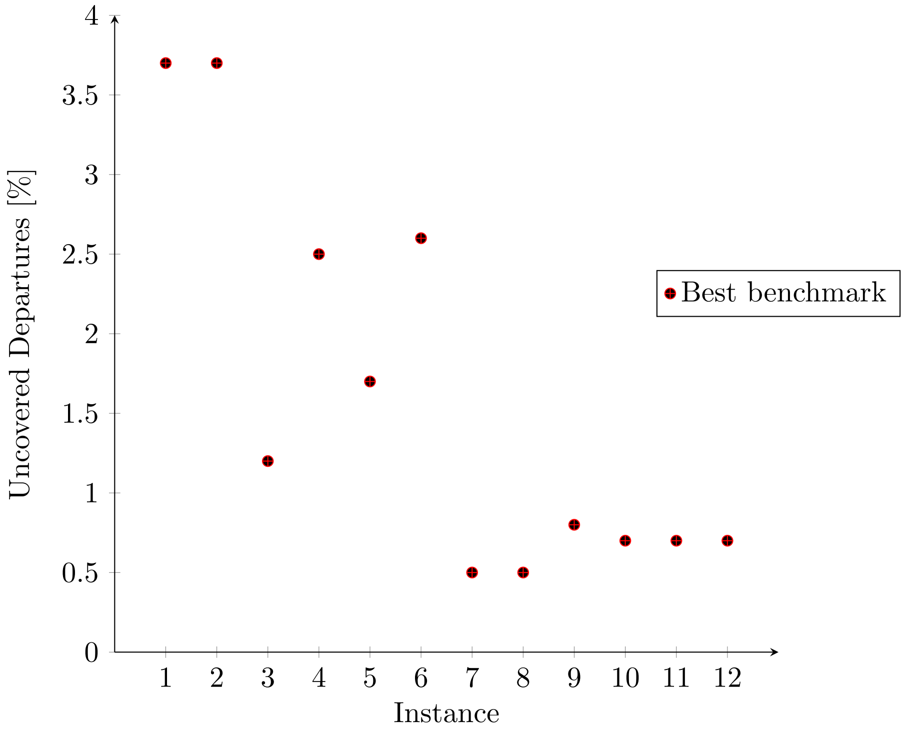

Figure 5.11: Illustration of the results obtained by the best participating team.

In order to have comparable results, this ratio was applied. The comparison with other qualified teams results are shown on figure 5.12.

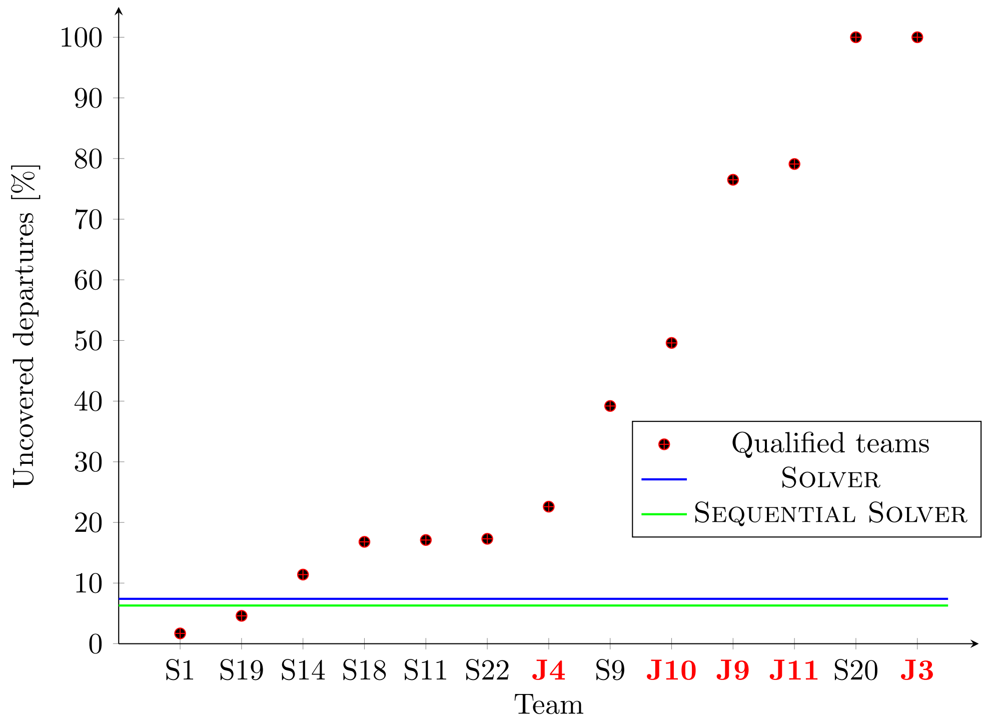

Figure 5.12: Illustration of the results of other qualified teams expressed in percentage of unassigned departures. The last two teams presented infeasible solutions.

This graph is quite encouraging, as the developed solver would be ranked third in general and first among junior teams. Besides, as the computational time is low, there is still room for improvement. Some ideas to improve the solution will be introduced in the following section.

The choice made to consider only the first objective function is confirmed by these results. As the results are very spread on this objective, the other objectives are indeed not taken into account for the ranking.

5.5 Future improvements

As this last section revealed, there is room for improvement. Just by changing the order of the matching, the solution was indeed improved. The analysis of the results have showed that the first objective function could be enhanced by allowing earlier maintenance to avoid congestion later on in the horizon and by developing a more advanced matching procedure for the joint departures allowing maintenances before junction. Given the speed of the algorithm, there are also opportunities to improve the second and third objective functions.

The second objective function concerns the conflicts on track groups and yard overloads. These conflicts could be decreased not only by taking this factor into account when querying the oracle for a compatible way during a timeslot, but also by post-optimizing the computed ways: the gates used during a way are chosen while post-processing. Actually, the algorithm always chooses the first gate regardless to other ways using the same resources at the same time. Optimizing the choice of gates could be done efficiently during post-processing. The yard overloads could be reduced indirectly by assigning more departures which implies there are less trains stored in the system, but also directly by using other resources for storage such as facilities and single tracks.

The third objective concerns performance costs. Among those are the (dis)-junction costs and the preferred platform costs. The actual algorithm will disjoin all the remaining joint arrivals. However, only disjoin those which will be used separately afterwards would decrease the (dis)-junction costs. The preferred platform costs could be decreased by taking them into account during the post processing.

Chapter 6

Conclusion

In this paper, an algorithm has been developed to solve the problem of handling trains between their arrival and departure while managing the resources in a railway station. While facing several objectives that need to be minimized, the algorithm focusses on the first one which consists in minimizing the number of unassigned departures. Letting a train leave for its journey is indeed more important than not letting it leave because it can not depart on its preferred platform.

In order to find and comprehend the solution, this work begins with a detailed description of the problem with its objectives and constraints. This has led to the actual objects definition in the algorithm. Afterwards, several approaches which led to the final solution are described, with an analysis why they did not always work. Most of them had indeed complexity issues and were discarded. However, they helped building up the functional solution.

Subsequently, the obtained results were benchmarked. It has shown that the developed algorithm is competitive. First of all, it is very fast. The algorithm indeed takes less than one minute while the bound on computation time of the challenge is ten minutes. Secondly, the benchmarking to the first objective, e.g. the number of unassigned departures, revealed that the solution extrapolated for the last six instances had the best place among the junior participants while being in the top three of the qualification results of the challenge. Finally, the decision to restrict the solution to focus on the first objective was a good one, as not only the qualification of the challenge was based on it, but as it is also more consistent.

This challenge allowed us to face a real industrial problem as we could encounter during our professional career as an engineer while putting into practice theoretical concepts acquired during our studies at the UCL.

Bibliography

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Graph algorithms. The MIT Press, 2009.

[2] M. Odersky, L. Spoon, and B. Venners. *Programming in* SCALA. Artima, 2010.

[3] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[4] F. Ramond, F. Marcos, and F. Sourd. Roadef euro 2014 challenge - trains don't vanish : Rolling stock unit management on railway sites, 2013. Available from `http://challenge.roadef.org/2014/files/Challenge_sujet_131031.pdf`.

[5] F. Ramond, F. Marcos, and F. Sourd. Roadef euro 2014 challenge - checker v1.5, 2014. Available from `http://challenge.roadef.org/2014/files/Checker_v1.5.zip`.

[6] ROADEF. Roadef/euro challenge 2014: Trains don't vanish !, 2014. Available from `http://challenge.roadef.org`.

[7] Kevin Stern. Hungarian algorithm java implementation, 2012. Available from `https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/optimization/HungarianAlgorithm.java`.

[8] Wikipedia. Bin-packing problem, 2014. Available from `http://en.wikipedia.org/wiki/Bin_packing_problem#First-fit_algorithm`.

[9] Wikipedia. Hungarian algorithm, 2014. Available from `http://en.wikipedia.org/wiki/Hungarian_algorithm`.

Appendix A

Examples of a railway station

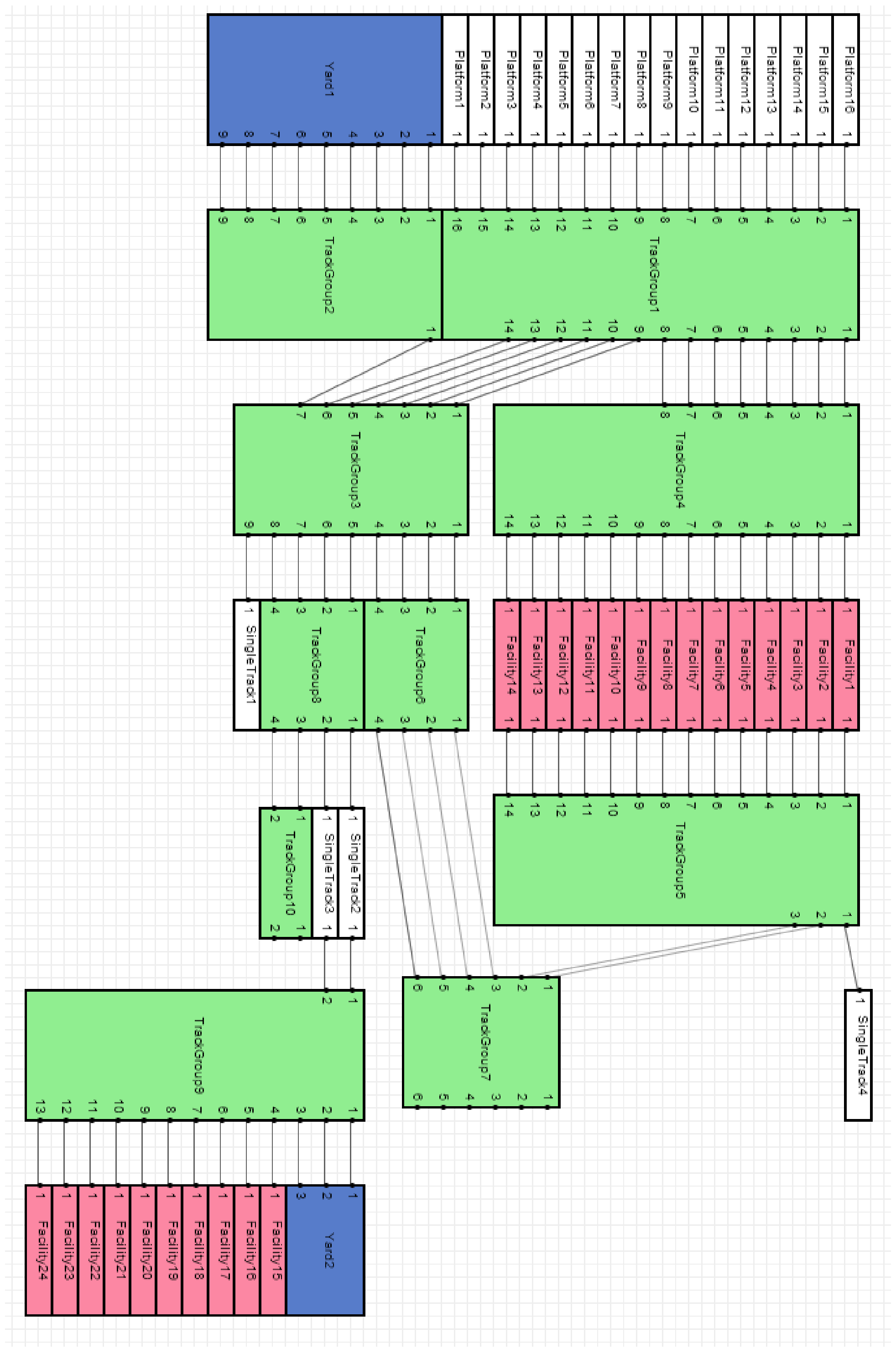

Figure A.1: Graphical representation of a system for the instances : A1, A2, A3, A5 and A6.

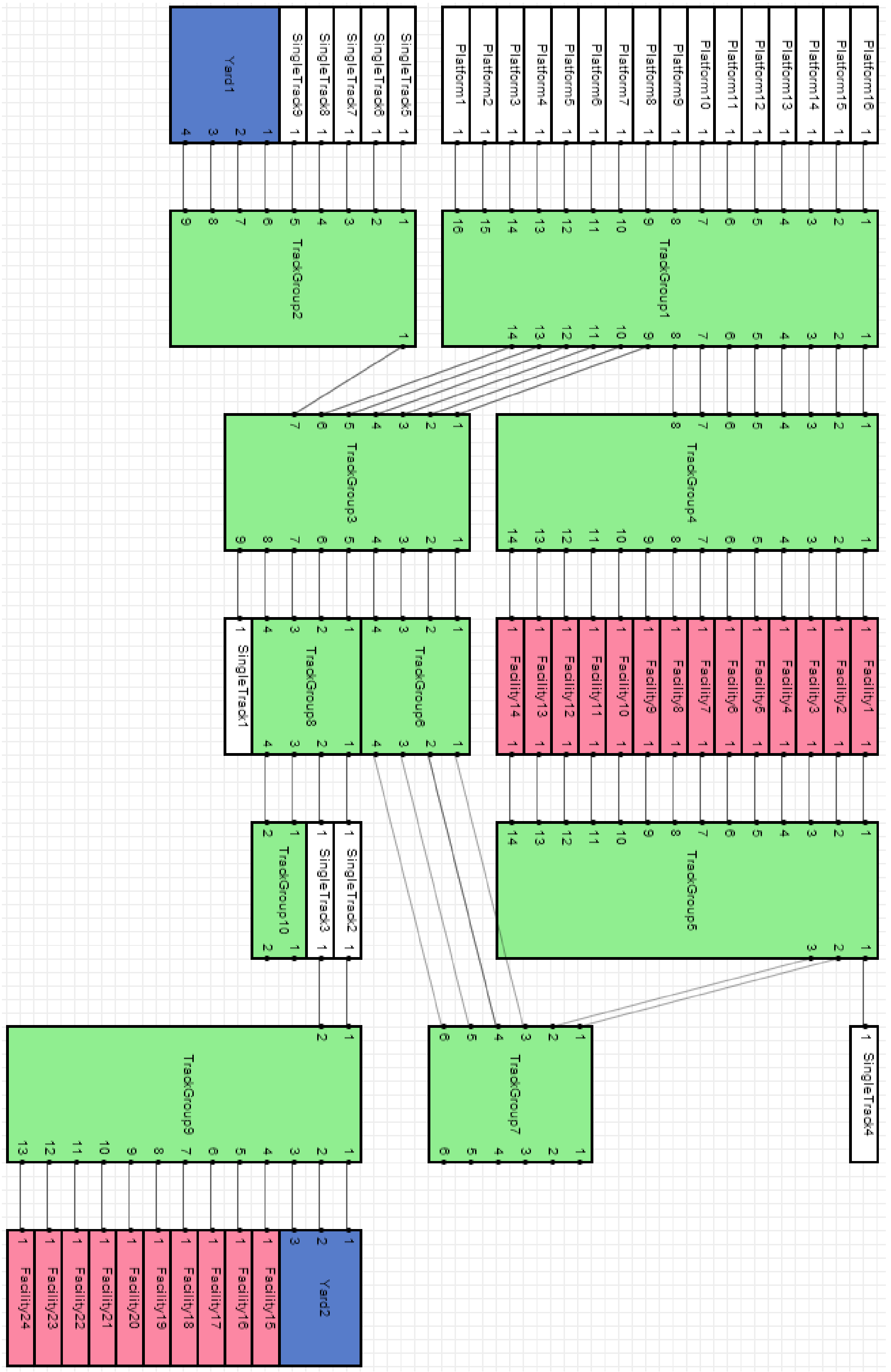

Figure A.2: Graphical representation of a system for the instance A4.

Appendix B

Complexity of the main Solver functions

putInitOnYard$(init : Array[Train])$

$$\mathcal{O}(|init| \cdot |res|^3 \cdot |trainCat| \cdot (|allTrains|))$$

initOracleDay$(oracle : Oracle, init : Array[Train], day : Int)$

$$\mathcal{O}(|imposedConsumptions| \cdot \log(|imposedConsumptions|) + |init|)$$

fDay$(a : Array[Train], d : Array[Departure], day : Int)$

$$\mathcal{O}(|a| + |d|)$$

fSM1$(a : Array[Train], d : Array[Departure])$

$$\mathcal{O}(|a| + |d|)$$

fJA$(a : Array[Train], d : Array[Departure])$

$$\mathcal{O}(|a| + |d| \cdot |jointDep|)$$

fJB$(a : Array[Train], d : Array[Departure])$

$$\mathcal{O}(|a| \cdot ((|plat| + |yards|) \cdot (|trainCat| + |allTrains|) \\ + |res|^3 \cdot |trainCat| \cdot |allTrains|) + |d| \cdot |jointDep|)$$

fSM2$(a : Array[Train], d : Array[Departure]$

$$\mathcal{O}\left(|d| \cdot (\log(|d|) + |jointDep|)\right)$$

fSM3$(a : Array[Train], d : Array[Departure]$

$$\mathcal{O}\left(|d| \cdot (\log(|d|) + |jointDep|)\right)$$

simpleMatch$(a : Array[Train], d : Array[Departure])$

$$\begin{gathered}\mathcal{O}(|d| \cdot (|a| \cdot \log(|a|) \\ +|a| \cdot ((|plat| + |yards| + |fac|) \cdot (|trainCat| + |allTrains|) \\ +|res|^3 \cdot |trainCat| \cdot |allTrains| + |res| + |arr|)\end{gathered}$$

jointMatchA($jA : Array[(Int, List[Train])], jD : Array[List[Departure]])$

$$\begin{gathered}\mathcal{O}(|jD| \cdot (|jA| \cdot \log(|jA|) \\ +|jA| \cdot (|res|^3 \cdot |trainCat| \cdot |allTrains| \\ +|plat| \cdot (|trainCat| + |allTrains|)) + |res| + |arr|)\end{gathered}$$

jointMatchB$(a : Array[Train], jD : Array[List[Departure]])$

$$\begin{gathered}\mathcal{O}(|jD| \cdot (|a| \cdot \log(|a|) \\ +|a| \cdot (|res|^3 \cdot |trainCat| \cdot |allTrains| \\ +|plat| \cdot (|trainCat| + |allTrains|)) + |res| + |arr|)\end{gathered}$$

processRemainingArrivals$(a : Array[Train])$

$$\begin{gathered}\mathcal{O}(|a| \cdot ((|plat| + |yards|) \cdot (|trainCat| + |allTrains|) \\ +|res|^3 \cdot |trainCat| \cdot |allTrains|) + |res|)\end{gathered}$$