Shunting passenger trains: getting ready for departure

Marjan van den Akker* Hilbrandt Baarsma† Johann Hurink†
Maciej Modelski‡ Jacob Jan Paulus† Ingrid Reijnen§
Dan Roozemond‡ Jan Schreuder†

Abstract

In this paper we consider the problem of shunting train units on a railway station. Train units arrive at and depart from the station according to a given train schedule and in between the units may have to be stored at the station. The matching between arriving and departing train units and the scheduling of the movements to realize this matching is called shunting. The goal is to realize the shunting using a minimal number of shunt movements.

For a restricted version of this problem an ILP approach has been presented in the literature. In this paper, we consider the general shunting problem and derive a greedy based heuristic approach and an exact solution method based on dynamic programming. Both methods are flexible in the sense that they allow the incorporation of practical planning rules and may be extended to cover additional requirements from practice.

KEYWORDS: shunting trains, greedy heuristic, dynamic programming

1 Introduction

In this paper we study a practical train shunting problem proposed by Dutch Railways. This problem has already been studied by the Kroon et al. [7], but their work does not exploit the full potential of shunting trains.

Shunting of trains is a process that supports the execution of the train schedule at the station. Trains arrive at and depart from the station according to the train schedule. Each arriving and departing train may consist of multiple and possibly different types of train units. This composition of the trains is specified in the train schedule. For an arriving train it now has to be decided what the next duties of the arriving train units are and for a departing train it has to be guaranteed that the corresponding train units are available on time on the platform. During rush hours almost all train units available are required to transport passengers and, thus, are on duty, but in between, and especially during the night, a lot of train units are not

[0] We thank Leo Kroon, Dutch Railways for supplying this problem and his valuable insight and Daniël Roelfsema, Scar Groep who also participated in the study group

* Utrecht University, Department of Information and Computing Sciences
† University of Twente, Department of Applied Mathematices
‡ Eindhoven University of Technology, Department of Mathematics and Computer Science
§ Eindhoven University of Technology, Department of Technology Management

needed for transporting passengers. Thus, train units may have to be parked at a shunt yard of a station for a certain period. An example of such a shunt yard is given in Figure 1, which represents the infrastructure of the station and shunt yard of Alkmaar.

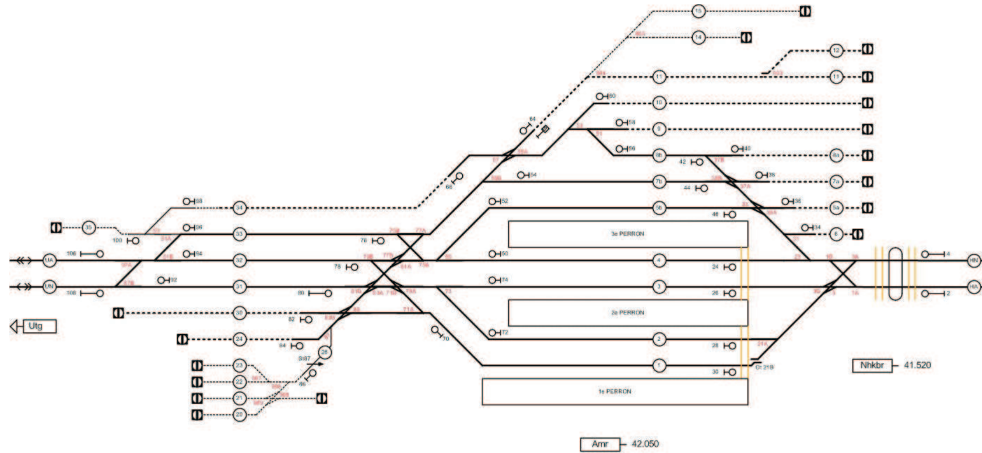

Figure 1: Shunt yard and station of Alkmaar

The train units are classified according to their types and subtypes. Train units of the same type can be combined into longer trains, even if their subtypes differ. An example of a train unit is an ICM (Inter City Material) with 3 carriages, as shown in Figure 2. ICM denotes the type, and the subtype is specified by the number of carriages. There also exist ICMs with 4 carriages, which can be combined with the ICM with 3 carriages since they are of the same type, although not of the same subtype.

Figure 2: Train unit of type ICM with 3 carriages

To park a train unit, a crew has to take several actions. If the train has to go only in one direction, the engine driver can stay on one side of the train and drive the train directly to the shunt yard. This is not always possible and it may e.g. be the case that the train has to go forth, back, and forth again to be parked. In that case the engine driver has to switch places two times, since he always has to be in the front of the train. Each time going back or forth is called a shunt movement. So, if we only need to go forth, this is counted as one shunt movement and if we have to go forth, back, and forth then this is counted as three shunt movements.

When a train unit parked at the shunt yard is needed again in the schedule to transport passengers, it has to be taken out of the shunt yard and put at the platform from which the corresponding train will depart. Again it may happen that

several shunt movements are needed to transport the unit to the platform. But it may even be worse in the sense that no train unit of the desired type is in the front of a shunt track. In this case, before getting the desired train unit, first some other blocking train units of another (sub)type have to be removed from a shunt track. This can also take several shunt movements.

As a consequence, to execute the train schedule, also a feasible shunt schedule is required at each station. A shunt schedule consists of a list of actions that indicate which train units are shunted and between which places. Besides this, also the exact shunt movements of the train units can be specified. A shunt schedule is feasible if all arrivals and departures of the train schedule can be executed in the desired way. This implies for example that a platform has to be empty when a train is passing through or that train units of the desired (sub)types and in the desired order are at the right time at the right platform for a departing train.

However, not every feasible schedule is desirable: if the shunt schedule consists of many shunt movement, the schedule causes a high workload for the crew and is very sensitive for disruptions. This can cause delays in the train schedule, which should be avoided. Therefore, the goal is to have a shunt schedule with a minimal number of shunt movements.

Besides the main goal to create shunt schedules with a low number of shunt movements, some other practical aspects have influence on the quality of a schedule and lead to additional rules to be taken into account in creating shunt schedules. For example, for the crew it is convenient to have a lot of consistency or regularity. This implies for instance that shunt tracks of the shunt yard should be used only for train units of the same type. Another practical aspect focuses on shunt movements just before a departure. Small disruptions in a shunt schedule with such movements directly may lead to delays of departing trains and, therefore, may disturb the train schedule. As a consequence, it is desirable that the number of shunt movements for a train that needs to depart is minimized. It would be even best if the train units are already waiting in the needed composition for the departure at the shunt yard.

1.1 Problem Description

The input for the shunting problem at a given railway station consists of the train schedule at that station and the layout of the station (platforms and shunt yard). The given train schedule prescribes the train arrivals and departures at the railway station. Each of these events is characterized by a time, composition of the train, direction, platform and whether the train arrives or departs. Since not all arriving train units are scheduled to leave the station immediately, the train units that stay behind may have to be stored at the shunt yard to clear the platform for the next train.

The shunt yard consists of a number of shunt tracks to store train units. Most of the shunt tracks are dead-end tracks. This implies that train units are blocked by train units parked at a later time. Thus, train units arrive and depart in *last in first out* (LIFO) order. The shunt tracks and platforms have a limited capacity for storing train units. There is a network of tracks connecting the shunt tracks with the platform tracks.

Between successive events of the train schedule, it may be necessary to move train units to make the next event possible. Such movements are called shunting. The route taken defines the cost of the shunt movement. A one-directional movement is counted as one movement and every change of direction is counted as an extra movements. A solution is a list of shunt movements that take place between the events such that all events can take place. The objective is to find a solution with minimum number of shunt movements.

In this paper we assume a timeless model; i.e. we assume that a shunt movement takes zero time. This implies that an unlimited number of shunt movements can be performed between two events. However, it is possible to add extra constraints within the developed methods, which restrict the number of shunt movements between two events.

1.2 An Example

To illustrate the shunting problem we give a small example. Consider a railway station with the layout given in Figure 3. In this example we consider four types of

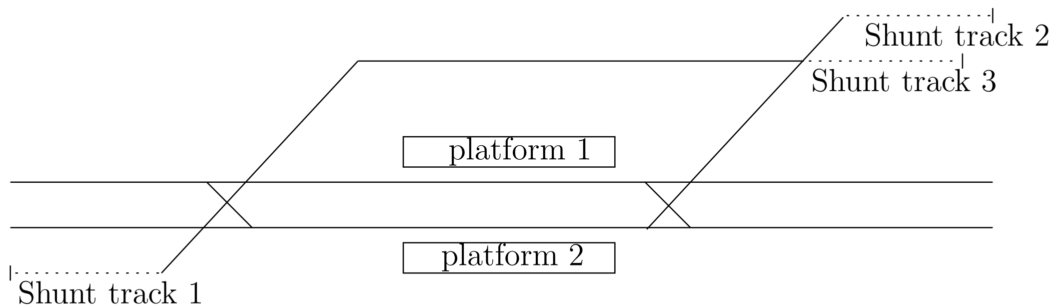

Figure 3: Layout of the example station

train units, denoted by A, B, C and D. Each train consist of some train units of these types. When we talk about a train AB we mean a train consisting of a train unit of type A and a train unit of type B, where the type A unit is positioned to the left of the type B unit. This is regardless of the direction the train is traveling in. Thus, trains AB and BA are different in composition. We assume that the capacity of all shunt tracks and platform tracks is limited to accommodate 3 train units. According to the train schedule the following trains are arriving and departing in the given order.

e_1	Train AB arrives from the left-side at platform 1.
e_2	Train AA arrives from the right-side at platform 2.
e_3	Train CCC arrives from the right-side at platform 1.
e_4	Train CC departs from the platform 1 to the left-side.
e_5	Train AA departs from the platform 2 to the right-side.
e_6	Train DC arrives from the left-side at platform 1.
e_7	Train CDC departs from platform 1 to the right-side.
e_8	Train BA arrives from the right-side at platform 2.
e_9	Train BB departs from platform 2 to the right-side.
e_{10}	Train AA departs from platform 1 to the left-side.

In this small example there are already a number of non-trivial shunting decisions to make. It is not difficult to verify that the following solution is a valid shunt schedule.

Between e_2 and e_3	Shunt train AB from platform 1 to shunt track 2.
Between e_5 and e_6	Shunt train C from platform 1 to shunt track 1.
Between e_6 and e_7	Shunt train C from shunt track 1 to platform 1.
Between e_8 and e_9	Shunt train A from platform 2 to shunt track 2, and shunt train AA from shunt track 2 to platform 1, and shunt train B from shunt track 2 to platform 2.

The solution contains six shunt movements. In this example the choice whether to shunt to the tracks on the left-hand side or to the tracks on the right-hand side is the most important decision. Observe that shunting train unit C to any of the shunt tracks on the right-hand side is not a good decision. When moving the unit back, it has to go around the DC train, to connect to it from the left to forms the CDC train. Going around the DC train implies a change of direction in the shunt movement and is counted as two shunt movements. Furthermore, if the AB train is shunted to the shunt track on the left-hand side, the efficient moves between e_8 and e_9 would not be possible. It turns out that the above solution is indeed optimal for the example.

1.3 Complexity of the Shunting Problem

The general problem of integrated matching (to which departing trains are the units of an arriving train matched?) and parking of train units is introduced in [7] and in [8] its computation complexity is resolved. The general problem as well as the subproblem of matching the train units and the subproblem of parking the train units are shown to be NP-hard.

In the train matching problem we are given a set of arriving trains and a set of departing trains. We are supposed to partition the incoming trains into parts which can later be assembled into departing trains. Since each produced part is shunted separately, our main goal is to minimize the number of parts into which we partition the arriving trains. This problem is a generalization of the minimum common string partition problem known from computational biology. In [5] the minimum common string partition problem is shown to be NP-hard even if we

restrict ourselves to instances with only two strings on input. This means that the train matching problem is hard even if we are given just one arriving and one departing train.

Blasum et al. [1] introduce a problem of scheduling the departures of trams from a shunt yard in the morning. This problem turns out to be NP-hard and the authors provide a dynamic program for a special case of the problem with restricted number of shunt tracks. This problem can be seen as a subproblem of our shunting problem where all the trains are already placed in the shunting yard.

Cornelsen et al. [2] study the problem of generating shunt-free schedules in stations consisting of parallel two-sided tracks. They reduce the problem to a graph coloring problem of a conflict graph resulting from the train schedule. For most of the versions of the problem the conflict graph is perfect and can be colored in polynomial time. For other cases efficient approximations algorithms are presented.

In similar setting Dahlhaus et al. [3] consider a problem of grouping of train units. In this problem a sequence of incoming train units is given. Each train has to be sent to one of the parallel tracks and later pulled out to the other side. The outgoing sequence has to be ordered in such a way that units of the same type are grouped together. Designing a schedule that minimizes the number of used tracks is shown to be NP-hard.

In freight train classification hump yards are commonly used for shunting. Jacob et al. [6] model the shunting task as a problem of finding a set of binary codes. It allows them to find optimal solutions for most versions of the problem. Some other versions are shown to be NP-hard.

1.4 Current Solution Approach

To solve the shunting problem, Dutch Railways is currently using an ILP-model of the problem [7]. However, this ILP-model has a number of drawbacks. First of all it does not cover all possible shunting moves. For example it does not allow trains to stay at a platform, waiting to be combined with a next train. It is clear that such a waiting possibility can be beneficial. Moreover, it does not model any kind of rearrangement of trains between different shunt tracks. Whenever a train arrives, it either has to be shunted away or depart immediately.

Furthermore, in the current ILP-model the number of variables and constraints is already very large, and extending this model to cover the above shunting possibilities would increase the number of constraints and variables even further. Although, for a typical instance the current model can be solved within a reasonable time, one may expect that the extensions make the number of variables so big that the computational time required to solve the problem becomes unacceptably large.

1.5 Goal of the Research

The task of this paper is to present alternative approaches to the shunting problem which do allow waiting on platforms and rearrangements of train units between shunt tracks. In the following we describe two solution approaches, one which aims for finding fast a reasonable cost solution (a greedy algorithm) and one which aims

for the optimal solution (a dynamic-programming algorithm). We conclude with an outlook on future research.

2 First Approach: A Greedy Algorithm

In this section, we present a heuristic approach for the shunting problem. This heuristic has to be fast and has to result in a reasonable good solution. The basic idea is to scan the event list and iteratively decide which actions to take. The decisions in each iteration are based on the situation resulting from the previous decisions and the current event. In this way, the approach tries to locally extend the given situation as well as possible and, therefore, falls in the category of *greedy* approaches.

From practice it is indicated that planners prefer situations where the train units of departing trains are already waiting somewhere (either on a platform track or a shunt track) in the composition they have to depart in. We take this philosophy, *be ready for departure*, as a guideline for building the greedy approach. As a consequence, we scan the event list backwards in time and make the decisions in such a way that they lead to the desired composition for the departing trains. Our algorithm consists of four main steps, which we explain in more detail later.

The Greedy Algorithm:

1. Start with empty platforms and shunt tracks
2. Scan the event list backwards in time, and for each event DO
3. If the event is a departure event: assign the entire train to a shunt track
4. If the event is an arrival event: match the train units to train units already placed on the shunt track

The most important steps in our algorithm are steps 3 and 4. In these steps the main decisions are made. In step 3 we decide on which shunt track we set the train ready for departure. At this point, we do not care how these train units come to this shunt track, but just decide that the units wait on the assigned shunt track for departure. How these units arrive on their positions on the shunt track will be decided in subsequent iterations. Possible rules for assigning the trains to shunt tracks are given later. In step 4 we match the train units of arriving trains to train units that are already placed for departure from a certain shunt track in one of the previous iterations. Again, concrete rules for this matching are given later.

Example To get a better understanding of the basic idea of this greedy approach we present an example consisting of two platforms and two shunt tracks. The event list of this example is presented in Table 1 (in this table platform numbers are given as well, since we use them later on).

If we scan the event list from the back, we first have to treat event e_8. Since this is a departure, we may decide to assign this train to shunt track 1. The situation

Events	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
Arrivals	C	A	BB			AA		
Departures				B	BA		AA	C
Platform	2	1	2	2	1	2	1	2

Table 1: Event list Example 1

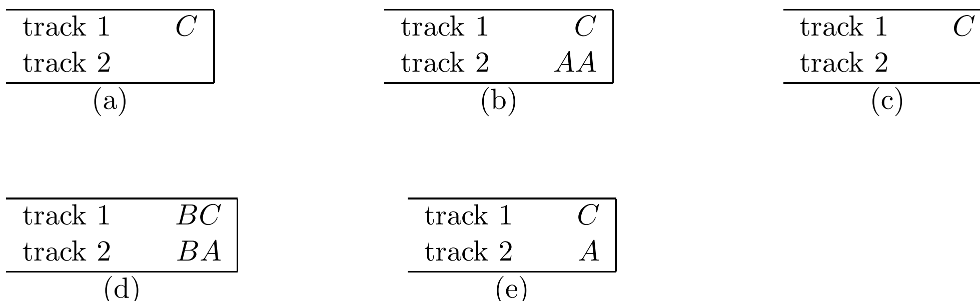

Figure 4: Situations on the shunt track for Example 1

on the two shunt tracks after this decision is given in Figure 4(a). The next event e_7 is also a departure, and we may assign the train AA to shunt track 2 (see Figure 4(b)). For shunting the arriving train units of event e_6 we now have the nice option to match the whole train to the two train units of type A being assigned to shunt track 2. By this matching, i.e. shunting the two train units to shunt track 2, this shunt track gets empty and the resulting situation is as in Figure 4(c). Next, we may assign the train of departure event e_5 to shunt track 2 and the train of departure event e_4 to shunt track 1 resulting in the situation as in Figure 4(d). If we now treat the arriving event e_3, the train consisting of two type B units cannot be matched as a whole to a shunt track, but we have to split the train and match the two type B units to the two type B units in front of the two shunt tracks leading to the situation in Figure 4 (e). Note that this matching leads to two separate shunt movements. Finally, the two arriving events e_2 and e_1 are processed by matching the corresponding train units to the units of the same type being still on the shunt tracks.

As can be seen from the above example, the presented algorithm decides for each arriving train unit to which departing unit it is coupled and via which shunt track this assignment takes place. In this way shunt movements are specified. For departing trains the shunt movement can be done with the train as a whole since we always assign to be 'ready for departure'. For arriving trains more complex shunt movements may be necessary. In the above example, all the shunt movements were directly possible, but in general it may be necessary to rearrange the train units on the shunt tracks at certain moments to achieve a feasible solution. If, for instance, the arriving event e_3 would have been of an AC train, first the two B units already being at the shunt tracks would have to be removed to place the A and C unit at the dead-end of the shunt tracks.

The advantage of the presented approach is that it always gives a feasible solution

as long as the list of arrival and departing events is consistent, in the sense that there is never a negative stock of train units of some type. Furthermore, the departing trains can always be handled efficiently. The price to achieve this is that we may create costly shunt movements for arriving events.

In the following we sketch some possible improvements of the greedy method and give some more detailed information on possible implementations of the assignment and matching in steps 3 and 4.

Leaving train units on platform tracks One of the goals of this research is to develop methods which allow the option of leaving train units on platform tracks or to move it from one platform to another platform without parking it in between at the shunt yard. A simple approach is to scan the solution achieved by the greedy heuristic and to search for 'shortcuts'. In the above example such a short cut is for example possible between the events e_6 and e_7. The AA train arriving on platform 2 (event e_6) may be passed directly to platform 1 from which it departs as event e_7. In this way, the AA train does not have to be moved to the shunt track 2 probably saving some shunt movements. Another short cut is possible by leaving one of the arriving type B units of event e_3 on platform 2. In this way the departing train of event e_4 is already on the platform without any movement.
A more effective method than a scan after finishing the greedy approach may be to take such possibilities already into account during the greedy algorithm. If we have to assign a departing event in step 3 of the algorithm, we may scan the event list some positions further back in time to detect if there is an assignment of this train to a shunt track which allows using shortcuts. Such an assignment is preferable over other assignments.

Delaying the shunt movement If for an arriving event the shunt movements of possible matchings take a large effort (e.g. the corresponding units do not occur at a reachable end of a shunt track), we may scan the event list back in time to see if we can improve the situation by letting some other arriving trains wait on their platform. To clarify this possibility, let us assume that in the given example the train of event e_2 is a B train and that of e_3 an AB train. If we now deal with event e_3, no easy matching is possible since on shunt track 2 the train units are not in the correct order (see Figure 4 (d)). But we may delay the movements belonging to event e_2, since this event is on a different platform. For the greedy approach this means that we consider event e_2 before e_3. By matching the B unit of that train to the B unit in front of shunt track 2, we achieve a situation where on shunt track 1 we have BC and on shunt track 2 we have A. Now we can match the two units in front of the two shunt tracks to form the AB train of event e_3.
Formally, in step 4 of the greedy approach we may search the event list backwards and consider for each platform the first occurring event. If this event is an arrival, we may delay this arrival over the current event by treating it before the current event. Note that it is not possible to delay departure events or two arriving events on the same platform.

Events	e_1	e_2	e_3	e_4	e_5	e_6
Arrivals	A	A	B			
Departures				A	A	B

Table 2: Event list Example 2

track 1	AB
track 2	A

track 1	AA
track 2	B

Figure 5: Situations on the shunt track for Example 2

Assignment rules in step 3 Up to now, we have not specified the way how we assign in step 3 the trains to shunt tracks. The most simple way is to assign them in some *round robin* way or to assign them based on some priorities of the tracks. Possible priorities may be: smallest number of shunt movements to reach the platform, largest free capacity, etc. However, it may be worthwhile to incorporate also planning rules of the planners of Dutch Railway into this step. One such rule is, for example: do not park more than two different unit types on the same shunt track. Furthermore, again a backward scan in the event list by a few positions may help to overcome problems in the next iterations. Consider for example the event list in Table 2. Two possible shunt track assignments after treating the events e_6, e_5, e_4 are given in Figure 5. The first assignment is made using round robin, but has not taken into account the arriving B train. The second assignment does not have this problem.

Matching rules in step 4 As in step 3, also in step 4 there may be some freedom in matching the arriving trains to units already assigned to the shunt tracks. Again, this decision may be based on priority rules like the number of necessary shunt movements, but as in the previous case, it may also be worthwhile to incorporate some backward scan to see which resulting remaining situation on the shunt tracks forms the better situation for the next events. To illustrate this, assume that in Example 2 the events e_2 and e_3 are interchanged and that after considering event e_4 we have the first shunt track assignment in Figure 5. If we now have to deal with event e_3, matching the type A unit of this event to the A unit in front of shunt track 1 allows a direct excess to the B unit on that track in the next iteration. Having chosen for the A unit on shunt track 2 would not have given this possibility leading to a situation where units on the shunt tacks have to be rearranged.

Improvements Several of the suggested improvements contain some sort of partial backward scan of the event list to improve the decision for the current event. In principle this means, that some sort of simultaneous treatment of several events is considered. Based on the outcome of this, a decision for the current event is fixed. This treatment of several events simultaneously, can be seen as a new optimization problem on its own. This problem gets harder the more events are taken into

account. An interesting topic of further research is to try to find a good balance between the effort spent on this backward scan and the improvement in quality. Furthermore, concrete decision rules for the treatment of several events simultaneously have to be developed.

To sum things up: the greedy algorithm we have developed is able to create feasible schedules for the shunting problem quite fast. However, without additional improvements, the achieved solution may not be of much practical use. Above we have shown, that the basic structure of the method forms a good framework which easily can be extended by more sophisticated elements and even with rules used by planners.

3 Second Approach: A Dynamic Programming Algorithm

To solve the shunting problem, a response to each event, arrival or departure, has to be given. This response has some influence on the position of train units on the different tracks and platforms and has to guarantee, that the next event can take place. *Getting ready for a departure* means that the right train composition is on the departure platform, and *getting ready for an arrival* means that the arrival platform can accommodate for the arriving train.

To describe the given situation of train units on the different shunt tracks and platform tracks (called a configuration), we define a vector S. S is called the state of the system and is an ordered list of train type units on each of the tracks. For the example given in Section 1.2, the first element in S describes the train units on platform 1, the second on platform 2, the third on shunt track 1, etc. With (S, e_i) we indicate that the train units are in state S just before event e_i happens. The pair (S, e_i) is valid if and only if event e_i can take place with the given state S; i.e. in state S we are ready for event e_i.

With this notation we can describe the solution for the example of Section 1.2 as in Table 3.

$$\begin{pmatrix} - \\ - \\ - \\ - \\ - \end{pmatrix}, e_1 \overset{c=0}{\rightarrow} \begin{pmatrix} AB \\ - \\ - \\ - \\ - \end{pmatrix}, e_2 \overset{c=1}{\rightarrow} \begin{pmatrix} - \\ AA \\ - \\ AB \\ - \end{pmatrix}, e_3 \overset{c=0}{\rightarrow} \begin{pmatrix} CCC \\ AA \\ - \\ AB \\ - \end{pmatrix}, e_4 \overset{c=0}{\rightarrow} \begin{pmatrix} C \\ AA \\ - \\ AB \\ - \end{pmatrix}, e_5 \overset{c=1}{\rightarrow}$$

$$\begin{pmatrix} - \\ - \\ C \\ AB \\ - \end{pmatrix}, e_6 \overset{c=1}{\rightarrow} \begin{pmatrix} CDC \\ - \\ - \\ AB \\ - \end{pmatrix}, e_7 \overset{c=0}{\rightarrow} \begin{pmatrix} - \\ - \\ - \\ AB \\ - \end{pmatrix}, e_8 \overset{c=3}{\rightarrow} \begin{pmatrix} AA \\ BB \\ - \\ - \\ - \end{pmatrix}, e_9 \overset{c=0}{\rightarrow} \begin{pmatrix} AA \\ - \\ - \\ - \\ - \end{pmatrix}, e_{10}$$

Table 3: Solution of the example

3.1 Network of (S, e_i)-pairs

The basic idea behind the dynamic programming algorithm is the following. From the initial state and the first event we can determine all possible responses which are compatible with the second event. In this way a set of new pairs (S, e_2) are created which are then treated recursively in the same way. For a formal description, let each pair (S, e_i) be a node and let each transition (set of shunt movements) leading to a following node be an arc. This way we get a network in which we can move from one pair (S, e_i) to an other pair (S', e_{i+1}). In this network we only allow valid pairs, and each transition has an associated cost equivalent to the number of shunt moves required for carrying out the transition. It is not difficult to see that the shunting problem is equivalent to finding a shortest path in this network.

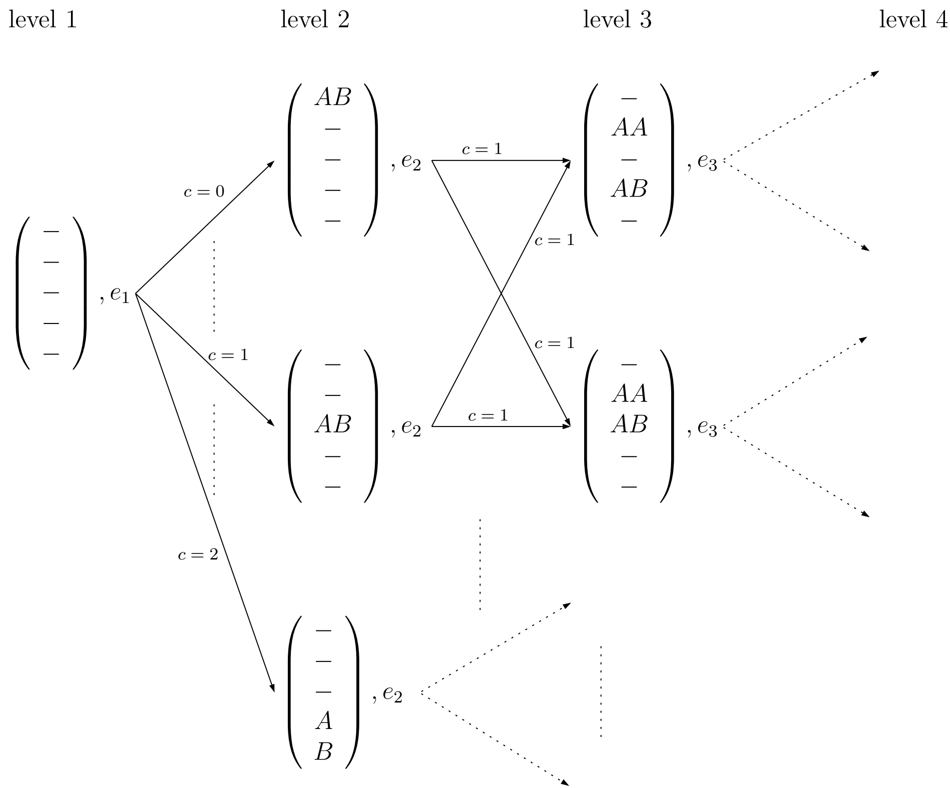

Figure 6: Dynamic programming network

Although the network becomes very large, the network is highly structured. The network consists of a number of levels, where each level corresponds to one event, see Figure 6. Hence, there are only arcs going from level i to level $i+1$. This means that the cost of getting to a particular state is given by the cost of the states in the previous level plus the cost associated with the arcs.

To obtain the optimal solution, we just have to construct the network level by level and calculate the cost of getting in each of its nodes. However, though this would work in theory, in practice the runtime of this algorithm may explode as the

instances get larger (remember, that the problem is NP-hard).

3.2 Eliminating Nodes

To make the dynamic programming approach work in practice we need to bring down the size of the dynamic programming network. In this section we present several suggestions to speed up the dynamic program algorithm. However, by applying some of these suggestions we can no longer guarantee that the optimal solution is found.

Eliminate symmetry Whenever there are two tracks with the same characteristics (same capacity and reachable with the same number of shunt movements from the platforms), there are many nodes in the network that are basically the same. In the example given earlier we have not used shunt track 3. If all the units assigned to track 2 are assigned to shunt track 2, we have a different solution which is essentially the same. So, in the network we can delete many states which are symmetric without affecting the solution.

Disallow costly transitions Given a transition with a high number of shunt movements, one might not want to allow this transition from a practical point of view. We can incorporate this, by simply deleting the arcs corresponding to these costly transitions from the network. This may reduce the number of outgoing arcs from nodes and may even lead to nodes which are not reachable anymore, reducing the number of nodes in the network. Note, that disallowing costly transitions may exclude the optimal solution.

Upper bounding the solution For each node in the dynamic programming network we know the cost of getting to this node. If by some (heuristic) procedure we know that there exists a solution with cost c, we do not have to proceed with nodes in the network that have cost exceeding c, i.e. these nodes can be deleted from the network. Reducing the dynamic programming network in this way does not affect the optimal solution.

Detecting bad paths Suppose we have created the dynamic programming network up to level i. If we now compare the cost of all nodes in level i, we may expect that the costly ones have only a small chance to result in the overall optimal solution. Deciding not to continue from the nodes with high costs reduces the dynamic programming network. However, this may exclude the optimal solution.

Rolling horizon To make a decision for level 1, we may restrict ourselves to creating the dynamic programming network only up to level i. Based on the information up to level i we may decide which arc to take leaving level 1. Starting with the resulting node on level 2, we now may create the network up to level $i + 1$ and use this network to decide upon the level 2, etc. This type of decision making is called *rolling horizon*. Each time we make a decision, only a small part of the network is considered. Again, we may exclude the optimal solution.

3.3 Computational Results

We have made a proof-of-concept implementation of the dynamic programming approach in C++, comprising about 1000 lines of code. The example of Section 1.2 is used to test both the implementation and some of the elimination rules. The results are summarized in Table 4.

In this table,

- **clm** indicates the maximum allowed cost between each level,
- **sym** indicates whether or not symmetry elimination is used,
- **ntp** indicates whether or not states, in which more than 2 types of train units are on the same shunt track, are forbidden,
- **tp** indicates whether or not states, in which unit of types A/B and C/D are mixed, are forbidden,
- **#states** gives the number of states on each level in the network. In most cases, only state counts up to level 4 are given, as the runtime increases dramatically after that,
- **runtime** gives the runtime for those computations that we ran to completion (the running times are after various optimizations of the code, on a 2.16GHz laptop),
- **cost** gives the resulting costs for those computations that we ran to completion.

The number of valid states does not tell the entire story, though. The number of *intermediate* states, i.e. those states that have to be computed and may or may not be valid, has a large impact on the runtime as well. In case I, each of the 128 states in level 2 generates about 25000 new states, of which in total only about 1500 are valid. This is quite a large number compared to e.g. case 4A, where the number 25000 is already reduced to about 3700.

The impact of limiting the costs between levels in the network is clear: If we do not enforce any limits, the network is simply too large to compute. If we limit to 4, we can complete the computation, but if we limit to 3 the speedup is almost a factor of 5 without losing the optimal solution. Limiting the costs of the arcs to 2 removes the optimal solution, but could provide a good heuristic for upper bounding the solution (see Section 3.2).

The other elimination rules also cut down the number of states significantly, although not as dramatically as limiting the costs of arcs.

One of the major advantages of this approach is that adding new rules (e.g. heuristics used by Dutch Railways planners) is extremely easy: in our implementation it is literally a matter of minutes. Furthermore, the chosen DP-approach is very suitable for parallelization.

id	Elimination rules				#states	run-time	cost
	clm	sym	ntp	tp			
I.	∞	-	-	-	$28 \rightarrow 128 \overset{\sim 25000}{\rightarrow} \sim 1500 \rightarrow \dots$		
4A.	4	-	-	-	$28 \rightarrow 128 \overset{\sim 3710}{\rightarrow} 1500 \rightarrow 180 \rightarrow \dots \rightarrow 14 \rightarrow 1$	737s	6
4B.	4	y	-	-	$19 \rightarrow 72 \overset{\sim 2410}{\rightarrow} 780 \rightarrow 108 \rightarrow \dots \rightarrow 10 \rightarrow 1$	280s	6
4C.	4	y	y	-	$19 \rightarrow 72 \overset{\sim 2410}{\rightarrow} 630 \rightarrow 90 \rightarrow \dots \rightarrow 10 \rightarrow 1$	242s	6
4D.	4	y	y	y	$19 \rightarrow 72 \overset{\sim 2410}{\rightarrow} 178 \rightarrow 40 \rightarrow \dots \rightarrow 10 \rightarrow 1$	153s	6
3A.	3	-	-	-	$28 \rightarrow 128 \overset{\sim 870}{\rightarrow} 1500 \rightarrow 180 \rightarrow \dots \rightarrow 14 \rightarrow 1$	146s	6
3B.	3	y	-	-	$19 \rightarrow 71 \overset{\sim 630}{\rightarrow} 776 \rightarrow 108 \rightarrow \dots \rightarrow 10 \rightarrow 1$	63s	6
3C.	3	y	y	-	$19 \rightarrow 71 \overset{\sim 630}{\rightarrow} 628 \rightarrow 90 \rightarrow \dots \rightarrow 10 \rightarrow 1$	54s	6
3D.	3	y	y	y	$19 \rightarrow 71 \overset{\sim 630}{\rightarrow} 178 \rightarrow 40 \rightarrow \dots \rightarrow 10 \rightarrow 1$	24s	6
2A.	2	-	-	-	$19 \rightarrow 121 \overset{\sim 140}{\rightarrow} 1196 \rightarrow 180 \rightarrow \dots \rightarrow 12 \rightarrow 1$	18s	7
2D.	2	y	y	y	$13 \rightarrow 64 \overset{\sim 120}{\rightarrow} 166 \rightarrow 40 \rightarrow \dots \rightarrow 8 \rightarrow 1$	3s	7

Table 4: Dynamic Programming Results

4 Alternative Approaches

Besides the presented the Greedy Algorithm and Dynamic Programming Algorithm, other solution approaches may be possible. In this section we give some comments on such approaches.

4.1 Local Search

One might expect that a local search approach is useful to obtain good solutions, since each solution is a list of shunt movements compatible with the event list. However, we feel that defining small local operations on this list which result in new compatible lists of shunt movements, is extremely difficult. When a small change is made in the movement list, many repair operations may be required to keep the list compatible with the event list. Consider the example given in Section 1.2, where between events e_2 and e_3 train AB is shunted to track 2. Suppose we modify this first shunt movement by moving AB to shunt track 1 instead of moving it to shunt track 2. This small change makes the remainder of the list incompatible with the events, i.e. the shunt movement between events e_8 and e_9 cannot be performed. This example shows that changing a single movement is not just a local change, it requires repair operations that can be much further down the list. Furthermore, it seems to be difficult to calculate the resulting change in the objective value in a simple way since we know nothing about the amount of repair operations. This convinces us that a local search approach may be not an easy way to go.

4.2 Integer linear programming

A possible approach is to extend the model from [7] by other shunt moves. For example, to include the possibility to wait at the platform and delay shunting, we need to include the 'shunting time' explicitly. The current model includes a variable z_{js} which equals 1 if train unit j is parked at or retrieved from tracks s. We could replace these variables by z_{jst} signaling if train unit j is parked at or retrieved from tracks s at time t. Another possibility is to add variables t_j representing the shunting time of train unit j. Although the number of reasonable shunting times for a train unit is limited, both options significantly complicate the model: the first by strongly increasing the number of variables and the second by the need for additional 'nasty' constraints. The computation time will probably increase accordingly.

A different LP-based approach is to apply *column generation*. In [4] a column generation algorithm for the planning of aircraft at gates or platform stands at Amsterdam Airport Schiphol is presented. Because of the similarity with the problem of planning train units on a shunt yard, i.e., shunt tracks correspond to gates at an airport, the idea seems useful to explore. The idea is that the problem is decomposed into two levels. At the highest 'master' level we have variables representing a complete shunting plan for one shunt track and the most important constraint is that the retrieval of each departing train unit and the parking of each arriving train unit is included in exactly one shunting plan. At the detailed or subproblem level we determine feasible shunt plans for one track which are expected to be beneficial for the optimization at the master level. Column generation approaches have been successful to solve large optimization problems in many different applications. However, certain shunt moves such as rearrangements of trains between different shunt tracks seem to be quite complicated to include in the model and therefore we are not convinced that it is worth to investigate this approach further.

5 Further research

In this paper we have presented two approaches for shunting train units. The first one is a greedy algorithm that can find a feasible shunt plan quickly. This algorithm typically chooses one single possibility that looks best at the current moment in time. The second one is a dynamic programming algorithm that can find the optimal shunt plan and typically explores a lot of possible states. We presented an outline and a basic version of the algorithms and developed a preliminary prototype of the dynamic programming algorithm.

Each of the algorithms can be improved by moving more towards the other approach. The greedy algorithm can be improved by including smart look-ahead rules and rules used by operational planners. The dynamic programming can be improved by rules to prune non-promising states and in this way make the set of states that have to be explored smaller. To have the best of both worlds, the two algorithms can also be combined. For example, a state within the dynamic program can be extended to a complete feasible solution by the greedy algorithm. This solution can then be used as an upper bound to prune non-promising states. Investigating these possible improvements is topic of future research.

References

[1] U. Blasum, M.R. Bussieck, W. Hochstättler, C. Moll, H.-H. Scheel, and T. Winter. Scheduling trams in the morning. *Mathematical Methods of Operations Research*, 49(1):137–148, 1999.

[2] S. Cornelsen and G. Di Stefano. Track assignment. *Journal of Discrete Algorithms*, 5(2):250–261, 2007.

[3] E. Dahlhaus, P. Horak, M. Miller, and J. F. Ryan. The train marshalling problem. *Discrete Applied Mathematics*, 103(1-3):41–54, 2000.

[4] G. Diepen, J.M. van den Akker, J.A. Hoogeveen, and J.W. Smeltink. Using column generation for gate planning at amsterdam airport schiphol. *Technical report UU-CS-2007-018, Department of Information and Computing Sciences, Utrecht University, submitted to Transportation Science*, 2007.

[5] A. Goldstein, P. Kolman, and J. Zheng. Minimum common string partition problem: Hardness and approximations. In *ISAAC*, pages 484–495, 2004.

[6] R. Jacob, P. Marton, J. Maue, and M. Nunkesser. Multistage methods for freight train classification. In *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, 2007.

[7] L.G. Kroon, R.M. Lentink, and A. Schrijver. Shunting of passenger train unit: an integrated approach. *Erasmus University Rotterdam, ERIM Report Series 2006-12-20*, 2006.

[8] R.M. Lentink. *Algorithmic Decision Support for Shunt Planning*. PhD thesis, Erasmus University Rotterdam, The Netherlands, 2006.