Robust Algorithms and Price of Robustness in Shunting Problems*

Serafino Cicerone[1], Gianlorenzo D'Angelo[1], Gabriele Di Stefano[1], Daniele Frigioni[1], and Alfredo Navarra[1,2]

[1] Dipartimento di Ingegneria Elettrica e dell'Informazione, Università dell'Aquila, Poggio di Roio, 67040 L'Aquila Italy. Emails: {cicerone,gdangelo,gabriele,frigioni}@ing.univaq.it

[2] Dipartimento di Matematica e Informatica, Università di Perugia, Via Vanvitelli 1, 06123 Perugia, Italy. Email: navarra@dipmat.unipg.it

Abstract. In this paper we provide efficient robust algorithms for shunting problems concerning the reordering of train cars over a hump. In particular, we study algorithms able to cope with *small disruptions*, as temporary and local unavailability and/or malfunctioning of key resources that can occur and affect planned operations. To this aim, a definition of *robust algorithm* is provided. Performances of the proposed algorithms are measured by the notion of *price of robustness*. Various scenarios are considered, and interesting results are presented.

Keywords: Shunting; Hump Yard; Disruption; Robustness; Recoverability; Robust Algorithm

1 Introduction

Optimization of railways involves many planning and scheduling activities spanning several time horizons. In this paper, among short term planning phases, we consider the *shunting problem*, that is the scheduling of activities at a shunting yard in depots or stations.

In railroad shunting yards, incoming freight trains are split up and rearranged according to their destinations. In stations and train depots, passenger trains are parked overnight or during low traffic hours. In either case we are given an ordering of arriving units, i.e., either cars, or trains or train units, and we have to decide how to use the tracks of the shunting yard to reorder the units according to a required departure sequence. Possible scheduling activities are limited by the fixed number of available tracks, by their length and by the way tracks may be approached. Many results have been reached in literature on shunting problems by assuming a perfect knowledge of the incoming and outcoming sequences of units (e.g., [4–6, 8, 10, 11]).

* This work was partially supported by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

On the other hand, a recent approach looks at the shunting problem as an online problem: since the trains could accumulate lateness before arriving at the depot, the time of arrival of each train could be unpredictable. The tracks must thus be assigned online, as the trains arrive, on the basis of departure times and previous assignments [13, 7].

These two approaches lack in reality, since *small disruptions*, concerning temporary and local unavailability and/or malfunctioning of key resources, can occur and then affect, e.g., the planned incoming unit sequence, but it is also unlikely that we have no idea about the order of the sequence, as in the online approach. What we need is a *robust solution* to the shunting problem that maintains feasibility by applying available recovery capabilities in the case of disruptions. This avoids both a recalculation from scratch of a new schedule and a complete online approach to the problem.

What is *robustness* for an optimization problem? Several attempts have been tried in order to provide a formal definition which is able to capture many different peculiarities (see for instance [1, 3, 9]). Recently, a special issue on robust optimization has been published in the central publication forum of the mathematical programming society [2].

However, the notion of robustness in every day life is much broader than that pursued in so-called robust optimization so far. In the most restricted sense, a robust plan stays unchanged in every likely scenario. The basic idea of robustness is given by a problem and some knowledge imperfection with which one has to cope. That is, the solution provided for a given instance of the problem must hold even though some changes in such an instance occur. This kind of robustness is not always suitable if some recovery strategies are not introduced. Moreover, in many practical applications, there might be the possibility to intervene before some scheduled operations are being performed. This suggests to study robustness with respect to available recovery capabilities. Usually, modifications that may occur are restricted to some specified subset of all possible ones. It is reasonable to require that if a disruption occurs, one would like to maintain as much as possible a pre-computed solution taking into account some "soft" recovery strategies. Recoverability should be simple and fast. Moreover there are cases where recoverability is necessary in order to still have some useful solution for a problem. A solution that undergoes slight changes is called robust even though it could require the use of some recovery capabilities.

In this paper we provide a definition for *robust algorithms* and a definition for the corresponding *price of robustness*. We follow directions given in [12], and emphasize algorithmic aspects. The purpose/hope is to capture useful properties that help to overcome the standard notion of robustness. Intuitively, given an optimization problem P, a set of possible disruptions, and a set of available recovery strategies $\mathcal{A}_{rec}$, we define the corresponding robustness problem R_P. An instance i of P becomes a set $M(i)$ of instances obtained by applying any possible disruption to i. A robust algorithm A_{rob} takes i as input and outputs a feasible solution for any instance in $M(i)$ with the chance to apply available recovery strategies. In other words, given an instance i of P and a disruption

$j \in M(i)$, a solution s for i provided by A_{rob} can be turned into a feasible solution for j by applying some recovery strategies allowed by $\mathcal{A}_{rec}$. Solution s is then called a robust solution. Clearly, robust solutions provided by A_{rob} can be far from the optimum. Such a distance is measured by the price of robustness. In [12] the aim is to provide the best robust solution, i.e., the one that minimize the price of robustness. We are interested in finding efficient robust algorithms, and evaluating them by comparing the corresponding prices of robustness.

We apply these definitions in a practical context given by shunting problems introduced in [11]. In a shunting plan, disruptions are given by different orders of the incoming trains/cars, new trains/cars, missing trains/cars, or faulty infrastructures like tracks. We provide robust shunting plans able to cope with bounded number of disruptions. We also study various levels of robustness according to different recovery capabilities.

The paper is organized as follows: Section 2 introduces the shunting problem in a hump yard as given in [11]. Section 3 introduces a model concerning robustness for optimization problems. Section 4 gives a robust interpretation to shunting problems arising in practical context, and for each problem we provide robust algorithms and evaluate their price of robustness. Finally, Section 5 gives some conclusive remarks and discusses some open problems.

2 Shunting Over a Hump

In this section we introduce the shunting problem in a hump yard as given in [11]. The problem is specified by an input train T_{in} composed of n cars and an output train T_{out} given by a permutation of T_{in} cars. Each car is assigned with a unique label. The considered hump yard appears as in Figure 1.

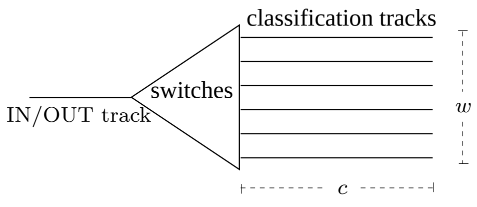

Fig. 1. Hump yard infrastructure composed of w classification tracks, each of size c.

There is an input track where trains arrive and a set of switches by which cars composing the incoming train can be shunted over the available classification tracks. A classification track is approached from a single side and works like a stack. The number of available classification tracks is denoted by w, and their size, i.e., the number of cars that can fit into a classification track, by c. This layout supports a sorting operation by repeatedly doing the so called track pull (operation) which is made up of:

- Connect the cars of one classification track into a pseudotrain;
- Pull the pseudotrain over the hump;
- Disconnect the cars in the pseudotrain;
- Push the pseudotrain slowly over the hump, yielding single cars that run down the hill from the hump towards the classification tracks;
- Control the switches such that every single car goes to a specified track.

The goal is to reorder T_{in} according to T_{out} by repeatedly performing the track pull operation (an example of reordering by means of track pulls can be seen in Figure 2). The cost of the reordering is measured by the number of track pulls. Clearly, at least one pull must be performed.

We consider three different variants of the shunting over a hump problem by specifying constraints for c and w. Namely,

Case 1- c bounded, w unbounded;
Case 2- c unbounded, w bounded;
Case 3- c and w unbounded.

In [11] a polynomial algorithm for each case is given. In particular, a 2-approximation algorithm for Case 1-, and optimal algorithms for Case 2- and Case 3-, are provided.

It is worth mentioning a further algorithm presented in [11] that solves the shunting problem when c is bounded and the input train is unknown in advance. Equivalently this can be seen as the order of the cars in T_{in} is the reverse of the order in T_{out}. The proposed solution provides a set of different operations for each car. In the remainder of the paper we refer to such an algorithm as A_{out}.

Before concluding this section we need to describe how the set of track pulls operations is specified and represented in [11] since we make use of the same notation. In general, a shunting plan has to specify a sequence of track pull operations, given by the track whose cars are pulled, and for every car which track it is sent to. Tracks are named according to the time they are pulled, i.e., $T = \{1, \dots, h\}$. This means that one physical track might get several such names (numbers) if it is pulled several times during the shunting plan. In such situations, the logical track is annotated by the name of a physical one. Of course, if there is no limit on the number of tracks ($w \geq h$), there is no need to reuse a track, and this annotation by names of physical tracks is not necessary. With this numbering of the tracks, the itinerary of a car can be described by the sequence of logical tracks it visits. For the task at hand, it is convenient to specify this sequence as a bitstring or code $b_1 \cdots b_h$ where the different bits stand for the logical tracks, and there is a 1 if and only if the car visits that track. Now, if track i is pulled, the new destination of a car is given by the position of its next 1 in its code, i.e., the lowest index $j > i$ with $b_j = 1$.

A shunting plan must specify a track pulls sequence T and it has to associate a code to each car. Codes length is determined by the length of T and cars may share the same code.

According to the previous notation, A_{out} provides n different bitstrings, one per car. Each string specifies the route that the corresponding car has to perform

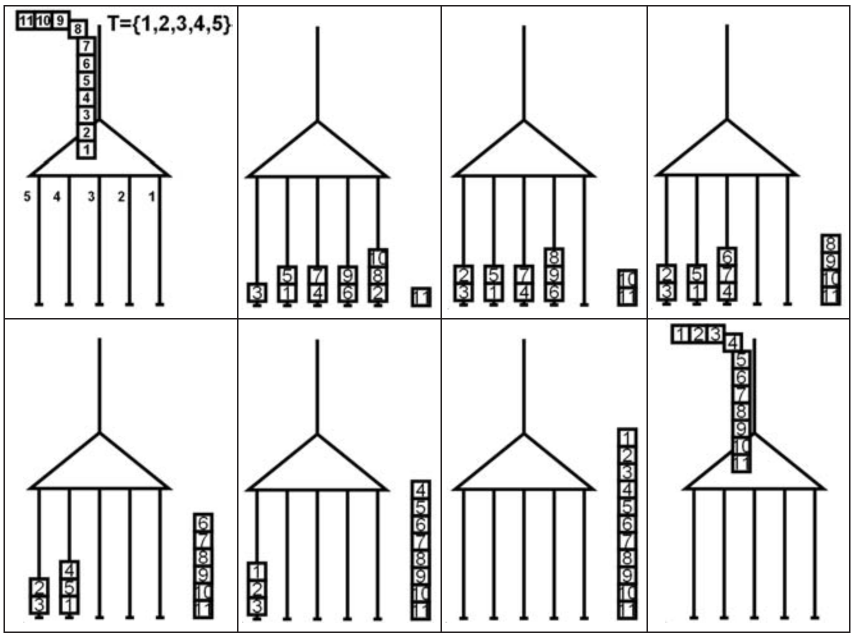

Fig. 2. Example of a shunting plan given by A_{out} when $c = 3$ and the number of track pulls is set to 5. Cars from 11 down to 1 are associated with codes 00000, 00001, 00010, 00011, 00100, 00110, 01000, 01100, 10000, 10001, 11000 respectively. The track where T_{out} is composed is not shown.

among the shunting yard in order to be placed in the desired position according to T_{out}. Moreover such an algorithm is optimal with respect to the minimum number of track pulls. For the sake of simplicity, it is assumed that T_{out} is composed on a track not used for shunting operations but that can contain the full train. A running example of A_{out} is shown in Figure 2. The sequence of track pulls is given by $T = \{1, 2, 3, 4, 5\}$ from right to left among classification tracks. In the example $c = 3$ and the number of track pulls is set to 5. The set of codes of length 5 provided by a feasible solution is such that at each position at most three codes have the corresponding bit set to 1. This implements the constraint on c and implies that at most eleven different codes can be generated. Cars from 11 down to 1 are associated with codes 00000, 00001, 00010, 00011, 00100, 00110, 01000, 01100, 10000, 10001, 11000 respectively. Figure 2 shows the subsequent configurations obtained after each track pull and reorder of the pulled cars according to their codes.

Note that, when T_{in} is known, two cars might be assigned with the same code. This would imply that they will have the same order in T_{out} as in T_{in}. Two cars that are consecutive in T_{out} can get the same code if they are in the correct order in T_{in}. A maximal set of cars in T_{out} that has this property is called a *run*.

Definition 1. *In a shunting plan, for each code x, a* pure run *is the maximal set of cars associated with x.*

Let $opt(k, c, w) \geq 1$ be the number of track pulls needed by an optimal shunting plan in order to manage k cars/runs with tracks size c and w tracks (in cases 1- and 3-, $w = \infty$; in cases 2- and 3-, $c = \infty$). Let $apx(k, c, w)$ be the best known approximation algorithm for the corresponding shunting problem, and let *apxr* be its approximation ratio. Whenever clear by the context we skip parameters equal to ∞ from previous notation.

3 Robustness

In this section, in the spirit of [12], we introduce a model concerning robustness for optimization problems. In particular, given an arbitrary optimization problem P, we first show how to turn P into a *robustness problem* R_P. Then, we define which feasible solutions for P solve R_P, that is, we formally define the notion of *robust solutions*. Finally, we define the concept of *robust algorithm* for R_P.

Moreover, we quantify the *price of robustness* of a robust algorithm. As usual, by using the theoretical best robust algorithm for R_P, we define the price of robustness of the problem R_P.

Without loss of generality, we always consider minimization problems. In the remainder, a minimization problem P is always characterized by the following parameters.

- I, the set of instances of P;
- F, the function that associate to any instance $i \in I$ the set of all feasible solutions for i;
- $f\colon S \to \mathbb{R}$ the objective function of P, where $S = \bigcup_{i \in I} F(i)$.

Based on a minimization problem P, we can define a robustness problem R_P as it follows.

Definition 2. *A* robustness problem *R_P is given by the triple $(P, M, \mathcal{A}_{rec})$, where:*

- *P is an optimization problem;*
- *$M : I \to 2^I$ is a modification function for instances of P;*
- *$\mathcal{A}_{rec}$ is a class of* recovery algorithms *for P. Each element of $\mathcal{A}_{rec}$ takes as input a triple $(i, s, j) \in I \times S \times I$ and outputs a solution $s' \in S$.*

Given an instance $i \in I$ for P, an element $s \in F(i)$ is a robust solution *for i with respect to R_P if and only if the following relationship holds:*

$$\exists A \in \mathcal{A}_{rec} : \forall j \in M(i),\ A(i, s, j) \in F(j).$$

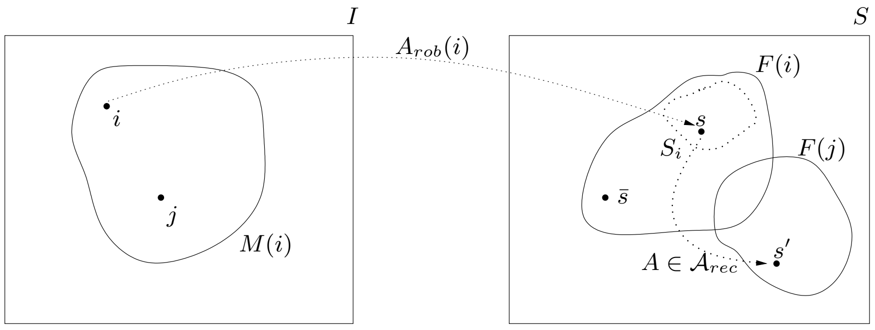

Fig. 3. Robustness problem: I, set of instances; S, set of solutions; $M(i)$, set of instances obtainable after a disruption; $F(i)$, set of feasible solutions for i; S_i, set of recoverable solutions; $\bar{s}$, optimal solution for i; s, robust solution obtained by A_{rob}; s', recovered solution obtained by an algorithm $A \in \mathcal{A}_{rec}$.

Let us explain the rationale underlying this definition. Given $i \in I$, $M(i)$ represents all the instances for P that can be obtained by applying all possible modifications to i. Such modifications model *disruptions* that can arise with respect to the current input for P. Algorithms in $\mathcal{A}_{rec}$ represent the capability of recovering against possible disruptions. An input triple $(i, s, j) \in I \times S \times I$ for every $A \in \mathcal{A}_{rec}$ is made of the input instance i for the original optimization problem P, a feasible solution s for i, and a possible disruption j for i, i.e., a modification of i. If $j \in M(i)$, and s is a robust solution, then there must exists an algorithm $A \in \mathcal{A}_{rec}$ such that starting from s it obtains a new solution $s' \in F(j)$. A possible scenario for this situation is depicted in Fig. 3, where S_i represents the subset of feasible solutions for i that can be recovered by an algorithm $A \in \mathcal{A}_{rec}$ when a disruption $j \in M(i)$ occurs.

A *robust algorithm* is any algorithm that computes robust solutions for R_P.

Definition 3. *Given $R_P = (P, M, \mathcal{A}_{rec})$, a* robust algorithm *for R_P is any algorithm A_{rob} such that $\forall i \in I, A_{rob}(i)$ is robust with respect to R_P.*

It is worth to mention that, if a robustness problem $R_P = (P, M, \mathcal{A}_{rec})$ is based on a single recovery algorithm A, $\mathcal{A}_{rec} \equiv \{A\}$, that fulfills the following condition:

$$\forall (i, s) \in I \times S,\ \forall j \in M(i),\ A(i, s, j) = s$$

then R_P represents the so called *strict robustness problem.* Note that, in this case, a robust algorithm A_{rob} for R_P must provide a solution s for i such that s is feasible for each possible modification $j \in M(i)$. This means that, since A has no capability of recovering against possible disruptions, then A_{rob} has to find solutions that "absorb" *any* possible disruption.

Now, let us consider again Fig. 3. Note that, if $\bar{s}$ denotes the optimal solution for P when the input instance is i, it is possible that $\bar{s}$ is not in S_i; this implies that every robust solution for i may be "very far" from $\bar{s}$. A "good" robust algorithm should find the best solution in S_i for P, for each possible input $i \in I$. The goodness of a robust algorithm is measured by the concept of *price of robustness* as in the following definition.

Definition 4. *The* Price of Robustness *of a robust algorithm A_{rob} for a robustness problem R_P is given by*

$$PoR(R_P, A_{rob}) = \max_{i \in I} \left\{ \frac{f(A_{rob}(i))}{\min\{f(x) : x \in F(i)\}} \right\}.$$

For every instance i, the price of robustness of A_{rob} is given by the maximum ratio between the cost of the solution provided by A_{rob} and the optimal solution. The price of robustness of R_P is given by the minimum price of robustness among all possible robust algorithms. Formally,

Definition 5. *The* Price of Robustness *of a robustness problem R_P is given by*

$$PoR(R_P) = \min\{PoR(R_P, A_{rob}) : A_{rob} \textit{ is a robust algorithm for } R_P\}.$$

Definition 6. *A robust algorithm A_{rob} is* exact *for a robustness problem R_P if $PoR(R_P, A_{rob}) = 1$.*

Definition 7. *A robust algorithm A_{rob} is* optimal *for a robustness problem R_P if $PoR(R_P, A_{rob}) = PoR(R_P)$.*

In the remainder, by "optimal" we may refer either to an optimization problem in the standard meaning or to a robustness problem in the meaning of Definition 7. Which definition must be applied will be clear by the problem we are referring to, if it is either an optimization problem or a robustness problem respectively.

4 Disruptions and Recoverability

In this section we evaluate the price of robustness defined in Section 3 in practical contexts arising from the shunting problems described in Section 2. In the following P is one of the three shunting optimization problems defined in Section 2. For Case 1-, for instance, P is defined by

– f : number of track pulls;
– I : pair (T_{in}, T_{out}) where train T_{in} is defined as a sequence of cars and train T_{out} is a permutation of T_{in} cars;
– $F(i)$: set of all feasible solutions for a given pair $i \equiv (T_{in}, T_{out}) \in I$, i.e. any sequence of track pulls combined with a set of codes (one per car) that transform T_{in} in T_{out} when c is bounded.

Sections 4.1 and 4.2 are devoted to two different modification function M respectively. Concerning classes of recovery algorithms we consider the following three possibilities.

$\mathcal{A}^1_{rec}$: $\forall A \in \mathcal{A}^1_{rec}$, $\forall (i,s) \in I \times S$, $\forall j \in M(i)$, $A(i,s,j) = s$, i.e., there are no recovery strategies to apply (strict robustness);

$\mathcal{A}^2_{rec}$: $\forall A \in \mathcal{A}^2_{rec}$, $\forall (i,s) \in I \times S$, $\forall j \in M(i)$, $A(i,s,j) = s'$, where s' may differ from s by at most one code, i.e., at most one pure run may be assigned with a new code of the same length;

$\mathcal{A}^3_{rec}$: $\forall A \in \mathcal{A}^3_{rec}$, $\forall (i,s) \in I \times S$, $\forall j \in M(i)$, $A(i,s,j) = s'$, where s' may differ from s by all the set of codes, i.e., every pure run may be assigned with a new code of the same length.

The three different classes of recovery algorithms imply three different robustness problems R_P for each shunting problem P. On the other hand, by definition, every upper bound to the price of robustness of each shunting problem with $\mathcal{A}^1_{rec}$ holds for $\mathcal{A}^2_{rec}$ as well as every upper bound obtained with $\mathcal{A}^2_{rec}$ holds for $\mathcal{A}^3_{rec}$. Moreover, every lower bound obtained with $\mathcal{A}^3_{rec}$ holds for $\mathcal{A}^2_{rec}$ as well as every lower bound obtained with $\mathcal{A}^2_{rec}$ holds for $\mathcal{A}^1_{rec}$.

Note that each of the three defined classes of recovery algorithms can not change/extend the scheduled track pulls sequence defined by a shunting algorithm A_{rob}. This is motivated by the fact that the cost of a shunting plan is assumed to be proportional to the number of track pulls (see Section 2). Recovery capabilities, instead, should be cheap operations since they can not be planned a priori but are used at run time.

In what follows, for every instance $i = (T_{in}, T_{out})$ we denote by r_i and n_i the number of runs and cars respectively in T_{in}.

4.1 One Car With Unexpected Incoming Position

Given an instance $i = (T_{in}, T_{out})$ of the shunting optimization problem P, let $M(i)$ represent all possible instances (T'_{in}, T_{out}) obtainable from i by changing the order of just one car in T_{in}. For each of the three cases of Section 2 we study feasibility of robust shunting plans for the three different classes of recovery algorithms defined above.

Before approaching every possible case, the following lemma describes which practical situation a robust plan must be able to absorb/recover with respect to a car incoming with an unexpected position. In detail, the lemma shows that if a car arrives at a position different than expected, then at most one additional pure run with respect to the original situation is needed.

Lemma 1. *Let v be a car arriving at the hump in a different position than expected. At most one additional pure run must be managed with respect to the expected case.*

Proof. If v composed a pure run itself, every shunting plan is robust since the same code assigned to v is valid also in the actual case. The same holds in all cases where the change in the incoming position of v does not affect its relative position with respect to the pure run it belongs to.

If v was the first (last, resp.) car of its original run, and it arrives after (before, resp.) some cars of that run then it becomes itself a pure run unless it can be joint with some other pure runs. All the other cars of its original pure run still compose a pure run since their relative order did not change.

If v was part (in the middle) of a pure run then v may arrive either before its original pure run (*case a*), or in the middle but before its expected placement (*case b*), or in the middle but after its expected placement (*case c*), or after its original pure run (*case d*). If *case a* occurs, then v with all cars of the original pure run after the expected position of v still compose a run but the remaining part of the original pure run can not be assigned with the same code. If *case b* occurs, then same arguments of *case a* still hold. If *case c* occurs then the first part of the original pure run until the expected position of v plus v compose a pure run, while the remaining cars must be another pure run. If *case d* occurs, then same arguments of *case c* still hold. Summarizing, in all cases at most one additional pure run is created. $\square$

In a shunting plan, Lemma 1 is reflected in the need of at most one additional code.

Lemma 2. *For every input train T_{in} and considering $\mathcal{A}^1_{rec}$, any robust shunting algorithm A_{rob} must provide a unique code to each car of T_{in}.*

Proof. Assume by contradiction that two cars v and w have the same code in A_{rob}. Without loss of generality, let v being expected before w in T_{in}. This means that v should appear before w also in the outgoing train. A_{rob} is assumed to be robust for any possible change of one car position. Let us consider the disruption where w precedes v in T_{in}. Since A_{rob} associates the same code to v and w, then w will appear before v also in the outgoing train. This contradicts the hypothesis that A_{rob} is a robust shunting algorithm with respect to any change in the position of one car. $\square$

Case 1-. As mentioned in Section 2, the solution proposed in [11] provides a 2-approximation of the optimum, i.e., $apxr = 2$. However, such a solution can not be used for robustness purposes when considering $\mathcal{A}^1_{rec}$ since it does not fulfil condition of Lemma 2. On the other hand, A_{out} turns out to be optimal (in the meaning of Definition 7).

Theorem 1. *Considering $\mathcal{A}^1_{rec}$, there exists an optimal robust shunting algorithm A_{rob} such that $PoR(R_P, A_{rob}) = \max_{i \in I} \frac{opt(n_i,c)}{opt(r_i,c)}$.*

Proof. We make use of A_{out} described in Section 2, i.e., we have one different code for each car without considering runs. Such a solution is clearly feasible for any change in the cars order since it is completely independent on the incoming order. From Lemma 2, $PoR(R_P) \geq \max_{i \in I} \frac{opt(n_i,c)}{opt(r_i,c)}$. Moreover, from [11], the solution provided by A_{out} is optimal in Case 1- when one unique code per car must be assigned. $\square$

Even though A_{out} is optimal for $\mathcal{A}^1_{rec}$, i.e., $PoR(R_P, A_{rob}) = PoR(R_P)$, it is not exact since in general $opt(n, c) \geq opt(r, c)$.

It is worth noting that the number of codes provided by the shunting algorithm A_{rob} of Theorem 1 is at most c times the number of codes provided by the optimal solution. In fact, we are in the case of tracks of bounded size c, and hence there cannot be more than c cars associated with the same code. This implies that if a run is composed by more than c cars, it must be split into more classification tracks.

Theorem 2. *Considering $\mathcal{A}^2_{rec}$, there exists a polynomial robust shunting algorithm A_{rob} such that $PoR(R_P, A_{rob}) = \max\limits_{i\in I} \frac{apx(r_i,c)+1}{opt(r_i,c)} \le 2 + \max\limits_{i\in I} \frac{1}{opt(r_i,c)} = 3$.*

Proof. By Lemma 1, the change in the order of one car may produce at most one additional pure run, hence at most one additional code is necessary to cope with such occurrence. By the solution proposed in [11] for Case 1-, the need of one additional code might imply the need of one additional track pull since it might be that codes of the original solution are already the maximum number available to manage r_i runs. However we are under Case 1- assumptions, i.e., unbounded number of tracks. This implies that A_{rob} must provide one additional track pull. This can be obtained by calculating codes as in [11] for Case 1- and then adding one bit (initially set to zero) corresponding to the new pull. In order to conclude the proof we need to show that the modification of at most one code as defined by $\mathcal{A}^2_{rec}$ is enough in order to make the solution provided by A_{rob} feasible with respect to M.

Let v be the car implementing disruption M. From Lemma 1, the actual situation is given by at most two pure runs instead of the pure run to which v belonged. Without loss of generality, let the actual pure run containing v be the one that composed the bottom part of the expected original pure run. Then an algorithm in $\mathcal{A}^2_{rec}$ simply assigns the same code as planned by A_{rob} to v and its actual pure run, and the same code but with the first bit set to one to the top part of the expected original pure run.

By construction, in the first pulled track there is only part of the original pure run to which v was expected to belong. This implies that the number of cars composing such a new run is less than c, otherwise they could not have been associated with the same code by A_{rob}. Once the first pull has been performed, the pulled run will be placed on top[3] of the second part of the pure run composing the expected pure run containing v, since their codes differ by just the first bit. Hence the expected pure run is now built and the shunting plan continues as was originally scheduled by A_{rob}. $\square$

As already said, every upper bound for $\mathcal{A}^2_{rec}$ holds for $\mathcal{A}^3_{rec}$. Up to now no better upper bound for $\mathcal{A}^3_{rec}$ has been found than that of $\mathcal{A}^2_{rec}$.

Cases 2- and 3-. When considering $\mathcal{A}^1_{rec}$, similar arguments of Theorem 1 can be applied, and the following corollary holds.

[3] Clearly there can be other cars in the middle but this does not influence the solution since codes exactly determine the outgoing order of the cars.

Corollary 1. *In Case 2- (Case 3- resp.), and considering $\mathcal{A}^1_{rec}$, there exists an optimal robust shunting algorithm A_{rob} such that $PoR(R_P, A_{rob}) = \max_{i \in I} \frac{opt(n_i,w)}{opt(r_i,w)}$ ($PoR(R_P, A_{rob}) = \max_{i \in I} \frac{opt(n_i)}{opt(r_i)}$ resp.).*

When considering $\mathcal{A}^2_{rec}$, in both Case 2- and Case 3-, for non-trivial plans we do not need to use one additional track since any track is big enough to contain the whole train. Hence, there is always enough space to wait for the missing car/run. The only exceptions arise when the number of track pulls required by the optimal shunting plan is too small in order to restore the expected car positions. For instance, this happens when $T_{in} \equiv T_{out}$. By applying similar arguments of Theorem 2, we can show the following theorem.

Theorem 3. *In Case 2- (Case 3-, resp.), considering $\mathcal{A}^2_{rec}$, there exists a polynomial robust shunting algorithm A_{rob} such that $PoR(R_P, A_{rob}) = \max_{i \in I} \frac{opt(r_i,w)+1}{opt(r_i,w)} = 1 + \max_{i \in I} \frac{1}{opt(r_i,w)} = 2$ ($PoR(R_P, A_{rob}) \leq 1 + \max_{i \in I} \frac{1}{opt(r_i)} = 2$, resp.).*

Concerning the price of robustness of the problem, the following theorem holds.

Theorem 4. *In Cases 1-, 2- and 3-, and considering $\mathcal{A}^2_{rec}$, $PoR(R_P) \geq 2$.*

Proof. As we have already remarked, by Lemma 1 the change in the order of one car might imply the need of one additional code which in turn implies the need of one additional track pull. Such a pull must be planned a priori by A_{rob} since every algorithm in $\mathcal{A}^2_{rec}$, by definition, affects only codes. This implies $PoR(R_P) \geq 1 + \max_{i \in I} \frac{1}{opt(r_i,c)} = 2$ for Case 1-, $PoR(R_P) \geq 1 + \max_{i \in I} \frac{1}{opt(r_i,w)} = 2$ for Case 2- and $PoR(R_P) \geq 1 + \max_{i \in I} \frac{1}{opt(r_i)} = 2$ for Case 3-. □

By Theorems 3 and 4, the following corollary can be stated.

Corollary 2. *There exists a robust algorithm in Case 2- (and one in Case 3-) that is optimal when considering $\mathcal{A}^2_{rec}$.*

4.2 One New Car

Another possible modification M is given by the arrival of one unexpected car v that was not scheduled in the original train but has to be consider in the actual shunting.

In all **Cases 1-, 2-, 3-**, v should be assigned, in general, with a new code. Again this might reflect the need of one further track pull.

Theorem 5. *If we consider $\mathcal{A}^1_{rec}$, no robust shunting algorithm exists.*

Proof. In order to have a robust shunting plan with $\mathcal{A}^1_{rec}$, v should be assigned a priori by A_{rob} with a code independent of its outgoing placement. On the other hand, each code exactly determines the outgoing position of the corresponding car with respect to all other cars, and the claim holds. □

However, if we use $\mathcal{A}^2_{rec}$ or $\mathcal{A}^3_{rec}$ it is possible to find a robust shunting plan. In particular, according to the incoming position of v, it might be enough to assign with it the same code of some already existent pure run. If v has to be placed at the end of the outgoing train, it may also happen that there are some spare codes available and the problem is easily solvable. If no codes are available (this happens if the size of the codes is already minimized according to the number of cars) or the incoming position of v does not allow the merge with an existent pure run, then we need some recovery strategy. Again, the strategy must be as less "invasive" as possible.

Theorem 6. *In Case 1-, considering $\mathcal{A}^2_{rec}$, there exists a polynomial robust shunting algorithm A_{rob} such that $PoR(R_P, A_{rob}) = \max_{i\in I} \frac{opt(n_i+1,c-1)+1}{opt(r_i,c)}$*

Proof. A possible solution A_{rob} is to use A_{out} (that assigns one different code for each car) by considering tracks of size $c-1$ instead of c and considering code 0 assigned to the new possible car. Clearly, decreasing tracks size and preserving code 0 from being used, implies an increase of needed track pulls. Moreover we add one further bit, initially set to zero, in the rightmost position of each code. In this way there are no consecutive integers represented by the provided set of codes. This implies that wherever a new car should be considered there is always an available code to which an algorithm in $\mathcal{A}^2_{rec}$ can change code 0. Moreover c constraint is preserved by having considered $c-1$ instead of c. □

In order to better understand the intuition behind proof of Theorem 6, we make use of an example. Assume we have tracks of size $c-1=3$ and we consider 5 tracks, then the available codes (as in the example of Figure 2) are: 00000, 00001, 00010, 00011, 00100, 00110, 01000, 01100, 10000, 10001, 11000 that must be assigned to the unexpected car and to cars from 10 to 1 respectively. If the new car must be inserted, for instance, between cars 2 and 1 we have many available codes (namely, 10010, 10011, 10100, 10101, 10110, 10111). An algorithm in $\mathcal{A}^2_{rec}$ could change, for instance, 00000 in 10100. Contrary, if we need to insert the new car between 10 and 9, then we do not have available codes since there is nothing in between 00001 and 00010. The new car may get code 00001 if it arrives after car 10 or code 00010 if it arrives before car 10 and car 9. If the new car arrives before car 10 but after car 9 then we get in trouble since there is no way to insert it between 9 and 10 without changing other codes. In order to cope with this case we can consider a different set of codes in which we do not allow to have two codes representing two consecutive integers. The new set of codes will be given by 000000, 000010, 000100, 000110, 001000, 001100, 010000, 011000, 100000, 100010, 110000. Now we have available codes in between any pair.

Theorem 7. *In Case 2- (Case 3-, resp.), and considering $\mathcal{A}^2_{rec}$, there exists a polynomial robust shunting algorithm A_{rob} such that $PoR(R_P, A_{rob}) = \max_{i\in I} \frac{opt(n_i+1,w)+1}{opt(r_i,w)}$ ($PoR(R_P, A_{rob}) = \max_{i\in I} \frac{opt(n_i+1)+1}{opt(r_i)}$, resp.).*

Proof. In Case 2-, similarly to proof of Theorem 6, we preliminarily assign code 0 to the new car and we use one different code for each car. All codes will be again not consecutive with respect to their integer representation by scheduling one additional initial track pull. In doing so, between two codes provided by A_{rob} there is always a code available to which code 0 can be changed by an algorithm in $\mathcal{A}^2_{rec}$. The claim then follows by observing that the proposed algorithm in [11] for Case 2- is optimal. Similar arguments hold for Case 3-. □

Lemma 3. *Considering $\mathcal{A}^2_{rec}$, any robust shunting algorithm A_{rob} must provide one different code for each car.*

Proof. Assume by contradiction that two cars v and w have the same code in A_{rob}. A_{rob} is assumed to be robust for a new car to be inserted in any position. Let z be an unexpected new car that must be inserted between v and w. Without loss of generality, let the code a priori associated with z by A_{rob} be inappropriate for the desired positioning of z. It is easy to verify that, in general, the insertion of z in between v and w requires a different code for z and for either v or w. Contrary, $\mathcal{A}^2_{rec}$ allows to change at most one code, hence the claim holds. □

The following corollary is a direct consequence of Lemma 3.

Corollary 3. *In Case 1- (2- and 3-, resp.), and considering $\mathcal{A}^2_{rec}$, $PoR(R_P, A_{rob}) \geq \max_{i \in I} \frac{opt(n_i+1,c)}{opt(r_i,c)}$ $(PoR(R_P, A_{rob}) \geq \max_{i \in I} \frac{opt(n_i+1,w)}{opt(r_i,w)}$ and $PoR(R_P, A_{rob}) \geq \max_{i \in I} \frac{opt(n_i+1)}{opt(r_i)}$, resp.).*

Theorem 8. *In Case 1- (2- and 3-, resp.), and considering $\mathcal{A}^3_{rec}$, there exists a polynomial robust shunting algorithm A_{rob} such that $PoR(R_P, A_{rob}) = \max_{i \in I} \frac{apx(r_i+1,c)}{opt(r_i,c)}$ $(PoR(R_P, A_{rob}) = \max_{i \in I} \frac{opt(r_i+1,w)}{opt(r_i,w)}$ and $PoR(R_P, A_{rob}) = \max_{i \in I} \frac{opt(r_i+1)}{opt(r_i)}$, resp.).*

Proof. A_{rob} simply computes a set of codes for the expected train by considering one additional pure run implied by a possible new car. If a new unexpected car v arrives, any algorithm in $\mathcal{A}^3_{rec}$ is able to reassign all codes, hence inserting v in the desired position. □

Theorem 9. *In Case 1- (2- and 3-, resp.), and considering $\mathcal{A}^3_{rec}$, $PoR(R_P) \geq \max_{i \in I} \frac{opt(r_i+1,c)}{opt(r_i,c)}$ $(PoR(R_P) \geq \max_{i \in I} \frac{opt(r_i+1,w)}{opt(r_i,w)}$ and $PoR(R_P) \geq \max_{i \in I} \frac{opt(r_i+1)}{opt(r_i)}$, resp.).*

Proof. The proof simply follows by observing that the new unexpected car, according to its required position, may constitute itself a pure run. The need of one further code is then necessary. □

From Theorem 8 and Theorem 9 the following corollary holds.

Corollary 4. *There exists a robust algorithm in Case 2- (and one in Case 3-) that is optimal when considering $\mathcal{A}^3_{rec}$.*

5 Conclusion

In this paper we have provided robustness in the context of shunting of train cars. Robustness by itself is a not well defined property for optimization problems when recovery strategies are available and/or necessary. We have focalized our attention on the definition of robustness algorithms. An algorithm is said to be robust according to some allowed recovery strategy, and against some specified disruptions, if it provides a solution which is valid also if a disruption occurs by possibly applying available recovery strategies. We also provide a measure for the price of robustness for an algorithm as the ratio between its performances and the performances of an optimal algorithm both applied on the expected input (without disruptions). The definition turns out to capture interesting properties among our evaluations on different shunting problems and scenarios. The proposed robust algorithms show how robustness heavily affects performances. Some algorithms that are optimal (in the robust meaning) with respect to some disruptions may become even unfeasible in other contexts. Another central issue concerns the available recovery capabilities. Intuitively, the more available recovery strategies are powerful, the less is the price of robustness for a robust algorithm. Contrary, we have shown that there are cases where increasing recovery capabilities does not affect obtained results.

This paper can be considered as a step forward in the definition and the application of notions concerning robustness. Many other applications related or not to shunting problems (or more in general to railways problems) can be studied by following the used approach. Another interesting future work would be also to study the dual of robust algorithms, i.e., recovery algorithms. What would be the design of a recovery algorithm once fixed the power/capabilities of a class of robust algorithms?

Acknowledgements

We like to thank Prof. Maria Domenica Di Benedetto for the constructive discussion and useful comments on robustness issues.

References

1. H. G. Bayer and B Sendhoff. Robust Optimization - A Comprehensive Survey. *Computer Methods in Applied Mechanics and Engineering*, 2007. to appear.
2. A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Mathematical Programming: Special Issue on Robust Optimization*, volume 107. Springer, Berlin, 2006.
3. D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.
4. 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.
5. S. Cornelsen and G. Di Stefano. Track assignment. *Journal of Discrete Algorithms*, 5(2):250–261, 2007.

6. E. Dahlhaus, P. Horak, M. Miller, and J. F. Ryan. The train marshalling problem. *Discrete Applied Mathematics*, 103(1-3):41–54, 2000.
7. M. Demange, G. Di Stefano, and B. Leroy-Beaulieu. On the online track assignment problem. Technical Report ARRIVAL-TR-0028, ARRIVAL Project, December 2006.
8. G. Di Stefano and M.L. Koči. A graph theoretical approach to the shunting problem. *Electr. Notes Theor. Comput. Sci.*, 92:16–33, 2004.
9. M. Fischetti and M. Monaci. Robust optimization through branch-and-price. In *Proceedings of the 37th Annual Conference of the Italian Operations Research Society (AIRO)*, 2006.
10. R. Freling, R. M. Lentink, L. G. Kroon, and D. Huisman. Shunting of passenger train units in a railway station. *Transportation Science*, 39(2):261–272, 2005.
11. R. Jacob. On shunting over a hump, Manuscript, 2007.
12. C. Liebchen, M. Lüebbecke, R. H. Möhring, and S. Stiller. Recoverable robustness. Technical Report ARRIVAL-TR-0066, ARRIVAL Project, 2007.
13. T. Winter and U. Zimmermann. Real-time dispatch of trams in storage yards. *Annals of Operations Research*, 96:287–315(29), 2000.