Project Number FP6-021235-2

ARRIVAL

Algorithms for **R**obust and online **R**ailway optimization:
Improving the **V**alidity and reli**A**bility of **L**arge scale systems

STREP
Member of the FET Open Scheme

ARRIVAL – TR – 0190

Recoverable robustness in shunting and timetabling

Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, Alfredo Navarra, Michael Schachtebeck, Anita Schöbel

February 2009

Recoverable robustness in shunting and timetabling[⋆]

Serafino Cicerone[1], Gianlorenzo D'Angelo[1], Gabriele Di Stefano[1], Daniele Frigioni[1], Alfredo Navarra[2], Michael Schachtebeck[3], and Anita Schöbel[3]

[1] Department of Electrical and Information Engineering, University of L'Aquila, Italy.
{serafino.cicerone,gianlorenzo.dangelo,gabriele.distefano,daniele.frigioni}@univaq.it
[2] Department of Mathematics and Computer Science, University of Perugia, Italy. navarra@dipmat.unipg.it
[3] Institute for Numerical and Applied Mathematics, Georg-August-University Göttingen, Germany.
{schachte,schoebel}@math.uni-goettingen.de

Abstract. In practical optimization problems, disturbances to a given instance are unavoidable due to unpredictable events which can occur when the system is running. In order to face these situations, many approaches have been proposed during the last years in the area of robust optimization. The basic idea of *robustness* is to provide a solution which is able to keep feasibility even if the input instance is disturbed, at the cost of optimality. However, the notion of robustness in every day life is much broader than that pursued in the area of robust optimization so far. In fact, robustness is not always suitable unless some *recovery strategies* are introduced. Recovery strategies are some capabilities that can be used when disturbing events occur, in order to keep the feasibility of the pre-computed solution. This suggests to study robustness and recoverability in a unified framework. Recently, a first tentative of unifying the notions of robustness and recoverability into a new integrated notion of *recoverable robustness* has been done in the context of railway optimization.

In this chapter, we review the recent algorithmic results achieved within the recoverable robustness model in order to evaluate the effectiveness of this model. To this aim, we concentrate our attention on two problems arising in the area of railway optimization: the *shunting* problem and the *timetabling* problem. The former problem regards the reordering of freight train cars over hump yards while the latter one consists in finding passenger train timetables in order to minimize the overall passengers traveling time. We also report on a generalization of recoverable robustness called *multi-stage recoverable robustness* which aims to extend recoverable robustness when multiple recovery phases are required.

1 Introduction

Many real world applications are characterized by a *strategic planning* phase and an *operational* phase. The main difference between the two phases resides in the time in which they are applied. The strategic planning phase aims to plan how to optimize the use of the available resources according to some objective function before the system starts to operate. The operational phase aims to have immediate reaction to disturbing events that can occur when the system is running. In general, the objectives of strategic planning and operational phase might be in conflict with each other.

In these scenarios, it is preferable to define a strategic plan which is able to keep feasibility even if the input is disturbed instead of a plan which optimizes the available resources for the undisturbed input. It follows that disturbances have to be considered both in the *strategic planning* phase and in the *operational* phase.

To face disturbances in the operational phase, the approaches used in the literature are mainly based on the concept of *online algorithms* [5]. An online recovery strategy has to be developed when unpredictable disturbances in planned operations occur and before the entire sequence of disturbances is known. The goal is to react fast while retaining as much as possible

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

of the quality of an optimal solution, that is, a solution that would have been achieved if the entire sequence of disturbances was known in advance.

To face disturbances in the strategic planning phase, the approaches used in the literature are mainly based on *stochastic programming* and *robust optimization*. Within stochastic programming (e.g., see [4, 25, 29]), there are two different approaches: *chance constrained programming* aims to find a solution that satisfies the constraints with high probability, while in *multi-stage stochastic programming*, an initial solution is computed in the first stage, and each time new random data is revealed, a recourse action is taken. However, stochastic programming requires detailed knowledge on the probability distributions of the disturbances which could be not available.

In robust optimization (e.g., see [1–3, 15]), the objective is purely deterministic. It aims to find a solution to an optimization problem which keeps feasibility when some disturbing events occur. For example, the notion of *strict robustness* introduced in [3] requires that a solution to an optimization problem has to be feasible for all admissible scenarios of a given set. The solution gained by this approach is fixed in the strategic planning phase and it does not need to be changed when disturbances occur. However, as the solution is fixed independently of the actual scenario, robust optimization leads to solutions that are too conservative and thus too expensive in many applications. One approach to compensate this disadvantage is the *light robustness* introduced in [16]. This approach adds slacks to the constraints. A solution is considered robust if it satisfies these relaxed constraints.

Despite the increasing interest, a final answer to the question "what is *robustness* for an optimization problem?" has not yet been given. In fact, the notion of robustness in every day life is much broader than that pursued in the area of robust optimization so far. The basic idea of robustness is given by a problem and some knowledge imperfection which one has to cope with. 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 unless some recovery strategies are introduced. Moreover, in many practical applications, there might be the possibility to intervene before some scheduled operations are being performed.

Usually, modifications that may occur are restricted to some specified subset of all possible ones. It is reasonable to require that, if a disturbance occurs, one would like to maintain as much as possible a pre-computed solution taking into account some "soft" recovery strategies. Recovering 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. This suggests to study robustness and recoverability in a unified way.

A first tentative of unifying the notions of *robustness* and *recoverability* into a new integrated notion of *recoverable robustness* has been done in [27] in the context of railway optimization. This new notion describes robustness with respect to (limited) recovery capabilities. It integrates robustness and recoverability as the solutions are required to be recoverable. The basic idea of recoverable robustness is to compute solutions that are robust against a limited set of scenarios and for a limited recovery. The quality of the robust solution is measured by its *price of robustness* that determines the trade-off between an optimal and a robust solution. Given an instance i of a problem, the price of robustness of i is the ratio between the cost of an optimal robust solution and the cost of the optimal (non robust) solution. The price of robustness of a recoverable robustness problem is then given by a worst case analysis, i.e., it is the maximum price of robustness among all the instances of the problem. Hence the price of robustness of a problem provides an upper bound to the loss that one has to pay in order to introduce recoverable robustness in an optimization problem by fixing some disturbances and recovery capabilities. Symmetrically, a lower bound is called *price of $\mathcal{A}$-recoverability*, where $\mathcal{A}$ is set of recovery capabilities. In [27], the aim is to provide the best robust solution, i.e., the one that minimizes the price of robustness.

In [6], algorithmic aspects of recoverable robustness have been highlighted by giving the definition of *robust algorithm* and of the corresponding *price of robustness*. A robust algorithm is an algorithm which provides a robust solution for each instance of a problem. The price of robustness of a robust algorithm A_{rob} is given by the worst case ratio, among all the possible instances of the problem, between the cost of the solution computed by A_{rob} for an instance i and the optimal (non robust) solution for i. The price of robustness of a recoverable robustness problem defined in [6] is then given by the price of the best possible robust algorithm which solves the given problem. Hence, the price of robustness of a problem here provides the loss that cannot be avoided by a robust algorithm, fixed some disturbances and recovery capabilities. If the price of robustness of an algorithm matches this minimal loss, then the algorithm is called *optimal*. If it is equal to 1, then no price has to be paid in order to achieve robustness and hence such an algorithm is called *exact*.

Notice that, given a recoverable robust problem $\mathcal{P}$, if there exists a robust algorithm that is able to find an optimal robust solution for any instance of $\mathcal{P}$, then the price of robustness of $\mathcal{P}$ defined in [27] and that defined in [6] are equivalent. However, the model given in [6] is more suitable for analyzing robust algorithms than that in [27] as the former concentrates on finding robust algorithms and comparing them by using their prices of robustness, while the latter concentrates on finding optimal robust solutions and recovery algorithms.

In this chapter, we intend to review the algorithmic results achieved within the recoverable robustness model given in [6] in order to evaluate the effectiveness of this model in both practical and theoretical frameworks. To this aim, we focus our attention on two problems arising in the area of railway optimization: the *shunting* problem and the *timetabling* problem. The former problem regards the reordering of freight train cars over hump yards while the latter one consists in finding passenger train timetables in order to minimize the overall passengers traveling time. We also report on a generalization of the recoverable robustness model called *multi-stage recoverable robustness* model proposed in [10]. It aims to extend recoverable robustness in the case of multiple disturbances which can arise in many practical optimization problems and require a sequence of recovery phases.

This chapter is organized as follows: in the next section we report the recoverable robustness model as given in [6]. In Section 3 we survey results on the shunting problem obtained in [6, 7]. In Section 4 we survey results on the timetabling problem obtained in [8–12]. In Section 5 we report the multi-stage recoverable robustness model given in [10] and provide an example on how to apply this model in the context of timetabling. Finally, in Section 6, we analyze the given models and propose some possible extensions, open problems and future research directions.

2 Recoverable robustness model

In this section, we report the recoverable robustness model given in [6] which is based on that given in [27].

The recoverable robustness model aims to introduce robustness in an optimization problem. In the remainder, an optimization problem P is characterized by the following parameters.

- I, the set of instances of P;
- F, a function that associates to any instance $i \in I$ the set of all feasible solutions for i;
- $f\colon S \to \mathbb{R}^{\geq 0}$, the objective function of P, where $S = \bigcup_{i\in I} F(i)$ is the set of all feasible solutions for P.

Note that, for several optimization problems, the objective function is defined to have values in $\mathbb{R}$. However, it is possible to turn any such problem into an equivalent one having values in

$\mathbb{R}^{\geq 0}$. Without loss of generality, from now on, minimization problems are considered. Additional concepts to introduce robustness requirements for a minimization problem P are needed:

- $M : I \rightarrow 2^I$ – a *modification* function for instances of P. This function models the following case. Let $i \in I$ be the considered input to the problem P, and let $s \in S$ be the planned solution for i. A *disturbance* is meant as a modification to the input i, and such a modification can be seen as a new input $j \in I$. Typically, the modification j depends on the current input i, and this fact is modeled by the constraint $j \in M(i)$. Hence, given $i \in I$, $M(i)$ represents the set of instances of P that can be obtained by applying all possible modifications to i. Of course, when a disturbance $j \in M(i)$ occurs, a new solution $s' \in F(j)$ has to be recomputed for P.
- $\mathbb{A}_{rec}$ – a class of *recovery algorithms* for P. Algorithms in $\mathbb{A}_{rec}$ represent the capability of recovering against disturbances. Since in a real-world problem the capability of recovering is limited in some way, the class $\mathbb{A}_{rec}$ can be defined in terms of some kind of *restrictions*, such as feasibility or algorithmic restrictions. An element $A_{rec} \in \mathbb{A}_{rec}$ works as follows: given a solution s for P and a modification $j \in M(i)$ of the current instance i, then $A_{rec}(s, j) = s'$ where $s' \in S$ represents the recovered solution for P.
  In what follows, some examples of recovery algorithm classes, used in the remainder of this chapter, are given.

  ***Class 1*: Strict robustness.** It models the case in which there are no recovery capabilities, that is, each algorithm $A_{rec} \in \mathbb{A}_{rec}$ fulfills the following constraint:

$$\forall i \in I, \forall s \in S, \forall j \in M(i),\ A_{rec}(s, j) = s. \tag{1}$$

  ***Class 2*: Bounded distance from the original solution.** $\mathbb{A}_{rec}$ is defined by imposing a constraint on the solutions provided by the recovery algorithms. In particular, the new (recovered) solutions computed by a recovery algorithm must not deviate too much from the original solution s, according to a distance measure d. Formally, given a real number $\Delta \in \mathbb{R}$ and a distance function $d : S \times S \rightarrow \mathbb{R}$, each element A_{rec} in such a class fulfills the following constraint:

$$\forall i \in I, \forall s \in S, \forall j \in M(i),\ d(s, A_{rec}(s, j)) \leq \Delta. \tag{2}$$

  Note that *Class 1* is contained in *Class 2*. In fact, for any distance function d, if $\Delta = 0$, then constraints (1) and (2) are equivalent.

  ***Class 3*: Bounded computational power.** $\mathbb{A}_{rec}$ is defined by bounding the computational power of recovery algorithms. Formally, given a function $t : S \times I \rightarrow \mathbb{N}$, each element A_{rec} in such a class fulfills the following constraint:

$$\forall i \in I, \forall s \in S, \forall j \in M(i),\ A_{rec}(s, j) \text{ must be computed in } O(t(s, j)) \text{ time.}$$

Given an optimization problem P, it can be turned into a recoverable robustness problem $\mathcal{P}$ as described below.

Definition 1. *A* recoverable robustness problem $\mathcal{P}$ *is defined by the triple* $(P, M, \mathbb{A}_{rec})$. *All the recoverable robustness problems form the class* RRP.

Definition 2. *Let* $\mathcal{P} = (P, M, \mathbb{A}_{rec}) \in$ RRP. *Given an instance* $i \in I$ *of* P, *an element* $s \in F(i)$ *is a* feasible solution *for* i *with respect to* $\mathcal{P}$ *if and only if the following relationship holds:*

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

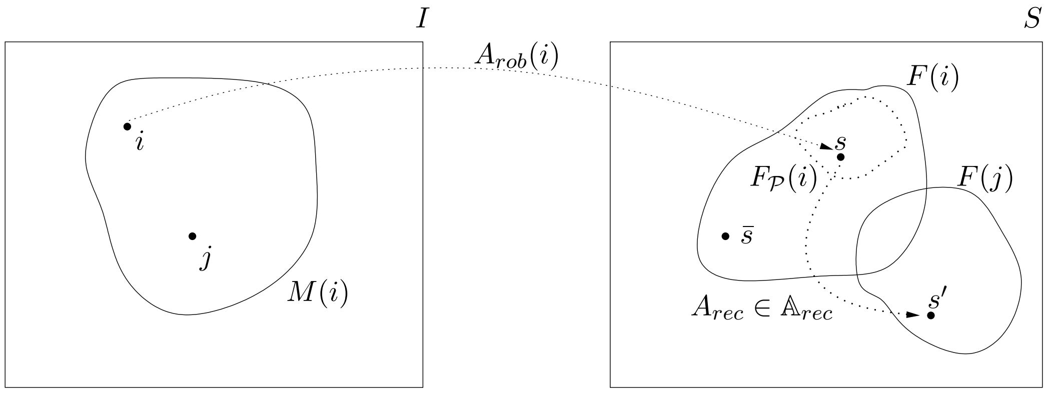

Fig. 1. A scenario for recoverable robustness problem: I, set of instances; S, set of solutions; $M(i)$, set of instances obtainable after a small modification; $F(i)$ and $F(j)$, set of feasible solutions for i and j respectively; $F_{\mathcal{P}}(i)$, set of recoverable solutions for i; $\bar{s}$, optimal non-robust solution for i; s, robust solution obtained by A_{rob}; s', recovered solution obtained by an algorithm $A_{rec} \in \mathbb{A}_{rec}$ after disturbance $j \in M(i)$.

In other words, $s \in F(i)$ is feasible for i with respect to $\mathcal{P}$ if it can be *recovered* by applying some algorithm $A_{rec} \in \mathbb{A}_{rec}$ for each possible disturbance $j \in M(i)$. The set of all the feasible solutions for i with respect to $\mathcal{P}$ is denoted by $F_{\mathcal{P}}(i)$. Formally:

$$F_{\mathcal{P}}(i) = \{s \in F(i) : s \text{ is a feasible solution for } i \text{ with respect to } \mathcal{P}\}.$$

In the remainder, solutions in $F_{\mathcal{P}}(i)$ are also called *robust solutions* for i with respect to the original problem P.

It is worth to mention that, if $\mathbb{A}_{rec}$ is *Class 1*, i.e., it is the class of algorithms that do not change the solution s, then the robustness problem $\mathcal{P} = (P, M, \mathbb{A}_{rec})$ represents the so-called *strict robustness problem* [3]. Note that in this case, given an instance i and a robust solution s for i, then for each possible modification $j \in M(i)$, $s \in F(j)$. This means that, since A_{rec} has no capability of recovering against possible disturbances, a robust solution has to "absorb" *any* possible disturbance.

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

A possible scenario for this situation is depicted in Figure 1. 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 $F_{\mathcal{P}}(i)$; this implies that every robust solution for i may be "very far" from the optimal solution $\bar{s}$. A "good" robust algorithm should find the best solution in $F_{\mathcal{P}}(i)$ for P, for each possible input $i \in I$. The quality of a robust algorithm is measured by the so-called price of robustness. The following definitions report the concepts of the price of robustness of both a robust algorithm and a recoverable robustness problem.

Definition 4. *Let $\mathcal{P} \in$ RRP. The* price of robustness *of a robust algorithm A_{rob} for $\mathcal{P}$ is*

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

Definition 5. *Let $\mathcal{P} \in$ RRP. The* price of robustness *of $\mathcal{P}$ is*

$$P_{rob}(\mathcal{P}) = \min\{P_{rob}(\mathcal{P}, A_{rob}) : A_{rob} \textit{ is a robust algorithm for } \mathcal{P}\}.$$

Definition 6. *Let $\mathcal{P} \in$ RRP and let A_{rob} be a robust algorithm for $\mathcal{P}$. Then,*

– A_{rob} *is* $\mathcal{P}$-optimal *if* $P_{rob}(\mathcal{P}, A_{rob}) = P_{rob}(\mathcal{P})$*;*
– A_{rob} *is* exact *if* $P_{rob}(\mathcal{P}, A_{rob}) = 1$.

A solution provided by an optimal (exact) robust algorithm is called an *optimal* (*exact*) solution. Notice that an exact algorithm is $\mathcal{P}$-optimal.

The price of robustness of an algorithm A_{rob} represents the quality of the solutions it provides. In particular, it measures the relative worst case loss induced by the value of the solutions provided by A_{rob} compared to the value of the optimal (non robust) ones. The price of robustness of the best robust algorithm defines the price of robustness of the problem. This value represents the minimal loss due to the introduction of robustness given by some disturbances and by some recovery capabilities.

3 Recoverable robust shunting

This section is devoted to survey on recent results concerning robustness in shunting problems [6, 7]. First the shunting over a hump yard model provided in [21–24] is described, and then, results obtained in this area in terms of recoverable robustness are reported.

3.1 Shunting over a hump yard

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 2. The hump yard is made of an input track where trains arrive,

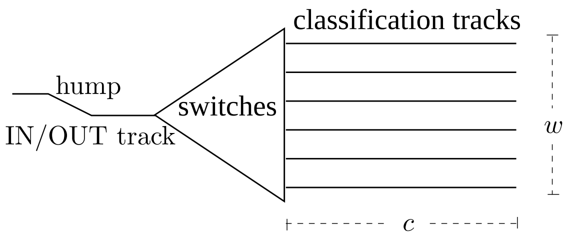

Fig. 2. Hump yard infrastructure composed of w classification tracks, each of size c.

and of 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 set of classification tracks is denoted by W, the size of W is denoted by w, and the size of each track, i.e., the number of cars that can fit into a classification track, by c. Therefore, an instance of the problem is given by a quadruple (T_{in}, T_{out}, W, c).

The hump yard supports a sorting operation by repeatedly doing the so called *track pull* operation which is made up of the following steps:

– connect the cars of one classification track into a train, called *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 3). The cost of the reordering is measured by the number of track pulls. Notice that at least one track pull must be performed as the hump yard is used only when one has to reorder or to park a train.

As in [22], three different variants of the shunting over a hump yard problem are considered by specifying constraints for parameters c and w. Namely,

Sh_1**:** c bounded, w unbounded;
Sh_2**:** c unbounded, w bounded;
Sh_3**:** c and w unbounded.

When convenient, Sh is used to refer to any of the above problems.

In [22] a polynomial algorithm for each of the above problems is given. In particular, a 2-approximation algorithm for Sh_1 and optimal algorithms for Sh_2 and Sh_3 are provided.

In what follows, the notation used in [6, 7, 22–24] to represent a *shunting plan* is described. A shunting plan specifies (i) a sequence S of h track pull operations given by the tracks whose cars are pulled, and (ii) for every pulled car which track it is sent to. Note that, if one track is pulled several times, then it appears in S more than once. Of course, if there is no limit on the number of tracks ($w \geq h$), then there is no need to reuse a track. Given S, the itinerary of a car can be described by the sequence of tracks it visits. For the task at hand, it is convenient to specify this sequence as a bit-string or code $b_1 \cdots b_h$ where the different bits stand for the pulled tracks and there is a 1 if and only if the car visits that track. Now, if track i is pulled, then the new destination of a car is given by the position of its next 1 in its code, i.e. the lowest index $j > i$ such that $b_j = 1$. A shunting plan must specify a track pull sequence S and it has to associate a code to each car. The length of each code is determined by the length of S and cars may share the same code.

An example is shown in Figure 3. The sequence of track pulls is given by $S = \{1, 2, 3, 4, 5\}$ 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 satisfies the property 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, and 11000, respectively. Figure 3 shows the sequence of configurations obtained after each track pull and reorder of the pulled cars according to their codes. The algorithm used in the example has been proposed in [22]. From now on, such an algorithm will be called A_{out}. It computes a shunting plan when c is bounded and the input train is unknown in advance. In particular, A_{out} provides n different codes, one for each car in T_{in}. Each code specifies the route that the corresponding car has to perform among the shunting yard in order to be placed in the desired position according to T_{out}. In [22] it has been shown that A_{out} 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.

Note that, when T_{in} is known in advance, two cars might be assigned with the same code. This would imply that they will have the same relative 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 *chain*. In a shunting plan, for each code x, a *pure chain* is the set of all cars associated with x.

In practice, the number of chains in T_{in} along with the hump yard structure represent the key quantity with respect to the number of track pulls that must be performed by a shunting plan in order to obtain the desired T_{out}. The following further notation is used: $opt(k, c, w) \geq 1$ is the number of track pulls needed by an optimal shunting plan in order to manage k cars/chains over

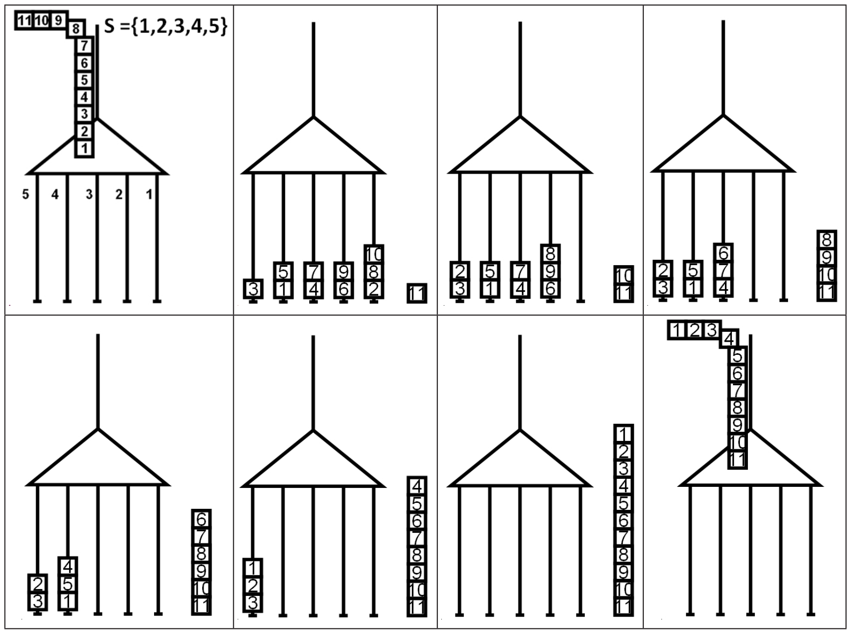

Fig. 3. Example of a shunting plan when $c = 3$ and the number of track pulls is set to 5. Cars from 11 down to 1 are associated with codes 00000, 00001, 00010, 00011, 00100, 00110, 01000, 01100, 10000, 10001, 11000 respectively. T_{out} is composed outside the hump yard and the corresponding track is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

a hump yard made of w tracks, each of size c (for Sh_1 and Sh_3, $w = \infty$, while for Sh_2 and Sh_3, $c = \infty$); $apx(k, c, w)$ is the best known approximation algorithm for the corresponding shunting problem, and $apxr$ is its approximation ratio (when it is clear by the context, parameters equal to ∞ are removed from the previous notation); C denotes the set of codes assigned by an algorithm to the cars. Furthermore, for every instance $i = (T_{in}, T_{out}, W, c)$, r_i and n_i denote the number of chains and cars in T_{in}, respectively.

3.2 Robust algorithms

This section presents a surveys of the results obtained in [6, 7] on robust algorithms for the shunting problems described above. For example, Sh_1 is defined by

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

Regarding the modification function M, four different possibilities are considered:

M_1**:** one car can arrive in an unexpected incoming position;
M_2**:** the incoming train contains one additional unexpected car;
M_3**:** the incoming train contains one car less than expected;
M_4**:** one of the classification tracks composing the hump yard may fault.

Concerning recovery algorithms, the following three classes are considered:

- $\mathbb{A}^1_{rec}$**:** $\forall A \in \mathbb{A}^1_{rec}$, $\forall (i,s) \in I \times S$, $\forall j \in M(i)$, $A(i,s,j) = s$, i.e., there are no recovery strategies to apply;
- $\mathbb{A}^2_{rec}$**:** $\forall A \in \mathbb{A}^2_{rec}$, $\forall (i,s) \in I \times S$, $\forall j \in M(i)$, $A(i,s,j) = s'$ where s' may differ from s by at most one code without affecting the track pull sequence, i.e., at most one pure chain may be assigned with a new code of the same length;
- $\mathbb{A}^3_{rec}$**:** $\forall A \in \mathbb{A}^3_{rec}$, $\forall (i,s) \in I \times S$, $\forall j \in M(i)$, $A(i,s,j) = s'$ where s' may differ from s by all the set of codes without affecting the track pull sequence, i.e., every pure chain may be assigned with a new code of the same length.

The class $\mathbb{A}^1_{rec}$ is equivalent to *Class 1*, while classes $\mathbb{A}^2_{rec}$ and $\mathbb{A}^3_{rec}$ belong to *Class 2*.

Note that each of the three defined classes of recovery algorithms does not affect the scheduled track pulls sequence defined by a robust shunting algorithm. This is motivated by the fact that modifying the track pulls sequence is expensive as it requires to change the switches setting or increase the number of track pulls. Recovery capabilities, instead, should be cheap operations since they cannot be planned a priori but are used during the operational phase. By definition, every upper bound to the price of robustness of each shunting problem with $\mathbb{A}^1_{rec}$ holds for the same problem with $\mathbb{A}^2_{rec}$ as well as every upper bound obtained with $\mathbb{A}^2_{rec}$ holds for $\mathbb{A}^3_{rec}$. Moreover, every lower bound obtained with $\mathbb{A}^3_{rec}$ holds for $\mathbb{A}^2_{rec}$ as well as every lower bound obtained with $\mathbb{A}^2_{rec}$ holds for $\mathbb{A}^1_{rec}$.

Given a shunting problem Sh, a modification function M and a class of recovery algorithms $\mathbb{A}_{rec}$, the corresponding recoverable robustness problem is denoted by $\mathcal{SH} = (Sh, M, \mathbb{A}_{rec})$. Tables 1, 2 and 3 summarize the obtained results for all the considered robustness problems arising from Sh_1, Sh_2 and Sh_3, respectively. In these tables, A_{rob} denotes the best robust algorithm given in [6, 7] for the specific problem at hand.

	Shunting Problem Sh_1			
Modifications		$\mathbb{A}^1_{rec}$	$\mathbb{A}^2_{rec}$	$\mathbb{A}^3_{rec}$
M_1	$P_{rob}(\mathcal{SH})$	$\geq \max_{i\in I} \frac{opt(n_i,c)}{opt(r_i,c)}$	≥ 2	≥ 2
	$P_{rob}(\mathcal{SH}, A_{rob})$	$\max_{i\in I} \frac{opt(n_i,c)}{opt(r_i,c)}$	3	3
M_2	$P_{rob}(\mathcal{SH})$	indef.	$\geq \max_{i\in I} \frac{opt((n_i+1)/3,c)}{opt(r_i,c)}$	$\geq \max_{i\in I} \frac{opt(r_i+1,c)}{opt(r_i,c)}$
	$P_{rob}(\mathcal{SH}, A_{rob})$	no solution	$\max_{i\in I} \frac{opt(n_i+1,c-1)+1}{opt(r_i,c)}$	$\max_{i\in I} \frac{apx(r_i+1,c)}{opt(r_i,c)}$
M_3	$P_{rob}(\mathcal{SH})$	1	1	1
	$P_{rob}(\mathcal{SH}, A_{rob})$	2	2	2
M_4	$P_{rob}(\mathcal{SH})$	indef.	≥ 2	≥ 2
	$P_{rob}(\mathcal{SH}, A_{rob})$	no solution	$\lvert C\rvert + 1$	3

Table 1. Price of Robustness for Sh_1

3.3 One car in an unexpected incoming position

In order to better understand the ideas behind recoverable robust algorithms, in this section, more details are provided concerning the case when the modification function M_1 is considered. Given an instance $i = (T_{in}, T_{out}, W, c)$ of a shunting optimization problem Sh, let $M_1(i)$ represent all possible instances (T'_{in}, T_{out}, W, c) obtainable from i by changing the position of just one car in T_{in}. The following lemma describes which practical situation a robust plan must be able to absorb/recover with respect to a car incoming at an unexpected position.

Lemma 1 ([6, 7]). *Let v be a car arriving at the hump yard in a different position than expected. At most one additional pure chain must be managed with respect to the expected case.*

		Shunting Problem Sh_2		
Modifications		$\mathbb{A}^1_{rec}$	$\mathbb{A}^2_{rec}$	$\mathbb{A}^3_{rec}$
M_1	$P_{rob}(\mathcal{SH})$	$\geq \max\limits_{i \in I} \frac{opt(n_i,w)}{opt(r_i,w)}$	≥ 2	≥ 2
	$P_{rob}(\mathcal{SH}, A_{rob})$	$\max\limits_{i \in I} \frac{opt(n_i,w)}{opt(r_i,w)}$	2	2
M_2	$P_{rob}(\mathcal{SH})$	indef.	$\geq \max\limits_{i \in I} \frac{opt((n_i+1)/3,w)}{opt(r_i,w)}$	$\geq \max\limits_{i \in I} \frac{opt(r_i+1,w)}{opt(r_i,w)}$
	$P_{rob}(\mathcal{SH}, A_{rob})$	no solution	$\max\limits_{i \in I} \frac{opt(n_i+1,w)+1}{opt(r_i,w)}$	$\max\limits_{i \in I} \frac{opt(r_i+1,w)}{opt(r_i,w)}$
M_3	$P_{rob}(\mathcal{SH})$	1	1	1
	$P_{rob}(\mathcal{SH}, A_{rob})$	1	1	1
M_4	$P_{rob}(\mathcal{SH})$	indef.	indef.	≥ 2
	$P_{rob}(\mathcal{SH}, A_{rob})$	no solution	no solution	$\max\limits_{i \in I} \frac{opt(r_i,w-1)+1}{opt(r_i,w)}$

Table 2. Price of Robustness for Sh_2

		Shunting Problem Sh_3		
Modifications		$\mathbb{A}^1_{rec}$	$\mathbb{A}^2_{rec}$	$\mathbb{A}^3_{rec}$
M_1	$P_{rob}(\mathcal{SH})$	$\geq \max\limits_{i \in I} \frac{opt(n_i)}{opt(r_i)}$	≥ 2	≥ 2
	$P_{rob}(\mathcal{SH}, A_{rob})$	$\max\limits_{i \in I} \frac{opt(n_i)}{opt(r_i)}$	2	2
M_2	$P_{rob}(\mathcal{SH})$	indef.	$\geq \max\limits_{i \in I} \frac{opt((n_i+1)/3)}{opt(r_i)}$	$\geq \max\limits_{i \in I} \frac{opt(r_i+1)}{opt(r_i)}$
	$P_{rob}(\mathcal{SH}, A_{rob})$	no solution	$\max\limits_{i \in I} \frac{opt(n_i+1)+1}{opt(r_i)}$	$\max\limits_{i \in I} \frac{opt(r_i+1)}{opt(r_i)}$
M_3	$P_{rob}(\mathcal{SH})$	1	1	1
	$P_{rob}(\mathcal{SH}, A_{rob})$	1	1	1
M_4	$P_{rob}(\mathcal{SH})$	indef.	≥ 2	≥ 2
	$P_{rob}(\mathcal{SH}, A_{rob})$	no solution	$\|C\| + 1$	2

Table 3. Price of Robustness for Sh_3

In a shunting plan, Lemma 1 is reflected in the need of at most one additional code. However, if the class of available recovery algorithm is $\mathbb{A}^1_{rec}$, then the following lemma indicates how an optimal robust algorithm should behave.

Lemma 2 ([6, 7]). *Let $\mathcal{SH} = (Sh, M_1, \mathbb{A}^1_{rec})$. For every input train T_{in}, any robust shunting algorithm A_{rob} must provide a unique code to each car of T_{in}.*

Let us consider problem Sh_1. As mentioned in Section 3.1, two solutions have been proposed in [22] for this case. The first solution provides a 2-approximation of the optimum, i.e., $apxr = 2$, but it cannot be used for robustness purposes when considering $\mathbb{A}^1_{rec}$ since it does not fulfill the condition of Lemma 2. The second solution, i.e., algorithm A_{out} described before, turns out to be $\mathcal{SH}$-optimal when $\mathcal{SH} = (Sh, M_1, \mathbb{A}^1_{rec})$.

Theorem 1 ([6, 7]). *Let $\mathcal{SH} = (Sh_1, M_1, \mathbb{A}^1_{rec})$. There exists a $\mathcal{SH}$-optimal robust shunting algorithm A_{rob} such that $P_{rob}(\mathcal{SH}, A_{rob}) = \max\limits_{i \in I} \frac{opt(n_i,c)}{opt(r_i,c)}$.*

Even though A_{out} is $\mathcal{SH}$-optimal for $\mathbb{A}^1_{rec}$, i.e., $P_{rob}(\mathcal{SH}, A_{rob}) = P_{rob}(\mathcal{SH})$, it is not exact since it could exist an instance i such that $opt(n_i, c) > opt(r_i, c)$.

4 Recoverable robust timetabling

In this section, we survey results in [8–12] concerning recoverable robust problems in the field of timetabling. First, we describe the problems at hand and then, for each problem, we give its complexity and some solving algorithms. Furthermore, we report experimental results in real world scenarios.

The problem of *timetable planning* (in short *timetabling*) arises in the strategic planning phase for transportation systems and requires to compute a timetable for passenger trains that determines minimal passengers traveling times. However, many disturbing events can occur during the operational phase that might completely change the scheduled activities. The main effect of such disturbing events is the arising of delays, caused by malfunctioning infrastructure/devices, special events, or extreme weather conditions. The conflicting objectives of strategic against operational planning are evident in timetable optimization. In fact, a timetable that lets trains sit at stations for some time will not suffer from small delays of arriving trains because delayed passengers can still catch potential connecting trains. On the other hand, big delays can cause passengers to lose trains and hence imply extra traveling time. The problem of deciding when to guarantee connections from a delayed train to a connecting train is known in the literature as *delay management problem* [14, 17–20, 30, 31] and has a twofold impact. On the one hand, if a connection is maintained, the passengers arriving late still catch their connection, but passengers in the connecting train now face a delay and may miss subsequent connections. On the other hand, if a connection is not maintained, the passengers on the departing train are on time, but those arriving late have to wait for the next train. The latter implies that the delay can propagate through the railway network. The trade-off between these two effects leads to the natural objective of minimizing the overall delay faced by the total passenger population. Despite its natural formalization, the problem turns out to be very complicated to be optimally solved. In fact, it has been shown to be NP-hard in the general case, while it is polynomial in some particular cases (see [17, 18, 30]).

In railway systems, events and dependencies among events are modeled by means of an *event activity network* (see [31]). This is a directed graph where the vertices represent events (e.g., arrivals or departures of trains) and the arcs represent activities occurring between events (e.g., waiting in a train, driving between stations or changing to another train). However, event activity networks are a particular class of Directed Acyclic Graphs (DAGs). Hence, in the remainder of the section, we will survey on more general results concerning DAGs.

Let us consider a DAG $G = (V, A)$ with one specified vertex v_0 such that there exists a directed path from v_0 to each other vertex.

A solution for a timetabling problem requires to assign a time π_v to each event $v \in V$ in such a way that all the constraints provided by the set of activities are respected. In detail, the acyclic timetabling problem is then given as

$$(TT) \quad \min f(\pi) = \sum_{u \in V} w_u \pi_u$$

such that

$$\pi_v - \pi_u \geq L_a \quad \forall a = (u, v) \in A \tag{3}$$

$$\pi_u \in \mathbb{R}^{\geq 0} \quad \forall u \in V \tag{4}$$

where $w_u \in \mathbb{R}$ are weights representing the importance of the corresponding events and $L_a \in \mathbb{R}^{>0}$ are given lower bounds indicating the minimal duration that is needed for activity $a \in A$. The amount $\pi_v - \pi_u - L_a$ is called *slack time* of activity a and is denoted by s_a. An instance i of TT is specified by a triple $i = (G, w, L)$.

The subproblem of TT where $w_v \geq 0$ for each vertex $v \in V$ is denoted by TT^v. Furthermore, a variant TT^a of TT where weights are associated to arcs instead of vertices is considered.

Formally, for this variant, w is defined as $w : A \to \mathbb{R}^{\geq 0}$, and TT^a is given as

$$(TT^a) \quad \min f_{arcs}(\pi) = \sum_{a=(v,u)\in A} w_a(\pi_v - \pi_u)$$

subject to constraints (3) and (4). In [10, 11] it has been shown that TT^a is a subproblem of TT.

In order to turn problems TT^v and TT^a into recoverable robustness problems, it is needed to define a modification function and a class of recovery algorithms.

The modification function is defined by admitting a single delay of at most α time. It is modelled as an increase of the minimal duration time of the delayed activity. Formally, given an instance i of TT, $M(i)$ is defined as follows:

$$M(i) = \left\{(G, w, L') \mid \exists\ \bar{a} \in A : L_{\bar{a}} \leq L'_{\bar{a}} \leq L_{\bar{a}} + \alpha \text{ and } L'_a = L_a\ \forall a \neq \bar{a}\right\}.$$

A recovery algorithm is allowed to change the time of at most Δ events where $\Delta \in \mathbb{N}$. The class of these algorithms is denoted by $\mathbb{A}_\Delta$. Formally, given an instance i of TT, a solution π for i and a modification $i' \in M(i)$, then a solution π' computed by a recovery algorithm in $\mathbb{A}_\Delta$ fulfills

$$\left|\left\{u \in V : \pi'_u \neq \pi_u\right\}\right| \leq \Delta.$$

Notice that class $\mathbb{A}_\Delta$ belongs to *Class 2*.

We denote by $\mathcal{TT}^v = (TT^v, M, \mathbb{A}_\Delta)$ and $\mathcal{TT}^a = (TT^a, M, \mathbb{A}_\Delta)$ the robust problems derived from TT^v and TT^a.

A general approach for tackling $\mathcal{TT}^v$ and $\mathcal{TT}^a$ is to add slack time to the initial timetable, i.e., to find a timetable π such that for each $a = (u, v)$

$$\pi_v - \pi_u \geq L_a + s_a$$

for some *slack times* s_a. It follows that $\mathcal{TT}^v$ and $\mathcal{TT}^a$ can be solved in two steps. The first step consists in finding a slack times assignment s which ensures the robustness for an instance i, while the second step consists in solving the instance i' of the (non robust) problem TT obtained by adding s to the minimal duration times of activities of i. The second step is performed by algorithm Alg_s^+ defined as follows.

Algorithm Alg_s^+

```
INPUT:      An instance $i = (G, w, L)$ of $TT$.
ALGORITHM:  1. $\bar{L}_a := L_a + s_a$ for all $a \in A$.
            2. Solve $\bar{i} = (G, w, \bar{L})$ optimally.
```

Instead of adding a positive slack time to the lower bounds L_a, one can also multiply them with factors $t_a \geq 1$. This approach is known as *proportional buffering* and has been studied and applied in the context of timetabling [28]. It leads to the following variant Alg_t^* which differs from Alg_s^+ only in Step 1:

Algorithm Alg_t^*

```
INPUT:      An instance $i = (G, w, L)$ of $TT$.
ALGORITHM:  1. $\bar{L}_a := t_a * L_a$ for all $a \in A$.
            2. Solve $\bar{i} = (G, w, \bar{L})$ optimally.
```

A special case of algorithm Alg_s^+ is $\mathsf{Alg}_{\bar{s}}^+$ where the same slack is added to each activity, i.e., $\bar{s} = (s, \ldots, s)$. Given two positive real numbers a, b, when $s = (a, \ldots, a) \in \mathbb{R}^{|A|}$ and $t = (b, \ldots, b) \in \mathbb{R}^{|A|}$, we denote such algorithms by Alg_a^+ and Alg_b^*, respectively. It is intuitively clear that algorithms Alg_s^+ and Alg_t^* are robust if s is large enough and that the price of robustness increases in s. These observations are formally stated in the following lemma (formulated only for Alg_s^+, the statement for Alg_t^* is similar).

Lemma 3 ([10, 11]). *Consider Alg_s^+ as an algorithm for solving both $\mathcal{TT}^v$ and $\mathcal{TT}^a$. Then:*

1. *Alg_s^+ is robust for $\Delta = 0$ (strict robustness) if $s_a \geq \alpha$ for all $a \in A$.*
2. *Let Alg_s^+ be robust. Then $\mathsf{Alg}_{s'}^+$ is robust if $s'_a \geq s_a$ for all $a \in A$.*
3. *The price of robustness is monotone in s. In particular, if $\mathsf{Alg}_{s^1}^+$ and $\mathsf{Alg}_{s^2}^+$ are robust and $s_a^2 \geq s_a^1$ for all $a \in A$, then:*
   - $P_{rob}(\mathcal{TT}^v, \mathsf{Alg}_{s^1}^+) \leq P_{rob}(\mathcal{TT}^v, \mathsf{Alg}_{s^2}^+)$.
   - $P_{rob}(\mathcal{TT}^a, \mathsf{Alg}_{s^1}^+) \leq P_{rob}(\mathcal{TT}^a, \mathsf{Alg}_{s^2}^+)$.

Problem TT (as well as Step 2 of algorithms Alg_s^+ and Alg_t^*) can be solved in polynomial time by linear programming, but, if G is a tree or if $w_u \geq 0$ for all $u \in V$, it can be solved more efficiently by the *Critical Path Method* (*CPM*) of project planning (see e.g. [26]). This method also plays an important role in the recovery algorithms in $\mathbb{A}_\Delta$. Given an instance $i = (G, w, L)$ of TT, *CPM* works as follows.

Algorithm *CPM*

```
INPUT:      An instance $i = (G, w, L)$ of $TT$.
ALGORITHM:  1. For each $v \in V$
            2.     if $v = v_o$ then $\pi_v := 0$
            3.     else $\pi_v := \max\{\pi_u + L_a \ : \ a = (u, v) \in A\}$
```

The next two subsections are devoted to problems $\mathcal{TT}^v$ and $\mathcal{TT}^a$, respectively. For each problem, we first give results for general DAGs, then we restrict the topologies of the graphs to trees or linear graphs.

4.1 Problem $\mathcal{TT}^v$

Results for general DAGs. In [10, 11], it has been shown that computing the price of robustness of problem $\mathcal{TT}^v$ is NP-hard by a reduction from 3SAT. Furthermore, when the value of Δ is fixed and is not in the input of the problem, then computing the price of robustness of such a problem remains NP-hard for any $\Delta \geq 3$.

As the problem is NP-hard, approximate solutions for $\mathcal{TT}^v$ have been provided. In [10, 11], the focus has been posed only on the strict robustness problem, i.e. the recovery algorithms have been restricted to *Class 1* by considering $\Delta = 0$. Note that, if an algorithm is robust for such a problem, then it is robust also for the problem arising by allowing any $\Delta > 0$. However, in these cases, the price of robustness of a robust algorithm could be far from the price of robustness of the problem and hence the solution obtained could be too expensive. With the aim of finding such hyper-robust solutions, in [10, 11], algorithms Alg_α^+ and Alg_γ^* have been used where $\gamma = (1 + \frac{\alpha}{L_{min}})$, α is the maximum delay allowed and L_{min} is the minimum value assigned by function L with respect to all the possible instances of $\mathcal{TT}^v$.

For $\Delta = 0$, every robust algorithm for $\mathcal{TT}^v$ must provide solutions that assign a slack time of at least α to each activity. Then, it follows that Alg_α^+ is a robust algorithm for $\mathcal{TT}^v$. To show that also Alg^*_γ is a robust algorithm for $\mathcal{TT}^v$, it is sufficient to observe that for each activity $a \in A$,

$$\gamma L_a = (1 + \frac{\alpha}{L_{min}})L_a = L_a + \alpha \frac{L_a}{L_{min}} \geq L_a + \alpha.$$

The following theorem shows the price of robustness of Alg^*_γ.

Theorem 2 ([10, 11]). $P_{rob}(\mathcal{TT}^v, \mathsf{Alg}^*_\gamma) = 1 + \alpha/L_{min}$.

In [10, 11], it has been shown that for each instance i of $\mathcal{TT}^v$, $f(\mathsf{Alg}^+_\alpha(i)) \leq f(\mathsf{Alg}^*_\gamma(i))$, that is, Alg^+_α is always better than Alg^*_γ. It implies the following result.

Corollary 1 ([10, 11]). $P_{rob}(\mathcal{TT}^v, \mathsf{Alg}^+_\alpha) \leq 1 + \alpha/L_{min}$.

Results for trees. In [12, 13], the attention is devoted to the subproblem of $\mathcal{TT}^v$ where the DAG G is an out-tree.

The motivation for focussing on trees is that, also in practice, a tree can be very useful to model dependencies among trains in the particular case where the railway system considered is a *single-line corridor*. A corridor is a sequence of stations, represented by a line, where each station is served by many trains of different types. Types of trains mostly concern the locations that each train serves and its maximal speed. For an example, see Figure 4. In these systems, it is a practical evidence that slow trains wait for faster trains in order to serve passengers to small stations. This situation is modelled with the only assumption that the changes of passengers from one train to another at a station must be guaranteed only when the second train is starting its journey from the current station.

Let us consider the real world example provided in Figure 4 where three trains serve the same line. The slowest train, the Espresso, goes from Verona to Bologna, the Interregionale goes from Fortezza to Bologna, and the Euro-City goes from Brennero to Bologna. The Euro-

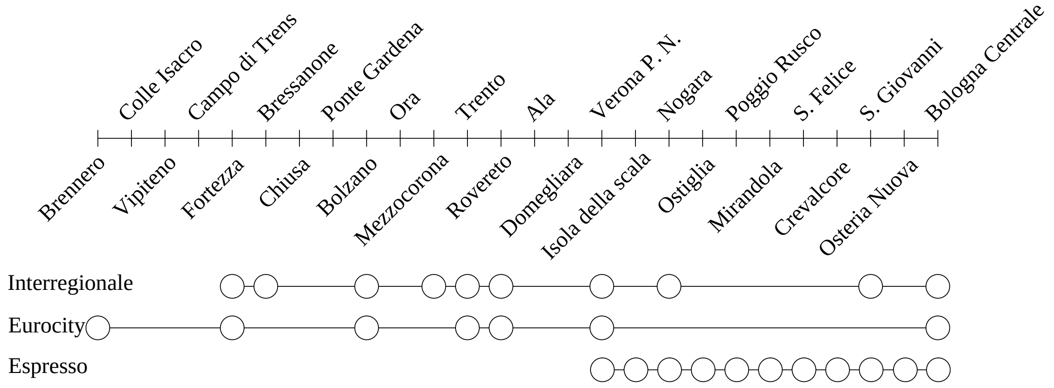

Fig. 4. Example of three trains serving a same line.

City starts its journey before all the other trains, and it arrives at Verona station before the Interregionale. There, the Espresso is scheduled to start its journey before the Interregionale arrives. Hence, there is an arc between the Euro-City and the starting event corresponding to the Interregionale at Fortezza station, and another arc connecting the Euro-City to the starting event of the Espresso at Verona station. As described above, an arc which represents a changing activity can only connect one vertex to the head of a branch. The DAG obtained by this procedure is a tree, the tree corresponding to the scenario in Figure 4 is shown in Figure 5.

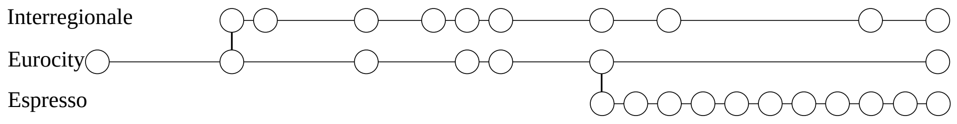

Fig. 5. A tree obtainable from the example provided by Figure 4.

Let $T = (V, A)$ be a tree rooted in v_0. If $v \in V$, T_v denotes the subtree of T rooted in v. Given a subtree T_v, $N_o(T_v)$ denotes the set of vertices y such that $(x, y) \in A$, $x \in T_v$ and $y \notin T_v$; $deg(v)$ denotes $|N_o(T_v)|$.

In [12], it has been shown that the problem $\mathcal{TT}^v$ restricted to trees remains NP-hard and a pseudo-polynomial time algorithm $\mathtt{SA}_{\Delta}$ for fixed Δ has been provided.

The proposed algorithm looks for solutions which assign only slack times of size α as it has been proved that for any instance, there exists a $\mathcal{TT}^v$-optimal solution which fulfills this condition.

In order to characterize a solution π, the following definition and lemma are needed.

Definition 7. *Given a solution π to $\mathcal{TT}^v$ and a vertex $v \in V$, a* ball *is the maximal subtree $B_\pi(v)$ rooted in v such that for each arc $a = (x, y)$ in $B_\pi(v)$, $s_a = 0$.*

Lemma 4 ([12]). *For each instance of $\mathcal{TT}^v$, there exists a $\mathcal{TT}^v$-optimal solution such that for each $v \in V$, $B_\pi(v)$ cannot be extended by adding any vertex from $N_o(B_\pi(v))$ while keeping feasibility and, unless $\Delta = 0$, at most one of two consecutive arcs has a slack time of α.*

For any $\Delta \geq 1$, there exists a $\mathcal{TT}^v$-optimal solution π with the following structure. By Lemma 4, for each arc a outgoing from the root v_0, $s_a = 0$. Then, for each $v \in N_o(v_0)$, π induces a ball $B_\pi(v)$ such that $|B_\pi(v)| \leq \Delta$. In particular, $|B_\pi(v)| < \Delta$ only if $|T_v| < \Delta$. As a consequence, $|B_\pi(v_0)| \leq 1 + \Delta \cdot deg(v_0)$. For each arc $a = (x, y)$ with $x \in B_\pi(v_0)$ and $y \notin B_\pi(v_0)$, $s_a = \alpha$. By Lemma 4, for each arc a outgoing from y, $s_a = 0$, and the same arguments used for $B_\pi(v_0)$ can be used to characterize $B_\pi(y)$.

A possible approach can be that of enumerating all the solutions with the above structure and choosing the cheapest one. Note that such an approach has a computational time which is exponential in n. In what follows we report the recursive approach of [12] which avoids to consider a large number of solutions and thus reduces the computational time to a polynomial in n. The algorithm works as follows. It assigns $\pi(v_0) = 0$ and no slack times to arcs outgoing v_0. Then, for each $v \in N_o(v_0)$ it has to decide which subtree of T_v belongs to $B_\pi(v_0)$. To do this, it evaluates the cost, in terms of the value of the objective function, of any possible subtree B of T_v rooted in v of size at most Δ and then chooses the subtree which implies the cheapest solution.

For each already defined ball B_π, this procedure is then repeated for each vertex $v \in N_o(B_\pi)$ which does not belong to an already defined ball by using v as the root.

The cost of a subtree B rooted in v is computed as the value of the objective function when B is chosen as a ball rooted in v. That is, for each arc $a \in B$, $s_a = 0$; for each $a = (x, y) \in A$ with $x \in B$ and $y \notin B$, $s_a = \alpha$; and for each vertex in T_y, an optimal solution is chosen. Computing this cost requires to know the optimal solution of a subtree, this is done by recursively using the algorithm.

The following theorems provide the theoretical results concerning the performances of $\mathtt{SA}_{\Delta}$, $\Delta \geq 1$, on a tree with n vertices.

Theorem 3 ([12]). *$\mathtt{SA}_{\Delta}$ is $\mathcal{TT}^v$-optimal.*

Theorem 4 ([12]). *$P_{rob}(\mathcal{TT}^v, \mathtt{SA}_{\Delta}) \leq 1 + \frac{\alpha}{2}$.*

Theorem 5 ([12]). $P_{rob}(\mathcal{TT}^v) \geq 1 + \frac{\alpha}{\Delta+1}$.

Theorem 6 ([12]). *$\texttt{SA}_\Delta$ requires $O(n^{\Delta+1})$ time and $O(n^2)$ space.*

In [12], faster algorithms for the special cases of $\Delta \in \{0, 1, 2\}$ have been provided. In particular, when $\Delta \in \{0, 1\}$ ($\Delta = 2$, resp.), linear (quadratic, resp.) time algorithms have been provided.

In [13] these algorithms have also been experimentally studied in practical, real world scenarios. One of the experimental results of [13] concerning a particular real-world corridor and three values of α is reported in Figure 6. It shows the price of robustness of the solutions obtained by $\texttt{SA}_\Delta$ and the computational time needed as functions of Δ. It turned out that algorithm $\texttt{SA}_\Delta$

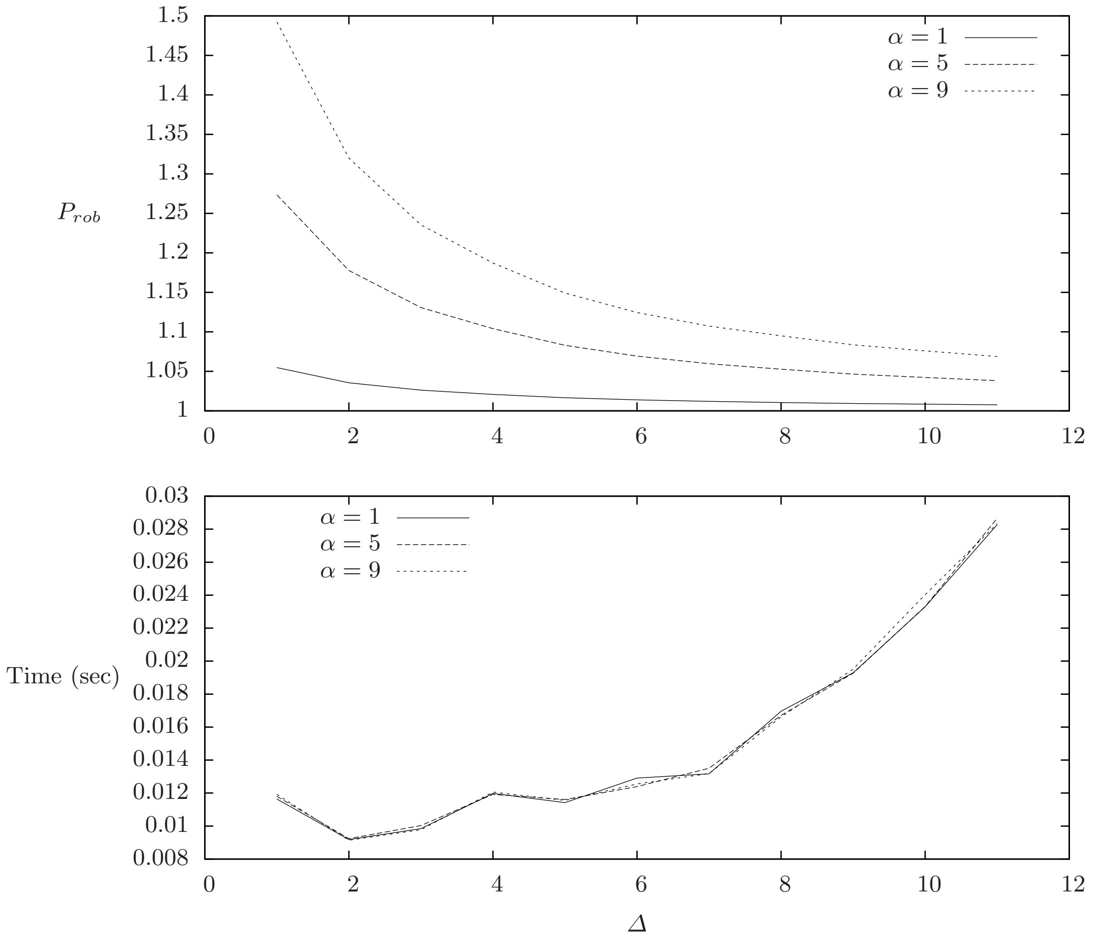

Fig. 6. Price of robustness and computational time needed by $\texttt{SA}_\Delta$ in a particular real-world corridor

is very fast in practice (it only needs less than 30 milliseconds in the case reported in Figure 6 and a few seconds in the worst cases), and its price of robustness rapidly tends to 1 while Δ increases. In all the cases analyzed, the real price of robustness is even less than the theoretical lower bound which is computed on the worst case instance.

Results for linear graphs. In [10, 11], it has been shown that, if the subproblem of $\mathcal{TT}^v$ where the DAG $G = (V, A)$ is a linear graph is considered, the price of robustness can be optimally computed in polynomial time. In detail, in [10, 11], a linear time algorithm is given. The idea of the algorithm is to add slack times "as late as possible". Let a linear graph be defined by a set of vertices $V = \{v_1, \dots, v_{|V|}\}$, ordered such that $A = \{a_1 = (v_1, v_2), \dots, a_{|A|} = (v_{|V|-1}, v_{|V|})\}$. Define s^α by

$$s^\alpha_{a_j} := \begin{cases} \alpha & \text{if } (\Delta+1)|j \\ 0 & \text{else} \end{cases} \tag{5}$$

for all arcs $a_j \in A$. Then, the algorithm $\mathsf{Alg}^+_{s^\alpha}$ adds s^α_a to L_a for each $a \in A$ and calculates a solution for $\mathcal{TT}^v$ by applying CPM on the resulting instance.

4.2 Problem $\mathcal{TT}^a$

Results for general DAGs. By exploiting similar arguments used for TT^v, in [8] it has been shown that computing the price of robustness of problem $\mathcal{TT}^a$ is NP-hard. Furthermore, in [9] it has been shown that computing the price of robustness $\mathcal{TT}^a$ remains NP-hard for any fixed $\Delta \geq 3$ and it remains NP-hard for any fixed $\Delta \geq 5$ when the considered DAG is restricted to an event activity network.

As computing the price of robustness of $\mathcal{TT}^a$ is NP-hard, there exist no polynomial $\mathcal{TT}^a$-optimal algorithms, unless P=NP. It makes sense then to investigate restricted scenarios which may be of practical interest.

In [8, 9], problem $\mathcal{TT}^a$ has been analyzed by using three different subproblems obtained by imposing constraints to the input instances I of $\mathcal{TT}^a$. In detail, the following constraints to functions L and w are imposed:

I_1: L is constant;
I_2: L and w are constant;
I_3: w is constant.

Note that $I_2 \equiv I_1 \cap I_3$ and $I_1 \cup I_3 \subseteq I$.

In the remainder of this section we use the following notation. The minimum and maximum values assigned by the function w (L, resp.) with respect to all the possible instances of $\mathcal{TT}^a$ are denoted by w_{min} and w_{max} (L_{min} and L_{max}, resp.). The maximum out degree of a DAG is denoted by deg. Moreover, given $i = (G, L, w) \in I$, $\Delta \in \mathbb{N}$ and $\alpha \in \mathbb{N}$, we denote

$$\gamma(\Delta) = 1 + \frac{\alpha}{(k+1)L_{min}} \quad \text{where} \quad \sum_{p=0}^{k-1} \deg^p \leq \Delta < \sum_{p=0}^{k} \deg^p \quad \text{for some } k \geq 0.$$

The following theorem shows the price of robustness of $\mathcal{TT}^a$ when the input instances are those of I_1.

Theorem 7 ([9]). *If L is constant, then $P_{rob}(\mathcal{TT}^a) \geq \gamma(\Delta)\frac{w_{min}}{w_{max}}$ for any fixed Δ.*

In [9] it has been proposed to use a proportional slack time of size $\gamma(\Delta)$ for each activity of a given instance, that is, algorithm $\mathsf{Alg}^*_{\gamma(\Delta)}$ has been used.

The following theorem shows that algorithm $\mathsf{Alg}^*_{\gamma(\Delta)}$ is robust for any possible Δ.

Theorem 8 ([9]). *For any $\Delta \geq 0$, $\mathsf{Alg}^*_{\gamma(\Delta)}$ is robust for $\mathcal{TT}^a$.*

The following theorem and corollary show the price of robustness of $\mathsf{Alg}^*_{\gamma(\Delta)}$ when the input instances are in I and that $\mathsf{Alg}^*_{\gamma(\Delta)}$ is $\mathcal{TT}^a$-optimal when the input instances are those of I_2.

Theorem 9 ([9]). *For any fixed $\Delta \geq 0$, $P_{rob}(\mathcal{TT}^a, \mathsf{Alg}^*_{\gamma(\Delta)}) \leq \gamma(\Delta)\frac{w_{max}}{w_{min}}$.*

Corollary 2 ([9]). *If w and L are constant, then $P_{rob}(\mathcal{TT}^a, \mathsf{Alg}^*_{\gamma(\Delta)}) = \gamma(\Delta)$, and $\mathsf{Alg}^*_{\gamma(\Delta)}$ is $\mathcal{TT}^a$-optimal.*

In [9] an algorithm $\mathsf{Alg}^+_{\alpha(\Delta)}$ which uses an additive constant slack time which depends on Δ instead of a proportional one has been proposed. It has been shown that $\mathsf{Alg}^+_{\alpha(\Delta)}$ returns the same timetable as $\mathsf{Alg}^*_{\gamma(\Delta)}$ when the input instances are those of I_1 and that $\mathsf{Alg}^+_{\alpha(\Delta)}$ is $\mathcal{TT}^a$-optimal when the input instances are those of I_3. When the input instances are those of the whole I, other than one would expect, in some cases the timetable returned by $\mathsf{Alg}^*_{\gamma(\Delta)}$ costs less than that returned by $\mathsf{Alg}^+_{\alpha(\Delta)}$.

Results for linear graphs. In [9] the attention also has been concentrated on linear graph with no restrictions imposed on functions L and w. In this case, a dynamic programming algorithm of time complexity $O(\Delta n)$ which is $\mathcal{TT}^a$-optimal has been given.

5 Multi-stage recoverable robustness and application to timetabling

As shown so far, the recoverable robustness model represents a significant improvement in the optimization area. It nevertheless has the following drawback: in many applications, one is typically not facing only one disturbing event, but several disturbances $i^1, i^2, \ldots, i^\sigma$ may occur. For example, assume that we expect at most two disturbances i^1 and i^2. In this case, a robust solution for i^1 should be also recoverable against the next disturbance i^2. This means that under all solutions which are robust for i^1, we should choose one that is again robust against the next disturbance i^2 (if it exists). This example can be extended to more than two disturbances, see Figure 7 for an illustration.

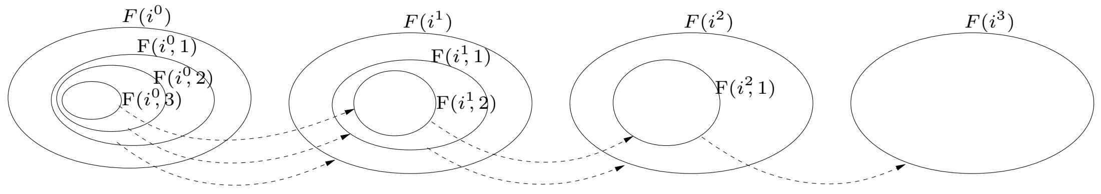

Fig. 7. The set of solutions that are recoverable against 1, 2, and 3 disturbances. Symbol $\mathrm{F}(i, n)$ is used to denote the set of feasible solutions for a problem which has to be solved against n disturbances. Dotted arrows represent recovery algorithms.

In this section, we survey results in [10, 11] concerning an extension of the recoverable robustness model (see Section 2) to the *multi-stage recoverable robustness* model which takes into account more than one disturbance and recovery stage.

To report the new model, some notation has to be recalled. Also in the multi-stage case, the task is to introduce robustness to an arbitrary minimization problems P. Problem P is characterized by the parameters I (set of instances), $F(i)$ for each $i \in I$ (feasible solutions), and f (objective function). The goal of a multi-stage recoverable robust problem is to find a *robust solution* for some given initial instance $i \in I$ of P. It is hence assumed again that a modification function M and a class $\mathbb{A}_{rec}$ of recovery algorithms for P are given. A_{rec} is defined as $A_{rec} : S \times I \to S$, where the initial feasible solution s^0 (of the initial undisturbed instance i^0) and the first modification $i^1 \in M(i^0)$ define the *minimal* amount of information necessary to recompute the solution. However, A_{rec} can also require additional information. In particular, when A_{rec} is used in the k-th stage, it can use everything that has been processed in the previous stages (in particular $i^0, ..., i^{k-1}$ and $s^0, ..., s^{k-1}$). Concerning possible classes of recovery algorithms, all the classes defined in Section 2 can be also used for multi-stage recoverable robustness. Additionally, the value $\sigma \in \mathbb{N}$ is used to denote the maximum number of expected modifications.

In [10, 11], the multi-stage recoverable robustness model has been introduced according to the following definitions.

Definition 8. *A* multi-stage recoverable robustness problem *is defined by* $(P, M, \mathbb{A}_{rec}, \sigma)$ *where* $(P, M, \mathbb{A}_{rec}) \in \mathrm{RRP}$ *and* $\sigma \in \mathbb{N}$*. The class* $\mathrm{RRP}(\sigma)$ *contains all the multi-stage recoverable robustness problems, that is, the recoverable robustness problems that have to be solved against* $\sigma \geq 1$ *possible disturbances.*

Definition 9. *Let* $\sigma \in \mathbb{N}$ *and* $\mathcal{P} = (P, M, \mathbb{A}_{rec}, \sigma)$ *be an element of* $\mathrm{RRP}(\sigma)$*. Given an instance* $i^0 \in I$ *for* P*,* s^0 *is a feasible solution for* i^0 *with respect to* $\mathcal{P}$ *if and only if the following relationship holds:*

$$\exists A_{rec} \in \mathbb{A}_{rec} : \quad s^0 \in F(i^0) \tag{6}$$

$$s^k := A_{rec}(s^{k-1}, i^k) \in F(i^k), \ \forall i^k \in M(i^{k-1}), \forall k \in [1..\sigma]. \tag{7}$$

This definition ensures that for each stage k, for any possible modification $i^k \in M(i^{k-1})$, and for any feasible solution s^{k-1} computed in the previous stage, the output s^k of algorithm A_{rec} is a feasible solution for i^k with respect to $\mathcal{P}$. If it is clear to which problem P, M and $\mathbb{A}_{rec}$ we refer to, we also say in short that s^0 is *feasible for* i *with respect to* σ *recoveries.*

Notice that $\mathrm{RRP}(1) = \mathrm{RRP}$. Hence, each problem in $\mathrm{RRP}(1)$ is called a *single-stage problem* and each problem in $\mathrm{RRP}(\sigma)$, $\sigma > 1$, is called a *multi-stage problem.*

Using the definition of a feasible solution, the robust algorithm that is used to compute the initial solution s^0 for the initial (undisturbed) instance i^0 is defined in the following.

Definition 10. *Given* $\mathcal{P} = (P, M, \mathbb{A}_{rec}, \sigma) \in \mathrm{RRP}(\sigma)$*, a* multi-stage robust algorithm *for* $\mathcal{P}$ *is any algorithm* $A_{rob} : I \to S$ *such that for each* $i \in I$*,* $A_{rob}(i)$ *is feasible for* i *with respect to* P*, i.e., such that* A_{rob} *outputs a solution that can be recovered against* σ *disturbances.*

Notice that in the case of strict robustness, a robust algorithm A_{rob} for $\mathcal{P}$ must provide a solution s^0 for i^0 such that for each possible modification $i^k \in M(i^{k-1})$, we have $s^0 \in F_{\mathcal{P}}(i^k)$ for all $k \in [1..\sigma]$. The meaning is the following: If A_{rec} has no recovery capability, then A_{rob} has to find solutions that "absorb" *any* possible sequence of disturbances.

Analogous to the single-stage case, the price of robustness has also been defined for multi-stage recovery algorithms. The following definition differs from the corresponding definition for the single-stage case only in the problem $\mathcal{P}$ (it now belongs to the larger class $\mathrm{RRP}(\sigma)$ instead of RRP).

Definition 11. *Let* $\mathcal{P} \in \mathrm{RRP}(\sigma)$*,* $\sigma \geq 1$*, and* A_{rob} *be a robust algorithm for* $\mathcal{P}$*. Then:*

- *The* price of robustness *of* A_{rob} *is*

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

- *The* price of robustness *of* $\mathcal{P}$ *is given by*

$$P_{rob}(\mathcal{P}) = \min\{P_{rob}(\mathcal{P}, A_{rob}) : A_{rob} \textit{ is a robust algorithm for } \mathcal{P}\}.$$

– A_{rob} *is* $\mathcal{P}$-optimal *if* $P_{rob}(\mathcal{P}, A_{rob}) = P_{rob}(\mathcal{P})$.
– A_{rob} *is* exact *if* $P_{rob}(\mathcal{P}, A_{rob}) = 1$.

We report a first, but important observation concerning the price of robustness:

Lemma 5 ([10, 11]). *For fixed P, M and $\mathbb{A}_{rec}$, consider a family of problems $\mathcal{P}_\sigma = (P, M, \mathbb{A}_{rec}, \sigma)$ for different values of σ, i.e., these problems vary in the expected number of recoveries only. For $\sigma_1 < \sigma_2$, the following holds:*

– *$F_{\mathcal{P}_{\sigma_2}}(i) \subseteq F_{\mathcal{P}_{\sigma_1}}(i)$ for all instances $i \in I$,*
– *$P_{rob}(\mathcal{P}_{\sigma_1}) \leq P_{rob}(\mathcal{P}_{\sigma_2})$, i.e., the price of robustness grows in the number of expected recoveries.*

The goal when discussing multi-stage recovery algorithms is to analyze how $P_{rob}(\mathcal{P}_\sigma)$ increases in σ, i.e., what it costs to establish a solution which is robust with respect to σ recoveries. In the next section we present such an analysis for the application of timetabling.

5.1 An application: Timetabling

An example of real world systems where the multi-stage model plays an important role is the timetable problem TT introduced in Section 4. In this problem, *many* unforeseen delays (caused by disturbing events such as bad weather conditions, repair work, signaling problems, or accidents) might occur during the operational phase, and they might completely change the schedule. In order to be able to deal with more than one delay, it is possible to consider the multi-stage recoverable robust version of the timetabling problem. To this end, it is necessary to formalize the modification function M and the class $\mathbb{A}_{rec}$ as follows:

– Given an instance $i^{k-1} = (G, w, L^{k-1})$ and a constant $\alpha \in \mathbb{R}^{>0}$, then

$$M(i^{k-1}) = \left\{ (G, w, L^k) : \exists\ \overline{a} \in A : L^0_{\overline{a}} \leq L^k_{\overline{a}} \leq L^0_{\overline{a}} + \alpha \text{ and } L^k_a = L^{k-1}_a\ \forall a \neq \overline{a} \right\},$$

i.e., one additional delay (whose size is bounded by α) is allowed in every stage k.
– In general, it is interesting to use *Class 2* with the two limitations specified next. Let π be a solution of TT and consider a recovery algorithm A_{rec}. Let x be the recovered timetable computed by A_{rec} with input π and $i^k \in M(i^{k-1})$.

  limited-events: This class of recovery algorithm is denoted by $\mathbb{A}_\Delta$ and contains the algorithm A_{rec} if and only if
  $$|\{u \in V : x_u \neq \pi_u\}| \leq \Delta,$$
  for all $x = A_{rec}(\pi, i^k)$, for any feasible disturbance $i^k \in M(i^{k-1})$ of step k with $k = 1, \ldots, \sigma$. Informally, for each recovery stage, it is required that the scheduled times of only a limited number of events might be changed during the recovery with respect to π. Notice that this class has been already defined in Section 4 where it has been used for the single-stage case.

  limited-delay: This class of recovery algorithm is denoted by $\mathbb{A}_\delta$ and contains the algorithm A_{rec} if and only if
  $$\|x - \pi\|_1 \leq \delta,$$
  for all $x = \mathbb{A}_{rec}(\pi, i^k)$, for any feasible disturbance $i^k \in M(i^{k-1})$ of step k with $k = 1, \ldots, \sigma$. Informally, $\mathbb{A}_\delta$ contains all polynomial algorithms producing a solution for which the sum of all delays is less than or equal to δ.

By using the multi-stage recoverable robustness model, the following problems in RRP(σ) have been investigated:

- $\mathcal{TT}^v_\sigma = (TT^v, M, \mathbb{A}_\Delta, \sigma)$ where TT^v denotes the timetabling problem TT with nonnegative weights w_u. Notice that studying $\mathcal{TT}^v_\sigma$ corresponds to investigating the problem $\mathcal{TT}^v$ (see Section 4) with respect to σ disturbances instead of just one.
- $\mathcal{TT}^a_\sigma = (TT^a, M, \mathbb{A}_\delta, \sigma)$ where TT^a denotes the timetabling problem defined for the special case f_{arcs} with nonnegative weights w_a (see Section 4).

The general approach of using the algorithm Alg^+_s (with the same slack s for all the activities) has been adopted to face both the previous problems. Of course, in this scenario, the main task is to find the smallest value for s such that Alg^+_s is robust. Finding a bound for the price of robustness of Alg^+_s is important as well.

In the following we summarize the results obtained in [10]. See Table 5 for an overview on upper bounds on the price of robustness for some special cases. As mentioned before, it is the main goal to analyze how the price of robustness increases in the number σ of expected recoveries. See Figure 8 for one special example.

In the following paragraphs, we will summarize the results obtained in [10] for both sub-problems $\mathcal{TT}^v_\sigma$ and $\mathcal{TT}^a_\sigma$:

Problem $\mathcal{TT}^v_\sigma$: When considering the number of affected events as limitation, the following observations are easily obtained:

- π is robust if $\Delta \geq |V| - 1$.
- Let $\sigma > \Delta$. Then the following holds:
  π is robust $\iff s_a \geq \alpha$ for all $a \in A \iff \pi$ is strictly robust.
- Let $\Delta = 0$:
  π is robust for $\sigma = 1 \iff \pi$ is robust for $\sigma > 1$.
- The set of robust solutions w.r.t $\sigma > 1$ is strictly contained in the set of robust solutions for $\sigma = 1$.

However, the problem of calculating the price of robustness with number of events as limitation is NP-hard for all fixed $\Delta \geq 3$ even for the case $\sigma = 1$ which corresponds to the problem faced in Section 4. Table 4 summarizes the computational complexity of calculating an optimal robust solution for $\mathcal{TT}^v_\sigma$ with respect to different graph topologies.

Problem	graph	σ	Δ	Complexity
$\mathcal{TT}^v_\sigma$	arbitrary	1	≥ 3	NP-hard
$\mathcal{TT}^v_\sigma$	linear	any	any	linear

Table 4. Computational complexity of calculating an optimal robust solution.

Notice that, in the case of a sequence of $\sigma \geq 1$ disturbances, it has been proposed a solution for $\mathcal{TT}^v_\sigma$ for arbitrary σ when the underlying graph is a linear graph. As mentioned above, the solution uses the algorithm $\mathsf{Alg}^+_{s^*}$ assigning the same slack

$$s^* = \min\left\{\alpha, \frac{\sigma\alpha}{\Delta + 1}\right\} \tag{8}$$

to each arc. The following result has been obtained:

Theorem 10 ([10, 11]). *Let s^* be defined as in Eq. (8), and let G be a linear graph. Then:*

- Alg_s^+ *is a robust algorithm if and only if* $s \geq s^*$*;*
- $P_{rob}(\mathcal{TT}_\sigma^v, \mathsf{Alg}_{s^*}^+) \leq 1 + \frac{s^*}{L_{min}}$*;*
- $\mathsf{Alg}_{s^*}^+$ *is optimal compared to all robust algorithms that add an equal slack s to all arcs.*

Problem $\boldsymbol{\mathcal{TT}_\sigma^a}$: Also for this problem, in [10] the algorithm Alg_s^+ (with the same slack s for all the activities) has been used to solve $\mathcal{TT}_\sigma^a$. The following properties hold:

- Alg_α^+ is strictly robust for all σ;
- $\mathsf{Alg}_{\alpha-\Delta}^+$ is robust for $\sigma = 1$ and $\Delta \leq \frac{\alpha}{2}$.

If G is a tree and Alg_s^+ is robust, the price of robustness can be bounded by $P_{rob}(\mathcal{TT}_\sigma^a, \mathsf{Alg}_s^+) \leq 1 + \frac{s}{L_{\min}}$ where the bound is obtained in the case of strict robustness. This leads to the results in Table 5.

For arbitrary σ and Δ, it has been shown that there exists some s^* such that Alg_s^+ is robust for all $s \geq s^*$. In linear graphs this value s^* can be calculated as

$$s^* = \frac{2\sigma\alpha\left(\left\lceil\frac{2\Delta}{\sigma\alpha}\right\rceil + \sigma\right) - \sigma\alpha(\sigma+1) - 2\Delta}{\left(\left\lceil\frac{2\Delta}{\sigma\alpha}\right\rceil + \sigma\right)\left(\left\lceil\frac{2\Delta}{\sigma\alpha}\right\rceil + \sigma - 1\right)},$$

leading to a price of robustness equal to $1 + s^*$.

In conclusion, the price of robustness can be exactly calculated in special cases only; in many cases, an approximation is possible. It is clear that the price of robustness increases with the number of expected recoveries, but its growth is smaller than expected as can be seen in Figure 8.

problem	graph	σ	Δ	P_{rob}
$\mathcal{TT}_\sigma^v$	DAG	1	any	$1 + \frac{\alpha}{L_{\min}}$
$\mathcal{TT}_\sigma^v$	tree	1	any	$1 + \min\left\{\frac{\alpha}{L_{\min}}, \frac{\alpha}{2}\right\}$
$\mathcal{TT}_\sigma^v$	linear	any	any	$1 + \frac{1}{L_{\min}} \min\left\{\alpha, \frac{\sigma\alpha}{\Delta+1}\right\}$
$\mathcal{TT}_\sigma^a$	tree	any	any	$1 + \frac{\alpha}{L_{min}}$
$\mathcal{TT}_\sigma^a$	tree	1	$\Delta \leq \frac{\alpha}{2}$	$1 + \frac{\alpha-\Delta}{L_{min}}$
$\mathcal{TT}_\sigma^a$	linear	any	any	$1 + \frac{2\sigma\alpha\left(\left\lceil\frac{2\Delta}{\sigma\alpha}\right\rceil+\sigma\right)-\sigma\alpha(\sigma+1)-2\Delta}{\left(\left\lceil\frac{2\Delta}{\sigma\alpha}\right\rceil+\sigma\right)\left(\left\lceil\frac{2\Delta}{\sigma\alpha}\right\rceil+\sigma-1\right)}$

Table 5. Upper bounds for P_{rob} in some special cases.

6 Conclusions

The chapter surveys on some recent algorithmic results achieved within the recoverable robustness model. This model provides the unification between the standard notion of robustness (a solution must remain feasible although some disturbances may occur) and the possibility to apply limited recovery strategies once the feasibility of the current solution is lost. The attention has been addressed to two main problems arising in the area of railway optimization: the shunting problem and the timetabling problem. The former problem regards the reordering of freight train cars over hump yards while the latter one consists in finding passenger train timetables in order to minimize the overall passengers traveling time. In the reviewed papers, algorithms for

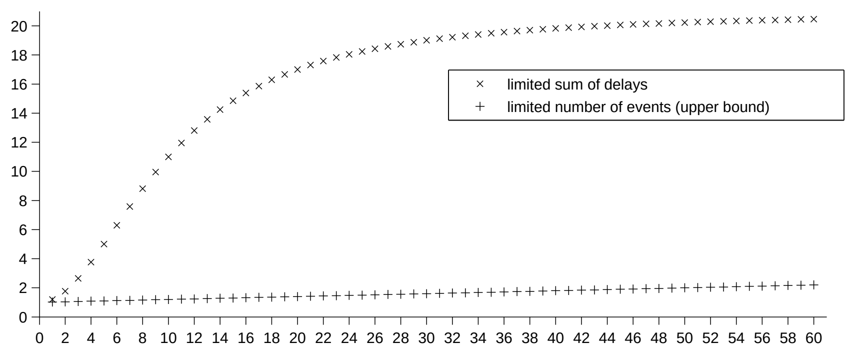

Fig. 8. The price of robustness of algorithm Alg_s^+ for $\alpha = 20$ and $\Delta = 1000$ as a function of σ.

both problems have been investigated with respect to different possible disturbances and different recovery capabilities. Moreover, the timetabling problem has been also used as a testbed for empirical experiments and for investigating on a natural extension of the original model, i.e. the so called multi-stage recoverable robustness. This extension aims to provide recoverable robustness in the case that multiple recovery phases are allowed.

The presented results reveal an interesting and practical applicability of the model under which useful algorithms have been designed in the studied contexts. The proposed notion of recoverable robustness provides a mean to compare the performances of different robust algorithms in terms of distance from the optimality. In particular, the price of robustness for an algorithm measures such a distance and provides a practical tool to rank among algorithms. This implies the possibility to apply standard techniques developed in algorithmic theory for choosing the "best" algorithm, robust with respect to the required constraints. Moreover, the presented experimental results confirm the effectiveness for the defined price of robustness. In fact, the evaluations show how the algorithmic performances are affected by the variation on both the recovery capabilities and the modification function. As theoretically expected, the less restrictive the available recovery capabilities are, the smaller the price of robustness of an algorithm is. Viceversa, the larger the set of modification is, the larger the price of robustness of an algorithm is. Both observations can be seen in Figure 6 where the price of robustness decreases with Δ and increases with α.

The reported results represent some initial contributions to the applicability of the recoverable robustness model. It might be worth investigating the feasibility of the model on different optimization problem, varying on modification functions and the class of recovery algorithms. Concerning this last point, an important issue that has not yet been addressed concerns the investigation of robust algorithms when the allowed recovery capabilities are that of *Class 3*, i.e. when the computational power of the recovery algorithms is bounded.

Extensions to the proposed model are also of interest. In particular, it is worth to note that the proposed notion of recoverable robustness does not enforce the requirement to design efficient robust algorithms in time and space. This is a key point in introducing recoverable robustness in most of the problems studied in the context of railway optimization as well as in the more general algorithmic optimization turn out to be NP-hard. Hence, the optimal algorithm on which the price of robustness depends might be exponential due to the computational complexity of the underlying robust problem. Then, it would be interesting to study the case in which the computational power of the robust algorithms is limited and hence to define the price of robustness of the problem according to the admitted class of robust algorithms.

Another interesting direction of research for the extension of the model is about the design of robust and recovery algorithms. Consider the typical scenario in the context of recoverable robustness: it is necessary to face a problem $\mathcal{P} = (P, M, \mathbb{A}_{rec})$ and, to this aim, in the operational phase, a plan s (computed by a robust algorithm) is being used. At this time, if a disturbing event $j \in M(i)$ occurs, the recoverable robustness model guarantees that an algorithm $A_{rec} \in \mathbb{A}_{rec}$ exists to *recover*, that is, to compute a new plan $s' = A_{rec}(s, j)$ which is feasible for j.

Notice that, the recoverable robustness model ensures only that A_{rec} exists, and it does not take care of defining details about such a recovery algorithm. In an operative environment, such details are needed, and hence a recovery algorithm has to be carefully designed. In this context, relevant questions are: How to measure the quality of a given recovery algorithm? When to design recovery algorithms? Concerning the first question, a method could be to define a sort of *price of recovery*. Concerning the other question, in general, a robust algorithm (planning stage) and a recovery algorithm (operational stage) can be designed independently of each other. The main motivation for this lies on the fact that $\mathbb{A}_{rec}$ can be defined by means of just some mathematical properties, and hence, the input of each recovery algorithm is completely defined, regardless of the definition of any possible robust algorithm. This leads to two possible scenarios:

- Considering the robust and recovery algorithms as belonging to different modules of the same system. If A_{rec} is designed *a priori*, i.e., independently of any information concerning robust algorithms, then the robust and recovery modules are decoupled. This guarantees that, in case, each module can be re-engineered without affecting the whole system.
  In this scenario, a possible definition for a price of recovery could be the following:

  Definition 12. *Let $\mathcal{P} = (P, M, \mathbb{A}_{rec})$ be a problem in* RRP*, and let $A_{rec} \in \mathbb{A}_{rec}$. The* price of recovery *of A_{rec} is:*

  $$P_{rec}(\mathcal{P}, A_{rec}) = \max_{i \in I,\, s \in F_{\mathcal{P}}(i),\, j \in M(i)} \left\{ \frac{f(A_{rec}(s, j))}{\min\{f(x) : x \in F(j)\}} \right\}.$$

  This price of recovery measures "how far" the recovered solution computed by A_{rec} is away from the optimum one, independently of which robust algorithm has been used.
  Unfortunately, this measure could be useless in most cases. For instance, consider the robustness problem $\mathcal{TT}^v$ for linear graphs defined at the end of Section 4.1 where each algorithm in the class $\mathbb{A}_{rec}$ can change the time of at most Δ events. In this case, it is easy to see that the price of recovery is unbounded since in $F_{\mathcal{P}}(i)$, for $i \in I$, there are solutions s which are robust due to slack times arbitrarily large.
- Designing a recovery algorithm A_{rec} in hindsight (i.e., after the definition of a *good* robust algorithm A_{rob}) could imply the design of a *specialized* recovery algorithm, since A_{rec} could exploit specific properties of the robust solutions computed by A_{rob}.
  In this scenario, a possible definition for the price of recovery could be the following:

  Definition 13. *Let $\mathcal{P} = (P, M, \mathbb{A}_{rec})$ be a problem in* RRP*, A_{rob} be a robust algorithm for $\mathcal{P}$, and $A_{rec} \in \mathbb{A}_{rec}$. The* price of A_{rob}-recovery *of A_{rec} is:*

  $$P_{rec}(\mathcal{P}, A_{rob}, A_{rec}) = \max_{i \in I,\, j \in M(i)} \left\{ \frac{f(A_{rec}(A_{rob}(i), j))}{\min\{f(x) : x \in F(j)\}} \right\}.$$

  This measure could represent a good parameter for the quality of a recovery algorithm.

Additionally, notice that a recovery algorithm A_{rec} with a small price of recovery is useless when its traditional *worst case execution time* is bad. For instance, if we consider the timetabling problem TT along with the class $\mathbb{A}_{\Delta}$ (remember, any algorithm in $\mathbb{A}_{\Delta}$ can change the time of at most Δ events), then we expect that an algorithm in $\mathbb{A}_{\Delta}$ should be computationally efficient, that is, running in $O(\Delta)$ time. Hence, the quality of a recovery algorithm should be measured by means of two parameters: the price of recovery and the worst case execution time. A trade-off between these two parameters could define a *good recovery algorithm*.

References

1. H. G. Bayer and B. Sendhoff. Robust Optimization - A Comprehensive Survey. *Computer Methods in Applied Mechanics and Engineering*, 196(33-34):3190–3218, 2007.
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. J. Birge and F. Louveaux. *Introduction to Stochastic Programming.* Springer Verlag, New York, 1997.
5. A. Borodin and R. El-Yaniv, editors. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.
6. S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, and A. Navarra. Robust Algorithms and Price of Robustness in Shunting Problems. In *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS07)*, pages 175–190. Schloss Dagstuhl Seminar proceedings, 2007.
7. S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, and A. Navarra. Robust algorithms and price of robustness in shunting problems. Technical Report ARRIVAL-TR-0072, ARRIVAL project, 2007.
8. S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, and A. Navarra. Delay management problem: Complexity results and robust algorithms. In *Proceedings of the 2nd Annual International Conference on Combinatorial Optimization and Applications (COCOA'08)*, volume 5165 of *Lecture Notes in Computer Science*, pages 458–468, 2008.
9. S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, and A. Navarra. Recoverable robust timetabling: Complexity results and algorithms. Technical Report ARRIVAL-TR-0172, ARRIVAL project, 2008.
10. S. Cicerone, G. Di Stefano, M. Schachtebeck, and A. Schöbel. Dynamic algorithms for recoverable robustness problems. In *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS08).* Schloss Dagstuhl Seminar proceedings, 2008.
11. S. Cicerone, G. Di Stefano, M. Schachtebeck, and A. Schöbel. Dynamic algorithms for recoverable robustness problems. Technical Report ARRIVAL-TR-0130, ARRIVAL project, 2008.
12. G. D'Angelo, G. Di Stefano, and A. Navarra. Recoverable robust timetables on trees. Technical Report ARRIVAL-TR-0163, ARRIVAL project, 2008.
13. G. D'Angelo, G. Di Stefano, and A. Navarra. Recoverable-robust timetables for trains on single-line corridors. In *Proceedings of the 3rd International Seminar on Railway Operations Modelling and Analysis - Engineering and Optimisation Approaches (RailZurich2009)*, 2009. To appear.
14. L. De Giovanni, G. Heilporn, and M. Labbé. Optimization models for the delay management problem in public transportation. *European Journal of Operational Research*, 189(3):762–774, 2007.
15. 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.
16. M. Fischetti and M. Monaci. Light robustness. Technical Report ARRIVAL-TR-0066, ARRIVAL project, 2008.
17. M. Gatto, B. Glaus, R. Jacob, L. Peeters, and P. Widmayer. Railway delay management: Exploring its algorithmic complexity. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory Algorithm Theory (SWAT 2004)*, volume 3111 of *Lecture Notes in Computer Science*, pages 199–211. Springer, 2004.
18. M. Gatto, R. Jacob, L. Peeters, and A. Schöbel. The Computational Complexity of Delay Management. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 3787 of *Lecture Notes in Computer Science*, pages 227–238, 2005.
19. M. Gatto, R. Jacob, L. Peeters, and P. Widmayer. Online Delay Management on a Single Train Line. In *Proceedings of the 4th Workshop on Algorithmic Methods for Railway Optimization (ATMOS04)*, volume 4359 of *Lecture Notes in Computer Science*, pages 306–320, 2007.
20. A. Ginkel and A. Schöbel. The bicriteria delay management problem. *Transportation Science*, 41(4):527–538, 2007.
21. R. S. Hansman and U. T. Zimmermann. Optimal sorting of rolling stock at hump yard. In *Mathematics - Key Technology for the Future: Joint Project Between Universities and Industry*, pages 189–203. Springer, 2008.
22. R. Jacob. On shunting over a hump. Technical Report 576, Institute of Theoretical Computer Science, ETH Zürich, 2007.

23. 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 (ATMOS07)*, pages 158–174. Schloss Dagstuhl Seminar proceedings, 2007.
24. R. Jacob, P. Marton, J. Maue, and M. Nunkesser. Multistage methods for freight train classification. Technical Report ARRIVAL-TR-0140, ARRIVAL project, 2007.
25. P. Kall and S. Wallace. *Stochastic Programming*. Wiley, Chichester, 1994.
26. F. Levy, G. Thompson, and J. Wies. *The ABCs of the Critical Path Method*. Graduate School of Business Administration. Harvard University, 1963.
27. C. Liebchen, M. Lüebbecke, R. H. Möhring, and S. Stiller. Recoverable robustness. Technical Report ARRIVAL-TR-0066, ARRIVAL project, 2007.
28. C. Liebchen and S. Stiller. Delay resistant timetabling. *Public Transport*, 2008. To appear. Tech. Rep. ARRIVAL-TR-0056, ARRIVAL project.
29. A. Ruszczynski and A. Shapiro, editors. *Stochastic Programming, Handbooks in Operations Research and Management Science Volume 10*. North-Holland, 2003.
30. A. Schöbel. A model for the delay management problem based on mixed integer programming. *Electronic Notes in Theoretical Computer Science*, 50(1), 2001.
31. A. Schöbel. Integer programming approaches for solving the delay management problem. In *Proceedings of the 4th Workshop on Algorithmic Methods for Railway Optimization (ATMOS04)*, volume 4359 of *Lecture Notes in Computer Science*, pages 145–170, 2007.