Iterated Greedy Algorithms for a Real-world Cyclic Train Scheduling Problem

Zhi Yuan[1], Armin Fügenschuh[2], Henning Homfeld[2], Prasanna Balaprakash[1], Thomas Stützle[1], and Michael Schoch[3]

[1] IRIDIA-CoDE, Université Libre de Bruxelles (ULB), Brussels, Belgium
{zyuan,pbalapra,stuetzle}@ulb.ac.be
[2] Arbeitsgruppe Optimierung, Fachbereich Mathematik, Technische Universität Darmstadt, Darmstadt, Germany
{fuegenschuh,homfeld}@mathematik.tu-darmstadt.de
[3] Deutsche Bahn AG, Frankfurt am Main, Germany
michael.schoch@bahn.de

Abstract. In this paper, we develop heuristic algorithms for a complex locomotive scheduling problem in freight transport that arises at Deutsche Bahn AG. While for small instances an approach based on an ILP formulation and its solution by a commercial ILP solver was rather successful, it was found that effective heuristic algorithms are needed for providing better initial upper bounds and for tackling large instances. The main contribution of this paper is the development of heuristic algorithms that strongly improve over the performance of the greedy algorithm used in the previous research efforts. The development process was done on a step-by-step basis ranging from improvements over the initial greedy construction heuristic, the development of a simple local search algorithm, the further extension to an iterated greedy procedure to the adoption of population-based stochastic local search methods. Our computational results show that the iterated greedy algorithm combined with a simple local search is a powerful algorithm for this real-world freight train scheduling problem.

1 Introduction

The problem we are tackling in this paper arises in the strategic planning of the Deutsche Bahn AG (DB), the largest railway company in Germany. In particular, the problem arises in the context of a complex simulation tool that is used at DB to provide long-term simulations and future predictions of the load of the railway network. The tool can be seen as a chain of modules, where information between the modules is exchanged through data files.

Our particular problem arises in a module called *train scheduler*, which is responsible for the buildup of trains from cars. A train starts as soon as it is built, that is, when enough cars are assembled. Hence, this also means that the starting times of the trains do not follow a specific timetable; rather they follow the estimated customers demand or production. Since the locomotives

are among the most expensive resources of the operation of railroad companies, their efficient scheduling is of high importance.

The locomotive scheduling, with which we deal here, can be characterized as a vehicle routing problem with time windows, a heterogeneous fleet of vehicles (due to different types of locomotives), and cyclic departures of trains. It also includes two important additional aspects: network-load dependent travel times and the transfer of cars between trains. In earlier work [1], we have developed an integer linear programming (ILP) formulation of the problem and proposed a solution approach for the problem using a commercial ILP solver (ILOG Cplex 10 [2]). In the corresponding experimental campaign [1], we noticed that (i) the minimum number of missed car transfers is relatively easy to find, (ii) with fixed starting times, relatively large instances with up to about 1 500 trips could be solved to optimality, (iii) the models that allowed the flexible choice of starting times within some predefined time windows made the problem much more difficult to solve: the size of the instances that could successfully be tackled by the commercial ILP solver (after some additional improvements such as providing good initial feasible solutions and problem-specific cutting planes) was limited to medium sized instances with a few hundreds of trips [1]. Despite the added computational difficulty, the hardest model with flexible starting times has highly desirable properties: allowing to vary the starting times within small time windows results in a considerable reduction of the required number of locomotives and, hence, a strong reduction in the total costs [1].

We tackle the most difficult problem variant studied in [1], namely the one that uses time windows and network-load dependent travel times. For this variant, which is also the most realistic and interesting one, heuristic algorithms are required to generate good quality solutions to large instance sizes, but also ILP approaches can benefit from improved initial upper bounds for medium sized instances. In this paper, we therefore report on our research for improving upon the performance of the greedy construction heuristics that have been used in [1]. Our development process departs from these greedy heuristics, extending them step-by-step. The first extensions comprise a direct modification of the greedy heuristic by changing the way solutions are constructed. Next, we extend the construction heuristic to an iterated greedy (IG) algorithm [3] and further hybridize it with a simple iterative improvement algorithm. Our experimental results clearly show that for computation times ranging from a few seconds to a few minutes on current CPUs, very strong improvements over the initial greedy heuristic can be obtained. A further hybridization of the IG algorithm with ant colony optimization (ACO) algorithms, however, gave rather mixed results and no further significant improvement in performance.

2 The freight train scheduling problem

It is convenient to first describe the nomenclature used in the context where this freight train scheduling problem arises.

2.1 Problem setting

Cars. A *car* is the smallest unit to be moved; cars have to be moved from a source to a destination within the railway network. A train is composed of a set of cars. Large customers require the transport of large amounts of goods so that they order whole trains; in such a case, the route of the cars is the same as the route of the train. Individual or small sets of cars are used by smaller customers. Several such cars are then assembled into a train, moved to some intermediate destination, which is called *shunting yard*, where trains are disassembled and reassembled into new trains. We assume that that the place and timing of these transfers is known for individual cars. Note that the scheduling of locomotives needs to take care that these transfers remain feasible.

Trains. A *freight train* (also called *active trip*) consists of several *cars*. Each train has a *start* and a *destination*, which are goods stations or railroad shunting yards, and *starting times* and *arrival times*. Times can be either fixed times or be taken from some interval. In our case, trips start cyclically every 24 hours. The *trip duration* is the difference between start and arrival times. Trains vary in lengths and weight and, depending on these two characteristics, different needs on the driving power of locomotives arise. Typically, a single locomotive is enough and only rarely two pulling locomotives are required. The handling of a train involves attaching a locomotive to a train at the start and the decoupling of the locomotive at the destination. For both *coupling* processes, some time (up to 30 minutes, depending on the trip) is required for technical checks or refueling.

Locomotives. The around 30 different models of locomotives that are used at DB share many similarities and so they can be classified into a small number of different *classes*. Differences between the classes concern mainly the driving power and the motor type, diesel or electrical. Electrical locomotives can only be used on electrical tracks, whereas diesel locomotives, in principle, can drive everywhere. However, diesel soots the electrical wires, so one wants to avoid their deployment on such tracks. Hence, it is only possible to assign such locomotives to trains that have a sufficient power and the right motor type for the track.

Deadheads. A locomotive is either active, that is, pulling a train, or deadheading, that is, driving alone, without pulling a train, from the destination station of one train to the start of another train. The distance between these points and the class of locomotive determine the duration of a deadhead trip.

2.2 Formulation of the problem

Our locomotive scheduling problem can be formulated as a cyclic Vehicle Scheduling Problem (CVSP), more specifically, as a CVSP with hard time windows and a heterogeneous fleet of vehicles (locomotives); we use the abbreviation CVSPTW in what follows. The locomotives differ in starting cost and capability, that is, the subset of customers that can be potentially served by the locomotives. Instead of starting and terminating at a depot as in standard vehicle routing problems, the locomotives are scheduled in a cyclic fashion every 24 hours. Given is also a set of customers, more specifically in our case a set of trips, that expect exactly

one of the locomotives to pull the train obeying restrictions on the starting and arrival times (or time windows).

More formally, the problem can be described as follows. We denote by $\mathcal{V}$ the set of active trips and by $\mathcal{B}$ the set of locomotive types. With each trip is associated a list of locomotive types which can serve it. $\mathcal{A} = \mathcal{V} \times \mathcal{V}$ denotes the set of potential connections of pairs of trips. Note that trips are *cyclic*, which for our problem means that the trips occur every day.

The time intervals for the starting and arrival times of trips are input data. In practice, the train scheduler of the tool chain defines fixed starting and arrival times. In our model, we let the starting time t_i of trip i vary in a time window $\underline{t}_i \leq t_i \leq \overline{t}_i$. The times are defined as the minutes passed since some time zero and, hence, one has that $0 \leq t_i \leq 1439$ (a day has 1440 minutes). Note that a further constraint imposes that a trip has to start on the first day; hence, in case $\overline{t}_i$ is larger than 1439, the time interval $\underline{t}_i \leq t_i \leq \overline{t}_i$ is replaced by two time intervals $([\underline{t}_i, 1439] \cup [0, \overline{t}_i]) \cap \mathbb{Z}$.

Moreover, trains $i, j \in \mathcal{V}$ that require a car transfer from one to the other need to be synchronized. Denote $\mathcal{P}$ the set of pairs (i, j) where cars transfer from i to j. We assume $\mathcal{P}$ is valid and given by the earlier computation module. For all $(i, j) \in \mathcal{P}$ the transferred car must be picked up by j within 12 hours after the arrival of i at the shunting yard.

Our model includes network-load dependent travel times. This is important since typically at day time the average traveling speed of a freight train is lower than during nighttime; the main reason is that passenger trains have a higher priority than most freight trains, leading to frequent waiting times for the latter. To model this aspect, we partition a day into a number of time slices $\mathcal{H} = \{1, \ldots, H\}$, that is, $[0, 1439] = \dot{\bigcup}_{h \in \mathcal{H}} [\underline{\psi}_h, \overline{\psi}_h]$. The starting time of a train then falls into one of the slices and the travel times are considered accordingly.

The total trip and deadhead durations $\delta_{b,(i,j)}$ for a locomotive of type b that serves both trips i and j are computed as

$$\delta_{b,(i,j)} := \delta_i^{trp} + \delta_i^{uncpl} + \delta_{b,(i,j)}^{dhd} + \delta_j^{cpl}, \tag{1}$$

where δ_i^{trp} denotes the trip duration, that is, the time the locomotive is active while pulling train i; δ_i^{uncpl} denotes the time for uncoupling the locomotive from the train at the arrival; $\delta_{b,(i,j)}^{dhd}$ denotes the time for deadheading from the end of i to the start of train j; δ_j^{cpl} denotes the time for coupling the locomotive to the train at the start of j.

Note that the driving time δ_i^{trp} is assumed to be independent of the actual locomotive class, whereas the deadhead time $\delta_{b,(i,j)}^{dhd}$ is class dependent (since diesel and electrical might use different routes). In the case of network-load dependent travel times, δ_i^{trp} gets replaced by $\delta_{i,h}^{trp}$, which gives the trip duration of train i when starting in slice h. Finally, in the case of car transfers between trains, the time for shunting a car from i to j, $\delta_{i,j}^{shnt}$, needs to be taken into account for the computation of the trip and deadhead durations.

The goal of the problem is to compute feasible starting and arrival times of the trains such that the operational costs are minimized. More concretely, two

cost components are considered. Let γ_b^{cls} be the cost for a locomotive of class b, $\gamma_{b,(i,j)}^{dhd}$ by the cost of a deadhead trip from i to j for locomotive class b. Typically, the following ordering holds between the costs: $\gamma_b^{cls} \gg \gamma_{b,(i,j)}^{dhd}$, that is, the most important objective is to reduce the number of used locomotives, and then at a subsidiary level, to minimize the total distance of all the deadhead trips.

A mathematical description of the constraints and objective as a mixed-integer programming model can be found in [1].

Finally, recall that in this paper we tackle the most difficult variant of the train scheduling problem from [1], that is, the variant with time windows (instead of fixed starting and arrival times) and with network-load dependent travel times, in part because the variant without these two features were well solved by the ILP approach. For the approach described in this paper, we do not consider the preliminary step of the minimization of the number of missed car transfers as in [1], but we fix the set of feasible car transfers as determined through some preliminary computation. The reason is that the minimum number of missed car transfers can, in practice, easily be done by an ILP approach or by a greedy heuristic [1]. Hence, we focus on the minimization of the operational costs, that is, the costs for the locomotive usage and the deadhead trips.

2.3 Benchmark instances

For our computational tests we used eight instances, five of which were also used as test instances in [1]. The names and the characteristics of these five instances are as follows.

- `A`, with 42 trains, 3 locomotive classes
- `B`, with 82 trains, 3 locomotive classes
- `C`, with 120 trains, 4 locomotive classes
- `KV`, with 340 trains, 6 locomotive classes
- `EW`, with 727 trains, 6 locomotive classes

From each of these instances, we obtain additional ones by imposing different ranges for the time windows. In particular, for each of the fixed starting and arrival times we allow symmetric time windows centered at these values from the set $\{\pm 0, \pm 10, \pm 30, \pm 60, \pm 120\}$ minutes.

An additional three instances have been used for tuning the parameters of the heuristic algorithms. The characteristics of these instances are as follows.

- `R1-101-3`, with 101 trains, 3 locomotive classes
- `R2-137-6`, with 137 trains, 6 locomotive classes
- `R3-295-5`, with 295 trains, 5 locomotive classes

Since we used iterated F-race [4], an automated tuning tool, it was desirable to derive a larger set of instances from these three. Hence, we selected the time windows as follows. For `R3-295-5`, `R2-137-6`, and `R1-101-3`, we selected as time windows the integers from $[0, 150]$ that can be divided by 2, 3, and 6, respectively; this results in 75, 50, and 25 instances, respectively. Note that the larger the base instance the more derived instances are available. This was done in order to bias the choice of the parameter settings towards better solving large instances.

3 Greedy algorithms

For effectively tackling the CVSPTW variant with network-load dependent travel times, an essential ingredient for the ILP-based approach [1] has been a randomized PGreedy-type algorithm [5]. This algorithm, called `g-CVSPTW`, forms the basis for the further developments and it is described next.

3.1 The `g-CVSPTW` heuristic

`g-CVSPTW` assumes a fixed set of car transfers as input and it seeks to minimize the operating costs. `g-CVSPTW` iteratively generates cyclic schedules for locomotives repeating the following series of steps until all trips are scheduled.

1. **Step 1: Locomotive selection.** First, a locomotive class is chosen. The choice is based on two factors: (i) the capability N_b, which is the number of trains that can be served by a locomotive from class b, and γ_b^{cls}. Each locomotive class is scored by the ratio N_b/γ_b^{cls}. A locomotive class with the highest ratio is chosen.
2. **Step 2: Start trip selection.** Among the still unscheduled trips, we select one with the highest number of deadhead trips that use a locomotive of the same class b as selected in step 1; ties are broken randomly.
3. **Step 3: Starting time selection.** Next, the starting time t_i of the trip is taken from the interval $[\underline{t}_i, \bar{t}_i]$. t_i is chosen such that the train arrival time is the earliest possible. (Recall that since we have netload dependent driving times, we need to subdivide the interval $[\underline{t}_i, \bar{t}_i]$ in dependence of the time slices.) After fixing the starting time, the impact of this decision on other starting time windows needs to be propagated. This update is necessary, if there is some trip j with (i, j) or $(j, i) \in \mathcal{P}$. For this task, a constraint propagation algorithm was implemented to propagate the effect of the decisions through the network.
4. **Step 4: Deadhead trip selection.** Next, a shortest deadhead trip to the start of the next train is chosen for the locomotive.

`g-CVSPTW` starts with steps 1 and 2 and then cycles through steps 3 and 4 until no trip can be added anymore to the locomotive's schedule. In particular, the construction is stopped if no trip can be accommodated within the 24 hours cycle (recall that `g-CVSPTW` uses a maximum 24 hour cycle length for each locomotive).

Instead of making deterministic choices in steps 1, 2, and 4, `g-CVSPTW` actually uses randomized greedy values, in spirit similar to the noising method [6]. Before doing the choice, the concerned greedy values are multiplied by a value that is generated randomly according to a uniform distribution in $[10000 * (1 - \mathit{NOISE}), 10000]$, where NOISE is a parameter from the range $[0, 1]$ that indicates the degree of noise in the random selection. Decisions are then taken deterministically based on the perturbed greedy values, breaking remaining ties randomly. The main rationale for using the randomization in the greedy heuristic is to allow the generation of a large set of different solutions.

3.2 Modified greedy heuristic

As a first step in the road to improved heuristic algorithms, we modified `g-CVSPTW`, based on the previous experience with it, resulting in the `mg-CVSPTW` heuristic.

The *first modification* is to add a parameter v to weigh the importance of the locomotive capability and we now use the ratio N_b^v/γ_b^{cls} for the computation of the greedy values.

The *second modification* derives an additional greedy value for biasing the solution construction. It is based on the observation that the choice of a trip restricts in very different ways the number of possible trips that feasibly could be added to a locomotive cycle. In the extreme case, locomotives contain only one trip, since it is infeasible to assign more trips within the 24 hour cycle of a locomotive. (We call such trips *isolated*.) Hence, we define for each trip i its *sociability score* s_i, which counts the number of other trips that could be served by a locomotive in a same cycle with trip i; for isolated trips, this score is zero.

A *third modification* is to change the order of steps 1 and 2 in `g-CVSPTW`, together with a more deterministic solution construction. For this modification, now trips are pre-ordered and chosen deterministically in the given order. The ordering is defined in a lexicographic way by (i) considering the number of locomotive classes able to serve the trip (the smaller, the higher the rank) and (ii) the sociability score of each trip (the smaller the score the higher the rank). In the construction process, `mg-CVSPTW` first chooses the starting trip in non-increasing order of the ranks and then fixes the locomotive class. There are two reasons why this modification was deemed to be helpful. Firstly, in this way the most constraining decisions are done first, postponing the ones with more flexibility; secondly, by selecting among a smaller set of candidates, we expected to speed-up the construction process. In fact, we observed that the computation times for the solutions construction decreased by a factor of about five.

Once the first trip and the locomotive class selection is done, the modified greedy heuristic continues iteratively with steps 3 and 4 as `g-CVSPTW`.

3.3 Locomotive type exchange heuristic

A common way of improving solutions returned by a construction method is to improve them by some type of local search. Since the development of very effective local search algorithms can be rather time consuming, we limited the local search to locomotive class exchanges: the idea is to try to replace the locomotive of each tour by a cheaper one. The local search procedure works as follows. First, all locomotive classes are sorted according to non-decreasing costs into a list $c = \langle c_0, c_1, c_{|\mathcal{B}|}\rangle$. Then, for all cycles that use a locomotive of class i we check whether the locomotive of cost c_i can be replaced by a cheaper locomotive, that is, by one of costs $c_0, \ldots, c_{i-1}$ (starting with index 0). If this is possible, the substitution is applied and the next cyclic tour is considered.

```
procedure Iterated Greedy
  s = GenerateInitialSolution
  repeat
    s_p = DestructionPhase(s)
    s' = ConstructionPhase(s_p)
    s'' = LocalSearch(s')          % Optional local search phase
    s = AcceptanceCriterion(s, s'')
  until termination condition met
```

Fig. 1. Algorithmic scheme of iterated greedy with optional local search phase.

4 Iterated greedy algorithms

As said previously, a main motivation for the randomization of the greedy steps in `g-CVSPTW` and in `mg-CVSPTW` is to allow to construct many different solutions. However, this results in independent applications of a heuristic, an approach which for many problems does not lead to excellent performance. Hence, we use *Iterated Greedy* (IG) as another way for iterating over construction heuristics. The central idea of IG is to avoid to repeat a greedy construction from scratch, but rather to keep parts of solutions between successive solution constructions [7, 3]. This is done by first destructing a part of a current complete solutions and then to reconstruct from the resulting partial solution a new complete solution. This new solution is accepted as the new incumbent solution according to some acceptance criterion. For a generic algorithmic outline of IG see Figure 1.

The IG algorithm adds additional parameters. Directly linked to the IG heuristic is the choice how much a complete candidate solution should be destructed; here, we use the parameter *destruction ratio* d. It determines the number of locomotive cycles that are removed from the current solution: if N_l is the number of locomotives in the current solution, then $\lfloor d \cdot N_l \rfloor$ randomly selected locomotives are removed from the current solution. For the re-construction of a complete solution (and the construction of the initial solution), we use the `mg-CVSPTW` heuristic, which proved to be more effective than `g-CVSPTW`.

The acceptance criterion uses the well-known Metropolis rule known from simulated annealing. A candidate solution that is better or of equal quality to the current candidate solution is deterministically accepted; a worse solution is accepted with a probability given by $\exp\{-\Delta/T\}$, where Δ is the cost difference between the new and the current solution and $T = \nu \cdot f(s^*)$, where s^* is the best solution found so far. Note that in the acceptance criterion no annealing is used.

The resulting algorithm we call `IG-CVSPTW`. Our algorithm uses one additional new feature, which was previously not used in IG algorithms. Instead of only reconstructing one complete solution, `IG-CVSPTW` generates from one partial solution $I_r \geq 1$ new solutions. This is done since generating the correct time windows for the partial solutions is relatively computation time intensive due to the constraint propagation for narrowing the time windows.

algorithm	parameter	range	selected value	
			without LS	with LS
`mg-CVSPTW`	*NOISE*	$[0, 1]$	0.67	0.15
	v	$[0, 10]$	1.8	6.3
IG	*NOISE*	$[0, 1]$	0.59	0.57
	v	$[0, 10]$	9.4	2.9
	d	$[0.01, 1]$	0.12	0.074
	ν	$[0, 0.05]$	0.046	0.019
	I_r	$[1, 50]$	9	25

Table 1. Parameter settings obtained from iterative F-race for greedy and IG heuristics. See the text for more details.

Clearly, as for `g-CVSPTW`, it is also desirable for `IG-CVSPTW` to consider the hybridization of the algorithm with a local search, and, hence, we also study this feature. Adding local search to IG is straightforward by improving each reconstructed solution, as indicated by the optional $\mathsf{LocalSearch}(s')$ procedure in Figure 1. As for `mg-CVSPTW`, we use the locomotive class exchange local search.

5 Experimental results for greedy and iterated greedy

In this section, we report on the experimental results and the range of improvements that we obtained with the extensions of the greedy heuristic. Since the clearest differences in performance have been observed on the two largest base instances, we focus on these two in the discussion of the experimental results.

5.1 Experimental Setup

All the codes were written in Java and share the same data structures. The code was compiled and executed using JDK 1.6.0_05. The experiments were run on computing nodes each with two quad-core XEON E5410 CPUs running at 2.33 GHz and 8 GB RAM. Due to the sequential implementation of the algorithms, each execution makes only use of one single core.

Each algorithm was first tuned by an automatic tuning algorithm called iterative F-race [4]; the only exception is `g-CVSPTW`, for which the pre-fixed setting of *NOISE* was used. The three instances `R1-101-3`, `R2-137-6`, and `R3-295-5` were used as training instances for tuning, using the time windows as explained in Section 2.3. Before tuning, the order of the instances is randomized. The computation time for all tuning instances was set to 300 CPU seconds, which corresponds approximately to the time `g-CVSPTW` requires to generate 10 000 solutions for base instance `R-295-5`. Iterative F-race is then run for a maximum of five iterations and in each iteration 100 candidate configurations are generated, that is, a total of 500 configurations for each algorithm are tested. Table 1 gives the range of values considered and the finally chosen ones by iterative F-race.

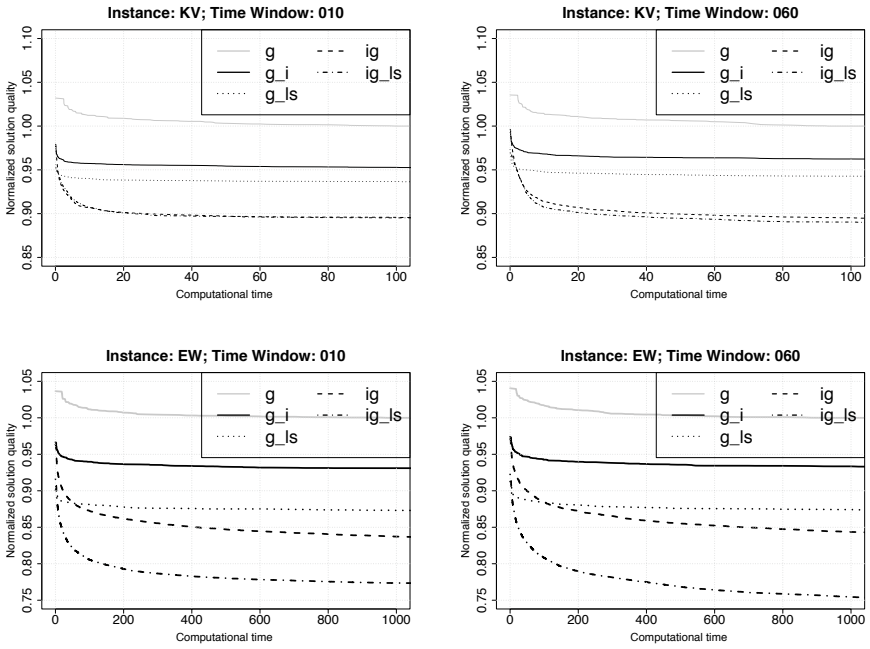

Fig. 2. Development of the solution quality over time for `g-CVSPTW` (g), `mg-CVSPTW` (g_i), `mg-LS-CVSPTW` (g_ls), `IG-CVSPTW` (ig), and `IG-LS-CVSPTW` (ig_ls) for instances `KV-10` (top left), `KV-60` (top right), `EW-10` (bottom left), and `EW-60` (bottom right); the numbers after the instance identifier indicate the time window range chosen. The variability of the averages is low as shown for the solutions after 100 CPU seconds (instances `KV-*`) and 1000 CPU seconds (instances `EW-*`) by the boxplots in Figure 3.

5.2 Experimental results

For an experimental evaluation of the five algorithms, we have run each of them 30 times on the 25 test instances (for each of the five base instances 5 settings of time windows have been considered). Each algorithm was run for the same maximum computation time as required by `g-CVSPTW` to construct 10 000 solutions. In Figure 2, we show for each of the algorithms the development of the average solution quality across 30 independent trials over time. The plots show that initially the solution quality improves very quickly and then the curves flatten off. However, especially the IG variants still show further improvements over time. The differences of the average solution quality reached by the algorithms are rather strong with the best performing one being `IG-LS-CVSPTW`. The boxplots, which are given in Figure 3, also indicate that the variability of each algorithm's final performance is very low. Hence, the significance of the differences could also be confirmed by statistical tests: all pairwise differences between the final solution quality reached by the algorithms are statistically significant according to the Wilcoxon test using Holms correction for the multiple comparisons; the

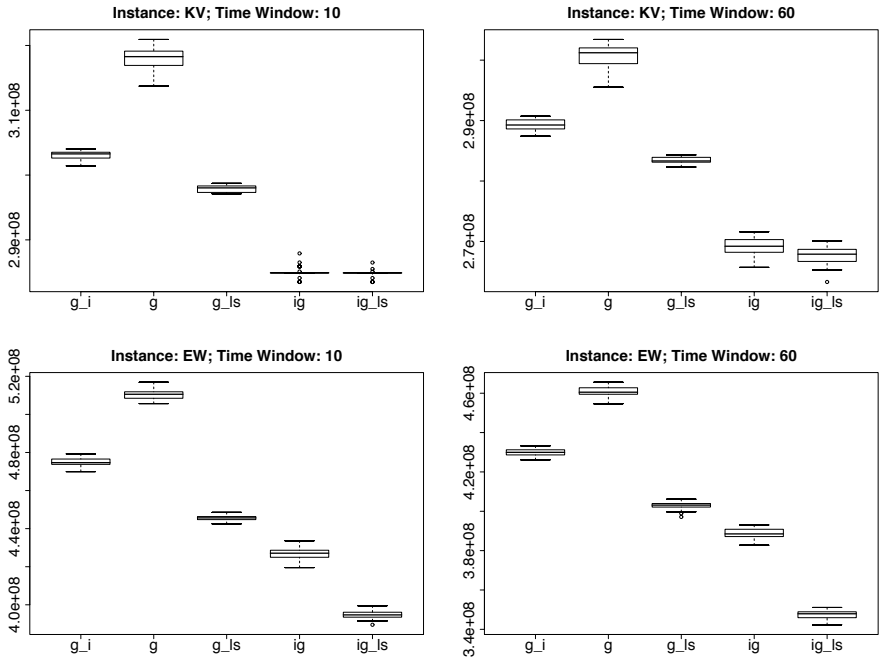

Fig. 3. Boxplots of the final performance after 100 CPU seconds for instances `KV-10` (top left) and `KV-60` (top right), and 1000 CPU seconds for instances `EW-10` (bottom left) and `EW-60` (bottom right). The y-axis shows the solution cost reached. For the explanation of the abbreviations see the caption of Figure 2.

only exception is for instance KV-10, where the differences between `IG-CVSPTW` and `IG-LS-CVSPTW` were not significant. In fact, also on few smaller instances, the differences between `IG-CVSPTW` and `IG-LS-CVSPTW` are not significant.

An important result of this comparison is that the improvement of the computational results over the repeated application of `g-CVSPTW` is very strong. For most instances, the final costs found by `IG-LS-CVSPTW` were about 10% to 20% lower than those reached by `g-CVSPTW`. Typically, the same final solution quality as that of `g-CVSPTW`, was reached by `IG-CVSPTW` or `IG-LS-CVSPTW` in less than a hundredth of the maximum time taken by `g-CVSPTW`.

6 Iterated Ants

The strong improvements by the IG algorithms motivated us to consider extensions that might yield further performance improvements. Given the strong importance of the constructive part, we decided to consider a hybrid between two constructive SLS methods: IG and ant colony optimization (ACO) algorithms; the resulting hybrid algorithm is called *iterated ants* [8]. The central idea of iterated ants is to make each ant in the ACO algorithm follow the steps of the IG

algorithm. The construction phase of the IG algorithm is then biased, as usual in ACO algorithms, by pheromones and heuristic information. The ACO approach we followed was based on a version of $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) that uses the pseudo-random proportional action choice rule known from Ant Colony System (see [9] for more details) and each ant follows the steps of the modified greedy construction heuristic. The pheromone trails have been associated to the choice of the next trip to be added to a locomotive's schedule; that is, a pheromone trail τ_{ij} refers to the desirability of serving trip j after having done trip i; the heuristic information is the inverse of the length of the deadhead trip from i to j. This choice matches the usual ACO approach to the TSP.

Without going into further details on the parameters involved (which were the usual ones arising in the ACO context plus the ones relevant for the IG part), we summarize our main observations. The first is deceptive, in the sense that the final configuration returned from iterated F-race had the importance of the pheromone trails set to zero (Note that the action choice rules in ACO algorithms are positively biased to choose components j for which the term $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$ is high, where α is a parameter that weighs the influence of the pheromone trails. Setting $\alpha = 0$ actually has the effect of not considering the pheromone trails at all in the solution construction.) However, the parameter weighting the influence of the heuristic information was set to its maximum value in the considered range. This is true for the variants with and without local search. Nevertheless, it is still interesting to compare the performance of the "iterated ants" algorithms to the IG ones since both use different rules for constructing complete candidate solutions. Exemplary results for this comparison with plots of the development of the solution quality over computation time are in Figure 4.

The overall best performing variant is `IG-LS-CVSPTW`. Across the 25 test instances, it is statistically better than "iterated ants" with local search on 12 instances and on none statistically worse. The situation is less clear, when considering the variants without local search; here on some instances `IG-CVSPTW` performs better than the "iterated ants" variant (such as the `KV` instance), while the opposite is true on others (such as on the largest instance tested here, `EW`).

7 Discussion

A direct comparison of `IG-LS-CVSPTW` to the performance of the best ILP-based approach in [1] would certainly be interesting. However, a direct comparison is, at least with the current heuristic algorithm code, not straightforward because of some minor differences in the constraints considered. In fact, in the greedy heuristics, a maximum cycle length of 24 hours is imposed for the schedule of each locomotive, which is not done in the ILP formulation: in the ILP formulation a schedule cycle of a locomotive can be an arbitrary multiple of 24 hours. Hence, the heuristic solutions generated within the 24 hour limit for each locomotive is a subset of the ones considered by the ILP approach. Anyway, the solutions generated by the heuristics are feasible for the ILP formulation, that is, despite this difference, the heuristic solutions are valid upper bounds for the

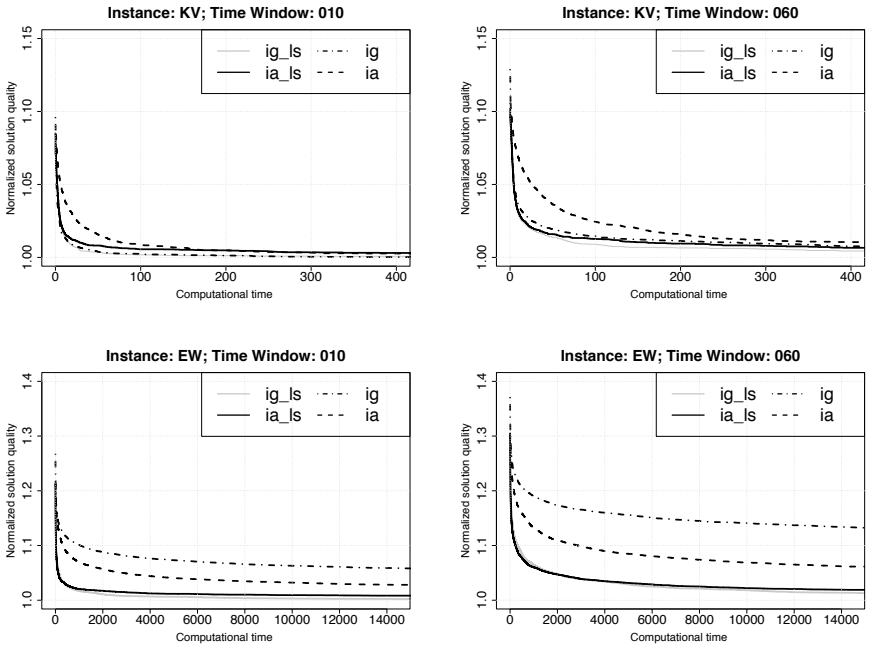

Fig. 4. Development of the solution quality over time for iterated greedy, iterated greedy plus local search, iterated ants and iterated ants plus local search for instances `KV-10` (top left), `KV-60` (top right), `EW-10` (bottom left), and `EW-60` (bottom right).

ILP. The difficulty in comparing our solutions to the ILP ones is that the ILP formulation allows possibly much better solutions to be reached. If we anyway compare the solutions, then we can see that for the same instance–time window combination, the `IG-LS-CVSPTW` algorithm typically finds solutions which use the same number of locomotives or one or two more than in available optimal solutions. Additionally, our heuristics have the advantage that they are applicable to very large instances, which are beyond the reach of the ILP-based approach. In fact, already for the largest instance for which the ILP solver could still deliver solutions in [1] (instance `EW` with time windows ± 30), improved upper bounds could be found by `IG-LS-CVSPTW`.

Our heuristic algorithms can generate good quality in relatively short computation times on the instances tested here, say a few minutes for the largest instance tested here. Given more computation time, `IG-CVSPTW` and `IG-LS-CVSPTW` can use this time effectively and further improve the quality of the solutions over time. On the largest instance tested here, computation times up to four hours have been considered to test the limiting behavior of the heuristics (see, for example, the results given in Figure 4 on instance `EW`). The computational effort is, however, still lower than the one given to the ILP approach, which was run for one CPU hour in parallel on eight cores of a similar machine as ours

(Cplex 10 makes effective use of multiple cores by parallelizing the branch-and-bound tree computations.). In addition, there are still a number of possibilities for improving the speed of the heuristic algorithms, ranging from the adoption of pre-processing techniques to reduce instance size (which is actually done for the ILP model) to more fine-tuned implementations.

Concerning the results for the iterated ants, one may wonder, whether the setting of $\alpha = 0$ is an artifact of the tuning and whether on larger instances the usage of pheromone trails would result into better performance. (Note that the `KV` and `EW` instances are in part much larger than the ones used for tuning.) To test this, we took the second ranked algorithm configuration from the race, which had for both cases–with and without local search–settings of α around one. However, on the two largest instances no significant improvements by using pheromone trails in the solution construction have been identified.

The negative results of the tentative extension of IG to iterated ants allows two different interpretations. On one side, it does not exclude that an iterated ants approach or another population-based extension may further improve performance. For example, different ways of defining the meaning of the pheromone trails may be tested, such as associating locomotive types to trips. However, significant further developments and tests would have to be done, time which could be also used to further improve the simpler algorithm, for example, by using more elaborate construction heuristics or improved local search algorithms. On the other side, these tests are also a confirmation that conceptually rather simple algorithms are a good means to improve the performance of basic heuristics especially in the context of real-world problems. In this sense, the results here give also an example of how a bottom-up development of SLS algorithms, which adds algorithm features in a step-by-step manner, is a viable approach to obtain high performing yet conceptually simple algorithms.

8 Conclusions

In this paper we have developed a high performing stochastic local search algorithm for a freight train scheduling problem arising in the strategical planning of Deutsche Bahn AG. In particular, we have shown that a combination of an iterated greedy heuristic with a simple local search algorithm yields very promising performance. With this algorithm we now can obtain high quality solutions to large problem instances, for which an approach based on a commercial solver ILP solver is not effective anymore.

There are a number of directions in which this research could be extended. A first is certainly the application and comparison of the iterated greedy and iterated ants algorithms on very large instances. For tackling large instances, computation time reductions may be obtained by adopting a pre-processing phase to reduce the effective instance size tackled (this was also indicated by initial tests). Another attractive possibility is to consider hybrids between the exact solution methods and the iterated greedy algorithms. One way to do this is by exploiting the very good performance of the commercial solver for small sized

instances: One may use the iterated greedy algorithm to define partial solutions and use the ILP solver to compute their optimal extensions to complete ones.

Acknowledgments Zhi Yuan acknowledges support from COMP^{2}SYS, a Marie Curie Early Stage Research Training Site funded by the European Community's Sixth Framework Programme and the ANTS project, an *Action de Recherche Concerteé* funded by the Scientific Research Directorate of the French Community of Belgium. Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS of which he is a Research Associate. Henning Homfeld acknowledges support from the German Ministry for Science BMBF, program "Math and Industry". The authors would like to thank also Dr. Gerald Pfau, Andreas Ginkel, and Jörg Wolfner from Deutsche Bahn AG for providing the problem and for fruitful and productive discussions.

References

1. Fügenschuh, A., Homfeld, H., Huck, A., Martin, A., Yuan, Z.: Scheduling locomotives and car transfers in freight transport. Preprint, available online at URL http://www.hausdorff-research-institute.uni-bonn.de/files/preprints/2006transsci.pdf (submitted to Transportation Science)
2. ILOG Ltd.: ILOG Cplex 10 Solver Suite. Technical report, ILOG Cplex Division, 889 Alder Avenue, Suite 200, Incline Village, NV 89451, USA (2006)
3. Ruiz, R., Stützle, T.: A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research **177**(3) (2007) 2033–2049
4. Balaprakash, P., Birattari, M., Stützle, T.: Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In Bartz-Beielstein, T., et al., eds.: 4th International Workshop on Hybrid Metaheuristics, HM 2007. Volume 4771 of Lecture Notes in Computer Science., Springer Verlag, Berlin, Germany (2007) 108–122
5. Fügenschuh, A.: Parametrized Greedy Heuristics in Theory and Practice. In Aguilera, M.B., Blum, C., Roli, A., Sampels, M., eds.: Hybrid Metaheuristics, Second International Workshop, HM 2005, 2005. Number 3636 in Lecture Notes in Computer Science, Springer Verlag (2005) 21 – 31
6. Charon, I., Hudry, O.: The noising method: A new method for combinatorial optimization. Operations Research Letters **14**(3) (1993) 133–137
7. Hoos, H.H., Stützle, T.: Stochastic Local Search—Foundations and Applications. Morgan Kaufmann Publishers/Elsevier, San Francisco, USA (2004)
8. Wiesemann, W., Stützle, T.: Iterated ants: An experimental study for the quadratic assignment problem. In Dorigo, M., et al., eds.: Ant Colony Optimization and Swarm Intelligence, 5th International Workshop, ANTS 2006. Volume 4150 of Lecture Notes in Computer Science., Springer Verlag, Berlin, Germany (2006) 179–190
9. Stützle, T., Hoos, H.: $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System and local search for combinatorial optimization problems. In Voss, S., Martello, S., Osman, I.H., Roucairol, C., eds.: Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization. Kluwer Academic Publishers, Dordrecht, The Netherlands (1999) 137–154