Train Scheduling with Hybrid ASP[⋆]

Dirk Abels[2], Julian Jordi[2], Max Ostrowski[1], Torsten Schaub$^{[0000-0002-7456-041X]}$[1,3⋆⋆], Ambra Toletti[2], and Philipp Wanko$^{[0000-0003-4986-4881]}$[1,3]

[1] Potassco Solutions, Germany
[2] SBB, Switzerland
[3] University of Potsdam, Germany

Abstract. We present an ASP-based solution to real-world train scheduling problems, involving routing, scheduling, and optimization. To this end, we pursue a hybrid approach that extends ASP with difference constraints to account for a fine-grained timing. More precisely, we exemplarily show how the hybrid ASP system *clingo*[DL] can be used to tackle demanding planning-and-scheduling problems. In particular, we investigate how to boost performance by combining distinct ASP solving techniques, such as approximation, heuristic, and optimization strategies.

1 Introduction

Densely-populated railway networks transport millions of people and carry millions of tons of freight daily; and this traffic is expected to increase even further. Hence, for using a railway network to capacity, it is important to schedule trains in a flexible and global way. This is however far from easy since the generation of railway timetables is already known to be intractable for a single track [3]. While this is not so severe for sparse traffic, it becomes a true challenge when dealing with dense networks. This is caused by increasing dependencies among trains due to connections and shared resources.

We take up this challenge and show how to address real-world train scheduling with hybrid Answer Set Programming (ASP [10]). Our hybrid approach allows us to specifically account for the different types of constraints induced by routing, scheduling, and optimization. While we address paths and conflicts with regular ASP, we use difference constraints (over integers) to capture fine timings. Similarly, to boost (multi-objective) optimization, we study approximations of delay functions of varying granularity. This is complemented by various domain-specific heuristics aiming at improving feasibility checking as well as solution quality. We implement our approach with the hybrid ASP system *clingo*[DL] [8], an extension of *clingo* [7] with difference constraints. Our approach provides us with an exemplary study of using a variety of techniques for solving demanding real-world planning-and-scheduling problems with hybrid ASP.

To begin with, we introduce in Section 3 a dedicated formalization of the train scheduling problem. This is indispensable to master the complexness of the problem. Moreover, it guides the development of our hybrid ASP encodings, presented in Section 4. We evaluate our approach along with various enhancements in Section 5 on increasingly difficult problem instances with up to 467 trains.

⋆ This work was partially funded by DFG grants SCHA 550/9 and 11.
⋆⋆ Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 Background

We expect the reader to be familiar with the basic syntax, semantics, and terminology of logic programs under stable models semantics, and focus below on the introduction of non-standard concepts. The base syntax of our logic programs follows the one of *clingo* [5]; its semantics is detailed in [4].

clingo[DL] extends the input language of *clingo* by (theory) atoms representing *difference constraints*. That is, atoms of the form '`&diff`$\{u{-}v\}$`<=` d', where u, v are symbolic terms and d a numeral term, represent difference constraints such as '$u{-}v \leq d$', where u, v serve as integer variables and d stands for an integer.[4] For instance, assume that '`&diff{e(T)-b(T)}<= D`' stands for the condition that the difference between the end and the beginning of a task `T` must be less or equal than some duration D. This may get instantiated to '`&diff{e(7)-b(7)}<= 42`' to require that `e(7)` and `b(7)` take integer values such that '$\mathtt{e}(7){-}\mathtt{b}(7) \leq 42$'. Note that u, v can be arbitrary terms; we exploit this below to use tuples like `(T,V)` as integer variables. Among the alternative semantic couplings between (theory) atoms and constraints offered by *clingo*[DL] (cf. [8, 6]), we follow the *defined, non-strict* approach (i) tolerating theory atoms in rule heads and (ii) enforcing their corresponding constraints only if the atoms are derivable. Hence, if a theory atom is false, its associated constraint is ignored. This approach has the advantage that we only need to consider difference constraints occurring in the encoding and not their negations. Obviously, the overall benefit of using such constraints is that their variables are not subject to grounding.

For boosting performance, we take advantage of *clingo*'s heuristic directives of form '`#heuristic` a`:`B`. [`w`,`m`]`', where a is an atom and B is a body; w is a numeral term and m a heuristic modifier, indicating how the solver's heuristic treatment of a should be changed whenever B holds. We use modifiers `sign` and `false`. Whenever a is chosen by the solver, `sign` enforces that it becomes either true or false depending on whether w is positive or negative, respectively. Similarly, with `false`, a is always assigned false and additionally pushed on priority level w (where 0 is the default; cf. [5]).

3 Real-world train scheduling

The train scheduling problem can be divided into three distinct tasks: routing, conflict resolution and scheduling.

First, each train is routed through a railway network. The directed graph in Figure 1 shows an example of such a network. It consists of all edges regardless of coloring, and nodes numbered from 1 to 12 mark the entry (or exit) of different edges of the track. Furthermore, Figure 1 depicts a valid routing of two trains, t_1 and t_2, through the network. Blue edges are traveled by t_1 and red edges by t_2. In this example, both trains have access to the whole network and may chose among possible start nodes 1,2 and 3, and end nodes 10, 11 and 12, respectively.

Second, edges of the railway network are associated with resources, and whenever the paths of two trains lead through an edge associated with the same resource, there

[4] Strictly speaking, we had to distinguish the integer from its representation.

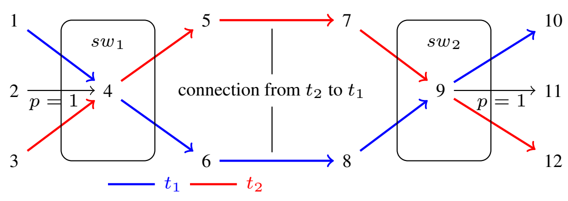

Fig. 1. Routing of two trains through a railway network.

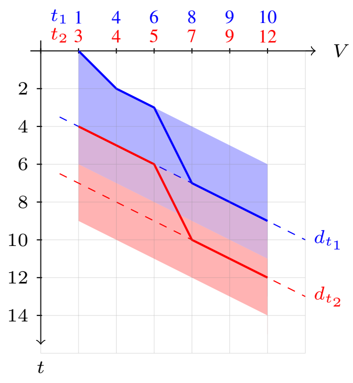

Fig. 2. Scheduling of two trains.

is a possible conflict and a decision has to be made which train accesses the edge first. The train going second has to wait until the first one leaves the edge plus a safety period. Each edge in the directed graph is associated with a resource representing the track, thus prohibiting two trains from entering it simultaneously. Furthermore, resources like railway switches may span several edges. Here, there are two switches, sw_1 and sw_2, represented by rectangles cutting the assigned edges. For instance, given the paths of t_1 and t_2, the two trains have resource conflicts on edges $(1, 4)$ and $(3, 4)$, and similarly, on $(9, 10)$ and $(8, 10)$, since the pairs of edges are assigned to sw_1 and sw_2, respectively.

Finally, for each train and each node visited by the trains, a time point has to be scheduled avoiding conflicts between trains and meeting all timing requirements, such as earliest arrival at nodes or connections between trains. For all trains and each node, an earliest point of arrival is defined, as well as, optionally, a latest point of arrival. Together, these two time points define the time span in which the train might be at a node in its path. Given the paths in Figure 1, Figure 2 shows the time spans and a valid schedule for t_1 and t_2. The horizontal axis indicates the nodes that the trains travel and the vertical axis the time. The light blue and light red areas show the possible arrival times for t_1 and t_2, respectively. The light violet area indicates that both trains may arrive in this time period. For instance, t_1 may arrive at node 4 between time points 2 and 7, and t_2 at node 5 between 6 and 11. The blue and red lines represent a feasible schedule for t_1 and t_2, respectively. In our example, every edge takes one time unit to pass, whenever conflicts are resolved, the second train may enter one time unit after the first has left, and connecting trains have to arrive one time unit before the train that ought to receive cargo or passengers leaves. The schedule in Figure 2 always prioritizes t_1 in resource conflicts and schedules the points of arrival as soon as possible. Resource conflicts at switch sw_1 do not impact t_2's schedule since t_1 leaves these edges several time points before t_2 may arrive. Train t_1 has to wait for t_2 in between nodes 6 and 8 due to their connection, and is allowed to leave at the earliest at time point 7, one time point after t_2 has entered edge $(5, 7)$. The resource conflicts induced by switch sw_2 forces t_2 to wait until time point 10, one time unit after t_1 leaves sw_2.

After obtaining a valid routing and scheduling, the resulting solution is evaluated regarding delay and quality of the trains' paths. For that purpose, edges are assigned penalties. Edges with higher penalties represent, for instance, tracks that can take less workload. In our example, only edges $(2,4)$ and $(9,11)$ are penalized, viz. $p = 1$. Figure 2 shows the time points after which trains t_1 and t_2 are considered delayed via dashed lines d_{t_1} and d_{t_2}, respectively. Every time point below the dashed lines is penalized for the respective train. Since both trains avoid the penalized edges and manage to travel their routes without delay, the solution shown in Figure 2 is optimal.

We formalize the train scheduling problem as a triple (N,T,C). N stands for the railway network (V,E,R,m,a,b), where (V,E) is a directed graph, R is a set of resources, $m : E \rightarrow \mathbb{N}$ assigns the minimum travel time of an edge, $a : R \rightarrow 2^E$ allocates resources in the railway network, and $b : R \rightarrow \mathbb{N}$ gives the time a resource is blocked after it was accessed by a train. Elements (S,L,e,l,w) of T are trains to be scheduled on network N, where (S,L) is an acyclic sub-graph of (V,E), $e : S \rightarrow \mathbb{N}$ and $l : S \rightarrow \mathbb{N} \cup \{\infty\}$ give the earliest and latest time a train may arrive at a node, respectively, and $w : L \rightarrow \mathbb{N}$ is the time a train has to wait on an edge. Note that all functions are total unless specified otherwise and we use seconds as the time unit. Elements (t_1,e_1,t_2,e_2,c) of C are connections, denoting that $t_1 \in T$ on edge $e_1 \in E$ has a connection to $t_2 \in T$ on $e_2 \in E$ requiring t_2 not to leave e_2 before t_1 has arrived by at least c seconds at e_1.

In Figure 1, the train scheduling problem is defined as: $V = \{1,\dots,12\}$, $E = \{(1,4),(2,4),\dots,(9,11),(9,12)\}$, $R = \{sw_1, sw_2\} \cup \{r_e \mid e \in E\}$, $m(e) = 1$ and $a(r_e) = \{e\}$ for $e \in E$, $a(sw_1) = \{(1,4),(2,4),(3,4),(4,5),(4,6)\}$, $a(sw_2) = \{(7,9),(8,9),(9,10),(9,11),(9,12)\}$, $b(r) = 1$ for $r \in R$, $T = \{t_1,t_2\}$ with $t_1 = (S_1,L_1,e_1,l_1,w_1)$, $t_2 = (S_2,L_2,e_2,l_2,w_2)$, where (V,E) equals $(S_1,L_1) = (S_2,L_2)$, e_1,l_1,e_2,l_2 are the upper and lower coordinates of the colored areas in Figure 2, $w_1(e) = w_2(e) = 0$ for $e \in E$, and $C = \{(t_2,(5,7),t_1,(6,8),1),(t_2,(6,8),t_1,(5,7),1)\}$.

A solution (P,A) to a train scheduling problem (N,T,C) is a pair of (i) a function P assigning each train the path it takes through the network, and (ii) an assignment A of arrival times to each train at each node on their path.

A path p is a connected sequence of nodes. We write $v \in p$ and $e \in p$ to denote that node $v \in V$ and edge $e \in E$ are contained in path p, respectively. A path $P(t) = (v_1,\dots,v_n)$ for $t = (S,L,e,l,w) \in T$ with $v_i \in S$ for $1 \leq i \leq n$ has to satisfy:

$$(v_j, v_{j+1}) \in L \text{ for } 1 \leq j \leq n-1 \tag{1}$$

$$in(v_1) = 0 \text{ and } out(v_n) = 0, \tag{2}$$

where in and out give the in- and out-degree of a node in graph (S,L), respectively. Intuitively, Condition (1) forces the path to be connected and feasible for the train in question and Condition (2) ensures that the path is between a possible start and end node.

An assignment A is a partial function $T \times V \rightarrow \mathbb{N}$, where $A(t,v)$ is undefined for $v \notin P(t)$. Given paths P, an assignment has to satisfy:

$$A(t,v_i) \geq e(v_i) \tag{3}$$

$$A(t,v_i) \leq l(v_i) \tag{4}$$

$$A(t,v_j) + m((v_j,v_{j+1})) + w((v_j,v_{j+1})) \leq A(t,v_{j+1}) \tag{5}$$

for all $t = (S,L,e,l,w) \in T, P(t) = (v_1,\dots,v_n), 1 \leq i \leq n, 1 \leq j \leq n-1$,

$$\text{either } A(t_1,v') + b(r) \leq A(t_2,u) \text{ or } A(t_2,u') + b(r) \leq A(t_1,v) \tag{6}$$

for $r \in R, \{t_1, t_2\} \subseteq T, t_1 \neq t_2, (v, v') \in P(t_1), (u, u') \in P(t_2)$ with $\{(v, v'), (u, u')\} \subseteq a(r)$, and

$$A(t_1, v) + c \leq A(t_2, u') \tag{7}$$

for $(t_1, (v, v'), t_2, (u, u'), c) \in C$ with $(v, v') \in P(t_1)$ and $(u, u') \in P(t_2)$.

Intuitively, conditions (3), (4) and (5) ensure that a train arrives neither too early nor too late and that waiting and traveling times are accounted for. Furthermore, Condition (6) resolves conflicts between two trains that travel edges sharing a resource, so that either the second train can only enter after the first train has left for a specified time or vice versa. Finally, Condition (7) handles connections between two trains: a train with a connection can only leave if the other train has arrived for a specified time. Note that connections only apply if both trains travel the specified edges.

For our solution in Figure 2, $P(t_1) = (1, 4, 6, 8, 9, 10)$, $P(t_2) = (3, 4, 5, 7, 9, 12)$, and $A(t_1, 1) = 0, \dots, A(t_1, 8) = 7, \dots, A(t_1, 10) = 9, A(t_2, 3) = 4, \dots, A(t_2, 5) = 6, A(t_2, 7) = 10, \dots, A(t_2, 12) = 12$.

To determine the quality of a solution, the aggregated delay of all trains as well as the quality of the paths through the network is taken into account. For that purpose, we consider two functions: the delay function d and route penalty function rp. Given train $t = (S, L, e, l, w) \in T$ and a node $s \in S$, $d(t, s) \in \mathbb{N}$ returns the time point after which the train t is considered late at node s. Note that $e(s) \leq d(t, s) \leq l(s)$. Given an edge $e \in E$, $rp(e) \in \mathbb{N}$ is the penalty a solution receives for each train traveling edge e. The quality of a solution (P, A) is determined via the following equation:

$$\textstyle\sum_{((t,v),a) \in A} \max\{(a - d(t, v)), 0\}/60 + \sum_{e \in \{u \mid p \in P, u \in p, e \in E\}} rp(e) \tag{8}$$

In our example, we get $rp((2, 4)) = 1, rp((9, 11)) = 1$ and $rp(v) = 0$ for $v \in V \setminus \{(2, 4), (10, 12)\}$, and, for instance, $d(t_1, 1) = 4, d(t_1, 10) = 9$ and $d(t_2, 3) = 7, d(t_2, 12) = 12$. As mentioned, our solution obtains the optimal quality value of 0.

4 An ASP-based solution to real-world train scheduling

In this section, we present our hybrid solution to the train scheduling problem. We start by describing the factual representation of problem instances, continue with the problem encoding and finish by introducing several domain-specific heuristics aimed at improving solving performance and solution quality.

Fact format. A train scheduling problem (N, T, C) with $N = (V, E, R, m, a, b)$ is represented by

$$\texttt{train}(t) \quad \texttt{edge}(t, v, v') \quad \texttt{m}((v, v'), m((v, v'))) \quad \texttt{w}(t, (v, v'), w((v, v')))$$

for every $t = (S, L, e, l, w) \in T$ and $(v, v') \in L$. For every $s \in S$, we add

$$\texttt{e}(t, s, e(s)) \quad \texttt{l}(t, s, l(s)), \quad \text{and} \quad \texttt{start}(t, s) \quad \text{or} \quad \texttt{end}(t, s)$$

if $in(s) = 0$ or $out(s) = 0$ in (S, L), respectively. We assign unique terms to each train for identifiability. For example, facts `train(t1)`, `edge(t1,1,4)`, `e(t1,1,0)`, `l(t1,1,6)`, `start(t1,1)`, `m((1,4),1)` and `w(t1,(1,4),0)` express that train `t1` may travel between nodes `1` and `4` taking at least 1 second, waiting on this edge for 0 seconds, and arrives between time points 0 and 6 at node `1`, which is a possible start node. Furthermore, we add

$$\texttt{resource}(r, e) \quad \texttt{b}(r, b(r))$$

for $r \in R$ and $e \in a(r)$. Akin to trains, resources are assigned unique terms to distinguish them. For example, facts `resource(sw1,(1,4))`, `resource(sw1,(4,5))` and `b(sw1,1)` assign resource `sw1` to edges $(1, 4)$ and $(4, 5)$ and the resource is blocked for 1 second after a train has left it. Finally, we add

$$\texttt{connection}(t, (v, v'), t', (u, u'), c)$$

for all $(t, (v, v'), t', (u, u'), c) \in C$. The fact that `t1` may not leave $(6, 8)$ before `t2` spent at least 1 second in $(5, 7)$ is encoded by `connection(t2,(5,7),t1,(6,8),1)`.

Given delay and route penalty functions d and rp, we add

$$\texttt{potlate}(t, s, u, p) \quad \texttt{penalty}(m, rp(m))$$

for $t = (S, L, e, l, w) \in T, s \in S, m \in L$ with $\{u, p\} \subseteq \mathbb{N}, d(t, s) < u \leq l(t, s)$ to evaluate a solution. While we encode route penalty optimization exactly, delay optimization is approximated via a combination of difference constraints and standard ASP optimization schemes. Intuitively, a fact `potlate(t1,1,5,1)` denotes that a solution receives a penalty of 1 if train `t1` travels over node `1` and arrives there at time point 5 or later. In Section 4, we introduce several schemes to create such facts and approximate the objective function to different degrees. For brevity, we cannot present the full instance representing the example in figures 1 and 2 but make it available here[5].

Encoding. In the following, we describe the general problem encoding. We separate it into three parts handling path constraints, conflict resolution and scheduling.

Listing 1.1. Encoding of path constraints.

```
1  1 { visit(T,V)        : start(T,V)    } 1 :- train(T).
2  1 { route(T,(V,V')) : edge(T,V,V') } 1 :- visit(T,V), not end(T,V).
3  visit(T,V')     :- route(T,(V,V')).
```

The first part of the encoding in Listing 1.1 covers routing. First, exactly one valid start node is chosen for each train to be visited (Line 1). From a node that is visited by a train and is not an end node, an edge in the relevant sub-graph is chosen as the next route (Line 2). The new route in turn leads to a node being visited by the train (Line 3). This way, each train is recursively assigned a valid path. Since those paths begin at a start node, finish at an end node and are connected via edges valid for the respective trains, conditions (1) and (2) are ensured.

Listing 1.2. Encoding of conflict resolution.

```
4  shared(T,(V,V'),T',(U,U')) :- edge(T,V,V'), edge(T',U,U'), T!=T',
5                                 resource(R,(V,V')), resource(R,(U,U')), b(R,B),
6                                 e(T,V,E), l(T,V',L), e(T',U,E'),
7                                 E <= E', E' < L+B.
8  shared(T',E',T,E)     :- shared(T,E,T',E').
```

[5] github.com/potassco/train-scheduling-with-clingo-dl/blob/master/example.lp

```
 9  conflict(T,E,T',E')  :- shared(T,E,T',E'), T < T',
10                          route(T,E), route(T',E').
11  { seq(T,E,T',E') } :-  conflict(T,E,T',E').
12    seq(T',E',T,E)   :-  conflict(T,E,T',E'), not seq(T,E,T',E').
```

The next part of the encoding shown in Listing 1.2 detects and resolves resource conflicts. A resource conflict is possible, if two trains have an edge in their sub-graphs that is assigned the same resource (lines 4 and 5), and they travel through the edges around the same time (lines 6 and 7), more precisely, whenever the time intervals in which the trains may enter and leave the edges in question, extended by the time the resource is blocked, overlap. Now, if both trains are routed through those edges a conflict occurs (lines 9 and 10). We resolve the conflict by making a choice which train passes through their edge first (lines 11 and 12).

Listing 1.3. Encoding of scheduling.

```
13  &diff{ 0-(T,V) } <= -E    :- e(T,V,E), visit(T,V).
14  &diff{ (T,V)-0 } <=  L    :- l(T,V,L), visit(T,V).
15  &diff{ (T,V)-(T,V') }  <= -D :- route(T,(V,V')), E = (V,V'),
16                                  D=#sum{ M,m : m(E,M); W,w : w(T,E,W) }.
17  &diff{ (T,V')-(T',U) } <= -M :- seq(T,(V,V'),T',(U,U')),
18                                  M = #max{ B : resource(R,(V,V')), b(R,B) }.
19  &diff{ (T,V)-(T',U') } <= -W :- connection(T,(V,V'),T',(U,U'),W),
20                                  route(T,(V,V')), route(T',(U,U')).
```

Finally, Listing 1.3 displays the encoding of scheduling via difference constraints. We represent arrival times of train `t` at node `v` with an integer variable `(t,v)`. In the following, we use ground terms to describe how the rules function while in the encoding variables are used. Lines 13 and 14 encode that every train arrives at a node in their path neither too early nor too late, respectively. Given the earliest arrival e and latest arrival l of a train `t` at node `v` in their path, difference constraint atoms `&diff{0-(t,v)}<=` $-e$ and `&diff{(t,v)-0}<=` l are derived. This ensures that $e \leq$ `(t,v)` $\leq l$ holds, therefore fulfilling conditions (3) and (4). The rule in lines 15 and 16 first calculates the sum d of minimal travel and waiting time for train `t` at edge `(v,v')` in their path, which is the minimal difference between arrival times at nodes `v` and `v'` for train `t`. Then, difference constraint atom `&diff{(t,v)-(t,v')}<=` $-d$ is derived, which in turn ensures `(t,v)` $+d \leq$ `(t,v')` (Condition (5)). The rule in lines 17 and 18 utilizes conflict detection and resolution from Listing 1.2. Given the maximum blocked time b of resources shared on `(v,v')` and `(u,u')`, and the decision that `t` takes precedence over `t'`, we derive difference constraint atom `&diff{(t,v')-(t',u)}<=` $-b$ expressing linear constraint `(t,v')` $+ b \leq (t', u)$ for two conflicting trains `t` and `t'` on edges `(v,v')` and `(u,u')`. Hence, `t'` may only enter edge `(u,u')` b seconds after `t` has left `(v,v')` (Condition (6)). Note that if several resources induce a conflict for two trains on the same edges, only one difference constraint with the maximum blocked time suffices since $x + k \leq y$ implies $x + k' \leq y$ for $k \geq k'$. Finally, Line 19 handles connections in a similar fashion. If train `t` on `(v,v')` has a connection to `t'` on `(u,u')` with connection time w, a difference constraint atom `&diff{(t,v)-(t',u')}<=` $-w$ is derived, ensuring linear constraint `(t,v)` $+ w \leq$ `(t',u')` to hold (Condition (7)). This condition is required if both trains are routed through the edges (Line 20).

Optimization. As mentioned above, we use instances of `potlate`/4 to indicate when a train is considered late at a node and how to penalize its delay. For this purpose, we choose sets $D_{t,v} \subseteq \mathbb{N}$ whose elements act as thresholds for arrival time of train

t at node v. Given delay function d, $d(t,v) \leq u \leq l(v)$ for every $u \in D_{t,v}$, train $t = (S, L, e, l, w) \in T$ and $v \in S$. We create facts potlate(t,v,u,$u - u'$) for $u, u' \in D_{t,v}$ with $u' < u$ such that there is no $u'' \in D_{t,v}$ with $u' < u'' < u$. We add potlate(t,v,u,$u - d(t,v)$) for $u = \min(D_{t,v})$. Intuitively, we choose the penalty of a potential delay as the difference to the previous potential delay, or, if there is no smaller threshold, the difference to the time point after which the train is considered delayed. This way, the sum of penalties amounts to a lower bound on the train's actual delay in seconds. For example, for $D_{t,v} = \{6, 10, 14\}$ and $d(t,v) = 5$, we create facts potlate(t,v,6,1),potlate(t,v,10,4) and potlate(t,v,14,4). Now, if t arrives at v at 12, it is above thresholds 6 and 10 and should receive a penalty of 5. This penalty is a lower bound on the actual delay of 7, and we know that the value has to be between 5 and 9 since the next threshold adds a penalty of 4. This method approximates the exact objective function in (8) in two ways. First, we do not divide by 60 and penalize in minutes since this would lead to rounding problems. Second, our penalty only gives a lower bound to the actual delay if thresholds are more than one second apart. While our method allows us to be arbitrarily precise in theory, in practice, creating a threshold for each possible second of delay leads to a explosion in size. We employ two schemes for generating sets $D_{t,v}$ given $t = (S, L, e, l, w) \in T$, $v \in S$ and delay function d.

Binary. This approximation detects if a train is a second late and penalizes it by one, therefore, only the occurrence of a delay is detected while its amount disregarded. We set $D_{t,v} = Bin_{t,v} = \{d(t,v) + 1\}$.

Linear. This scheme for $D_{t,v}$ evenly distributing thresholds m seconds apart across the time span in which a delay might occur. Here, if train t arrives at or after $n*m+d(t,v)$ at v, we know that the real delay is between $n*m$ and $(n+1)*m$ for $n \in \mathbb{N} \setminus \{0\}$. We also add $Bin_{t,v}$ to detect solutions without delay. We set $D_{t,v} = Bin_{t,v} \cup Lin^m_{t,v}$ with $Lin^m_{t,v} = \{y \in \mathbb{N} \mid y = x*m + d(t,v), x \in \mathbb{N} \setminus \{0\}, y \leq l(v)\}$.

Listing 1.4. Delay and routing penalty minimization.

```
1  { late(T,V,D,W) : visit(T,V) } :- potlate(T,V,D,W).
2  &diff{ 0-(T,V) } <= -D  :- late(T,V,D,W).
3  &diff{ (T,V)-0 } <=  N  :- not late(T,V,D,W), potlate(T,V,D,W),
4                             N=D-1, visit(T,V).
5  #minimize{ W,T,V,D : late(T,V,D,W) }.
6  #minimize{ P,T,E : penalty(E,P), route(T,E) }.
```

Given thresholds $D_{t,v}$ for all trains and nodes and the corresponding instances of predicate potlate/4, Listing 1.4 shows the implementation of the delay minimization. The basic idea is to use regular atoms to choose whether a train is delayed on its path for every potential delay (Line 1), deriving difference constraint atoms expressing this information (lines 2–4), and ultimately using the regular atoms in a standard minimize statement (Line 5). In detail, for every potlate(t,v,u,w), a late(t,v,u,w) can be chosen if t visits v. If late(t,v,u,w) is chosen to be true, a difference constraint atom &diff{0-(t,v)}<= $-u$ is derived expressing $(t,v) \geq u$ and, therefore, that t is delayed at v at threshold u. Otherwise, &diff{(t,v)-0}<= $u-1$ becomes true so that $(t,v) < u$ holds. The difference constraints ensure that if the truth value of a late atom is decided, the schedule has to reflect this information. The minimize statement then sums up and minimizes the penalties of the late atoms that are true.

Finally, Line 6 in Listing 1.4 shows the straight forward encoding of the routing penalty minimization. The minimize statement merely collects the paths of the trains, sums up their penalties, and minimizes this sum.

Domain-specific heuristics. We devise several domain-specific heuristics to, first, improve solving performance, and second, improve quality of solutions regarding delay and routing.

Sequence heuristic. The heuristic in Listing 1.5 attempts to order conflicting trains by their possible arrival times at the edges where the conflict is located. In essence, we analyze how the time intervals of the trains are situated and prefer their sequence accordingly. Line 1 derives those intervals by collecting the earliest and latest time a train might be at an edge. Given two trains `t` and `t'` with intervals $[e, l]$ and $[e', l']$ at the conflicting edges, respectively, we calculate $s = e' - e - (l - l')$ to determine whether `t` should be scheduled before `t'`. If s is positive, the preferred sign of the sequence atom is also positive, thus preferring `t` to go before `t'`, if it is negative, the opposite is expressed. In detail, $e' - e$ is positive if `t'` may arrive later than `t` thus making it more likely that `t` can go first without delaying `t'`. Similarly, if $l - l'$ is negative, `t'` may leave later, suggesting `t` to go first. If the results of both expressions have the same sign, one interval is contained in the other and if the difference is positive, the center of the interval of `t` is located earlier than the center of the interval of `t'`. For example, in Figure 2, we see that t_1 and t_2 share a resource in $(1, 4)$ and $(3, 4)$ and the time intervals in which they potentially arrive at those edges are $[0, 7]$ and $[4, 10]$, respectively. Due to $4 - 0 - (7 - 10) = 7$, we prefer t_1 to be scheduled before t_2, which in the example clearly is the correct decision, since t_1 precedes t_2 without delaying t_2.

Listing 1.5. Heuristic that orders conflicting trains by their possible arrival times.

```
1  range(T,(V,V'),E,L) :- edge(T,V,V'),e(T,V,E), l(T,V',L).
2  #heuristic seq(T,E,T',E') : shared(T,E,T',E'),
3                              range(T,E,L,U),
4                              range(T',E',L',U'). [L'-L - (U-U'),sign]
```

Delay heuristic. Listing 1.6 gives a heuristic aimed at avoiding delay at earlier nodes in the paths. For that purpose, we first assign each node in the sub-graph of a train a natural number signifying their relative position (lines 1–4). Start nodes receive position 0, and from there, the number increases the farther a node is apart from the start nodes, indicating that they are visited later in the possible paths of the train. The maximum position of the end nodes is also the longest possible path minus one (Line 5). For a potential delay, we then select the position p and the maximum position m and modify the delay atom with value $m - p$ and modifier `false`. This accomplishes two things. First, the earlier the node, the higher the value, thus delays for earlier nodes are decided first. Second, the preferred sign of all delays is false. Intuitively, we assume that early delays are to be avoided since they likely lead to delays at subsequent nodes. Considering again our example in Figure 2, node 1 for t_1 receives position 0 and node 5 position 3, respectively, while the maximum position is 5. Therefore, we receive values 5 and 2 for nodes 1 and 5, respectively, avoiding the delay at node 1 first, while also preferring t_1 to be on time at both nodes.

Listing 1.6. Heuristic discouraging delays early on.

```
1  node(T,(V;V'))    :- edge(T,V,V').
```

```
node_pos(T,V,0)   :- start(T,V).
node_pos(T,V',M+1) :- node(T,V'), not start(T,V'),
                      M = #max{ P : node_pos(T,V,P), edge(T,V,V')}.
last_node(T,M)    :- train(T), M = #max{ P : node_pos(T,V,P), end(T,V) }.
#heuristic late(T,V,U,W) : potlate(T,V,U,W),
                           node_pos(T,V,P),
                           last_node(T,Max). [Max-P,false]
```

Routing heuristic. Akin to the straight-forward routing penalty minimization, the heuristic in Listing 1.7 merely tries to avoid routes where there is a penalty. The higher the penalty, the more those routes are to be avoided. In our example (Figure 1), this amounts to t_1 and t_2 equally shunning $(2, 4)$ and $(9, 11)$.

Listing 1.7. Heuristic for avoiding paths with penalties.

```
#heuristic route(T,E) : train(T), penalty(E,P). [P,false]
```

Note that all three domain-specific heuristics are static, i.e., they are active immediately at the start of solving.

5 Experiments

We evaluate our train scheduling solution using the hybrid solver *clingo*[DL] v1.0, which is build upon the API of *clingo* 5.3.[6] We use nine real-world instances published by Swiss Federal Railway (SBB) to test different configurations of *clingo*[DL] with optimization strategies and domain-specific heuristics (60 in total). For brevity, we omit slight grounding and propagation optimizations in the encoding presented in Section 4; the full encoding and instance set is at github.com/potassco/train-scheduling-with-clingo-dl. We validate solution feasibility and quality via an external program also provided by SBB.[7] All benchmarks ran on Linux with a Xeon E3-1260L quad-core 2.9 GHz processors and 32 GB RAM; each run limited to 3 hours and 32GB RAM. In detail, we examine the following techniques:

Optimization schemes. (BB) *Model-guided* optimization iteratively producing models of descending cost until the optimum is found by establishing unsatisfiability of finding a model with lower cost. (USC) *Core-guided* optimization relying on successively identifying and relaxing unsatisfiable cores until a model is obtained. (nT) Natural number n determines the number of threads with which the solver is run. Threads use the same search space but might learn different clauses that are exchanged. If either BB or USC is additionally specified, both threads use the respective optimization scheme.

All other parameters are using the default of *clingo*[DL]. In particular, the default for 2T configures thread 1 with BB and thread 2 with USC in the hope that the shared information improves overall performance and solution quality.

Objective function approximation. We only vary delay optimization and use the same minimize statement for route penalty (see Section 4). (BIN) Delay approximation only penalizing instances and not amount of delay. We set $D_{t,v} = Bin_{t,v}$ for each train t at node v. (LIN) Delay approximation creating thresholds evenly within time span

[6] We use the releases for both *clingo*[DL] and *clingo* that are available at github.com/potassco/clingoDL and github.com/potassco/clingo.

[7] www.crowdai.org/challenges/train-schedule-optimisation-challenge.

of possible delay. We set $D_{t,v} = Bin_{t,v} \cup Lin^{180}_{t,v}$ for each train t at node v. For LIN, we chose the distance of thresholds, viz. 180, such that there are 5 thresholds with a maximum threshold of 15 minutes. We also examined an exponential distribution of thresholds where the distance doubles every time so that the precision is higher for lower delays, and significant delays receive a greater penalty. We omit the results since the approach does not improve quality and displays worse performance compared to LIN.

Domain-specific heuristics. For details, see Section 4. (NONE) Domain-specific heuristics are disabled. (SEQ) Sequence heuristic in Listing 1.5. (DELAY) Delay heuristic in Listing 1.6. (ROUTES) Routing heuristic in Listing 1.7. (ALL) All heuristics SEQ, DELAY and ROUTES are enabled.

HEU / OPT	NONE T	NONE QU	SEQ T	SEQ QU	DELAY T	DELAY QU	ROUTES T	ROUTES QU	ALL T	ALL QU
BIN-1T	4767	175	4136	165	2578	165	3215	175	**684**	165
BIN-2T	933	181	*575	184	937	173	909	175	**574**	**165**
BIN-2T-BB	5050	166	4723	175	2481	165	1916	177	**600**	165
BIN-2T-USC	*877	165	581	184	*881	165	*881	175	**574**	173
LIN-1T	–	–	23343	33	6380	33	–	–	**705**	33
LIN-2T	1118	33	694	33	1264	33	926	33	**611**	33
LIN-2T-BB	–	–	16495	33	4561	33	11667	33	**605**	33
LIN-2T-USC	4047	33	**2351**	33	–	–	–	–	–	–

Table 1. Aggregated wall time and quality.

				ALL-BIN-2T			ALL-LIN-2T-BB		
INS	#T	#N	#E	T	AQU	QU	T	AQU	QU
1	4	159	159	2	0	0	2	0	0
2	58	1839	1816	5	0	0	5	0	0
3	143	2117	2090	8	0	0	9	0	0
4	148	2371	2352	12	0	0	13	0	0
5	149	2376	2356	19	8	165	42	21	33
6	365	3128	3109	149	0	0	144	0	0
7	467	3128	3109	252	0	0	251	0	0
8	133	3228	3314	127	0	0	139	0	0
9	287	34488	34827	–	–	–	–	–	–

Table 2. Instance details and best results.

In our experiments, we used *clingo*[DL] to report one optimal solution for each configuration. Table 1 shows the sum of wall time in seconds in columns T, and the value of the exact objective function as reported by the external validation tool in columns QU, for all combinations of optimization strategy (rows of the table) and domain-specific heuristics (columns of the table) that were able to report one valid optimal solution for instances 1 through 8. Note that all values are rounded to integers. We omit results for instance 9 since grounding was not possible within 32GB of memory. Combinations where some instances timed out are marked with –. This way, we are able to exclude inferior results while being able to accurately compare performance and solution quality of successful configurations. The best results in a row and in a column for wall time and quality are marked bold and with *, respectively, unless at least two configurations achieved the same result.

Regarding wall time, ALL clearly performs best and improves performance up to one order of magnitude compared to NONE. While each domain-specific heuristic has a positive impact, either reducing wall time or allowing all instances to be solved optimally, we observe that SEQ has the most benefit on its own, but the joint effect of the three heuristics is vital in achieving the best possible performance. Furthermore, optimization approximation BIN performs best, displaying no timeouts and best aggregate wall time by a slight margin. Since weights in the optimization statement for BIN are all one, USC is very effective for it. For LIN, on the other hand, running both BB and USC simultaneously proved to be successful, most likely due to a mixture of different weight values and the benefit of the shared clauses between threads. As expected, the simple approximation of the objective function BIN is easier to solve but provides solutions of less quality. Note that while all the solutions returned by *clingo*[DL] were optimal, the value of the exact objective function varies. If there are several optimal solutions, a different one might be reported depending on heuristics or thread-based interference, i.e., we cannot guarantee that the same optimal model is found for different configurations. Approximations LIN has overall more timeouts and worse wall time, but solution quality is higher.

Table 2 shows for all instances the number of trains(#T), nodes(#N) and edges(#E) along with wall time(T), approximated quality(AQU) and exact quality(QU) for the configuration with the best performance, viz. ALL-BIN-2T, and best quality-performance ratio, viz. ALL-LIN-2T-BB. We found optimal solutions to the approximated objective functions for instances with up to 467 trains, 3228 nodes and 3314 edges within 5 minutes. Except for instances 4 and 5, which were crafted specifically to contain obstructions inducing delay, we could provide solutions without any penalties regarding the exact objective function. For Instance 4, the delay is negligible and for Instances 5, we achieve a value close to the best possible solution according to SBB. We see that BIN is a good choice for instances that are expected to be solvable without delay, but for more difficult instances, like Instance 5, the approximation is too inaccurate. On the other hand, LIN achieves more accuracy with similar performance mostly thanks to the domain-specific heuristics that steer the solving process to promising regions of the search space.

Overall, we observe that all domain-specific heuristics, SEQ in particular, linear approximation of delay optimization, and several threads with multiple optimization strategies, allow us to successfully solve the train scheduling problem for a variety of real-world instances in acceptable time.

6 Discussion

At its core, train scheduling is similar to classical scheduling problems that were already tackled by ASP. Foremost, job shop scheduling [16] is also addressed by *clingo*[DL] and compared to other hybrid approaches in [8]; solutions based on SMT, CP and MILP are given in [9, 2], [1], and [11], respectively. In fact, job shop scheduling can be seen as a special case of our setting, in which train paths are known beforehand. Solutions to this restricted variant via MILP and CP are presented in [12, 15]. The difference to our setting is twofold: first, resource conflicts are not known beforehand since we take routing and scheduling simultaneously into account. Second, our approach encompasses a global view of arbitrary precision, i.e., we model all routing and scheduling decisions across hundreds of trains and possible lines down to inner-station conflict resolution. Furthermore, using hybrid ASP with difference constraints gives us inherent advantages over pure ASP and MILP. First, we show in [6] that ASP is not able to solve most shop scheduling instances since grounding all the integer variables leads to an explosion in problem size. We avoid this bottleneck by encapsulating scheduling in difference constraints and, hence, avoid grounding integer variables. Second, while difference constraints are less expressive than linear constraints in MILP, they are sufficient for expressing the timing constraint needed for train scheduling and are solvable in polynomial time. Finally, routing and conflict resolution require Boolean variables and disjunctions for which ASP has effective means.

Since we produce timetables from scratch, our train scheduling problem can be characterized as tactical scheduling [17]. In the future, we aim at addressing re-scheduling [13, 14], where existing timetables have to be adapted to sudden deviations. While our hybrid ASP encoding can be easily modified to accommodate such advanced reasoning tasks, we currently could not address them in real-time. The main challenge lies in reducing the size of the problem. We have shown that, if grounding is possible, we can effectively solve real-world train scheduling with *clingo*[DL]. The problem size can be reduced by first, compressing the graph and removing nodes that are redundant in terms of timing constraints that they pose to the schedule, and second, identifying groups of conflicts of trains that only require a single decision to be resolved in a preprocessing step.

References

1. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-based scheduling: applying constraint programming to scheduling problems, vol. 39. Springer Science & Business Media (2012)
2. Bofill, M., Palahí, M., Suy, J., Villaret, M.: Solving constraint satisfaction problems with sat modulo theories. Constraints **17**(3), 273–303 (2012)
3. Caprara, A., Fischetti, M., Toth, P.: Modeling and solving the train timetabling problem. Operations Research **50**, 851–861 (2002)
4. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5), 449–463 (2015).
5. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S.: Potassco User Guide. second edition edn. (2015), http://potassco.org
6. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Technical Communications of the International Conference on Logic Programming (ICLP'16). vol. 52, pp. 2:1–2:15. OASIcs (2016)
7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019)
8. Janhunen, T., Kaminski, R., Ostrowski, M., Schaub, T., Schellhorn, S., Wanko, P.: Clingo goes linear constraints over reals and integers. Theory and Practice of Logic Programming **17**(5-6), 872–888 (2017)
9. Janhunen, T., Liu, G., Niemelä, I.: Tight integration of non-ground answer set programming and satisfiability modulo theories. In: Proceedings of the Workshop on Grounding and Transformation for Theories with Variables (GTTV'11). pp. 1–13 (2011)
10. Lifschitz, V.: Answer set planning. In: Proceedings of the International Conference on Logic Programming (ICLP'99). pp. 23–37. MIT Press (1999)
11. Liu, G., Janhunen, T., Niemelä, I.: Answer set programming via mixed integer programming. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR'12). pp. 32–42. AAAI Press (2012)
12. Oliveira, E., Smith, B.: A job-shop scheduling model for the single-track railway scheduling problem. University of Leeds, LU SCS RR (21) (2000)
13. Pellegrini, P., Douchet, G., Marlière, G., Rodriguez, J.: Real-time train routing and scheduling through mixed integer linear programming: Heuristic approach. In: Proceedings of the international conference on industrial engineering and system management. pp. 1–5 (2013)
14. Pellegrini, P., Marlière, G., Pesenti, R., Rodriguez, J.: Recife-milp: An effective MILP-based heuristic for the real-time railway traffic management problem. IEEE Transactions on Intelligent Transportation Systems **16**(5), 2609–2619 (2015)
15. Rodriguez, J.: A constraint programming model for real-time train scheduling at junctions. Transportation Research: Methodological **41**(2), 231–245 (2007).
16. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64**(2), 278–285 (1993)
17. Törnquist, J.: Computer-based decision support for railway traffic scheduling and dispatching: A review of models and algorithms. In: Proceedings of the Workshop on Algorithmic Methods and Models for Optimization of Railways. OASIcs. vol. 2. (2006)