



Multiple Train Repositioning Operations in a Railyard Network

Mina Aliakbari¹ · Joseph Geunes¹

Received: 9 December 2021 / Accepted: 25 September 2022 / Published online: 14 October 2022
© The Author(s), under exclusive licence to Springer Nature Switzerland AG 2022

Abstract

This paper considers the simultaneous movement of multiple trains within a railyard network, where each of a number of trains has an origin location and destination location on the network. We wish to minimize the total time required to move all trains from their origin to destination locations, while ensuring that at most one train occupies each track segment at any given time. We propose an integer programming model that is able to solve small problem instances exactly, as well as a heuristic solution method for solving problems of realistic size in acceptable computing time. Our constructive heuristic approach uses a ranked priority list of trains that require repositioning, and sequentially determines a route on the network for each train in priority order. We then relax the strict priority ordering rule by applying a Greedy Randomized Adaptive Search Procedure (GRASP) based on the underlying constructive heuristic. As we demonstrate via a set of computational tests, this heuristic approach is able to find good quality feasible solutions in fast computing time, drastically reducing the labor hours typically dedicated to routinely solving this problem in practice.

Keywords Railway logistics · Mixed integer programming · Shortest route

1 Introduction

A freight rail network generally consists of a set of geographically dispersed railyards that are interconnected via so-called main track lines, or main lines, which often traverse long distances (e.g., several hundred miles) between connected railyards. Each railyard (also called a hub facility) acts as an origin and destination

Joseph Geunes contributed equally to this work.

✉ Joseph Geunes
geunes@tamu.edu

Mina Aliakbari
mina@tamu.edu

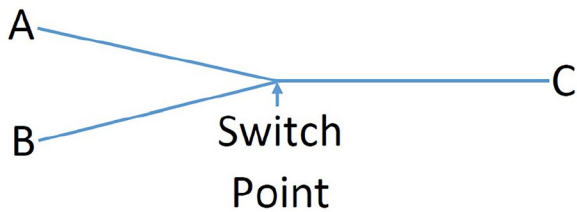
¹ Wm. Michael Barnes Department of Industrial and Systems Engineering, Texas A&M University, College Station, TX 77845, USA

point for freight trains consisting of connected railcars. These railyards often correspond to intermodal logistics freight terminals, where containers are transferred to and from tractor trailers for regional over-the-road transportation. Within a railyard, the assembly and disassembly of trains comprise a significant portion of the workload and activity within a railyard. That is, a railyard operates as a type of sorting facility, where railcars from inbound trains with different origins are disassembled, while cars with a common destination hub are assembled to form new outbound trains.

Efficiently repositioning subsets of railcars in a railyard is necessary for ensuring on-time train departures and a high degree of equipment utilization. The required reassembly of railcars to create outbound trains necessitates frequent railcar repositioning among and between various track segments in the railyard network. Generically, we will refer to any set of railcars that remains connected during a move from one position to another on the rail network as a *train*. Thus, given a current state of the railyard network (i.e., the position of each train on the network), we wish to find a set of routes and the start time for each route that minimizes the total time required until all trains reach their final destination positions on the network. We call this the multiple-train shortest-route (MTSR) problem. Based on the authors' experience with a large rail carrier, this problem is routinely solved manually using a basic set of priority rules, requiring a relatively large commitment of labor and time. The contributions of this work, therefore, include modeling and solution methods that permit obtaining good quality feasible solutions in a short amount of time, while substantially reducing the associated labor hours required to obtain these solutions. We use the term *shortest* in reference to distance or time, assuming distance can be scaled by the inverse of an average value of travel distance per unit time, in order to obtain the total time spent in travel. While trains may travel at a wide range of speeds between railyards, travel speeds are relatively small and limited within the railyard (e.g., typically less than 10-15km/hr), such that using an average within-yard travel speed tends to provide a relatively reliable estimate of total travel time between two points.

Clearly if we are able to determine the shortest route independently for each train on the empty network, and the execution of each of these routes starting at time zero results in no collisions, then the combination of these individual shortest routes solves the MTSR problem. This single-train shortest-route subproblem, however, cannot be solved directly on the rail network, as physical travel constraints on the railyard network do not always permit directly moving from each edge to all connecting edges at a node. For example, while track segments connect to one another via switch points, it is not always possible for a train to approach a switch point and directly proceed onto any other segment connected to the switch point. This results because multiple segments connected to a switch point may form an acute angle, which a train cannot physically traverse. In some cases, it is possible to ultimately traverse such an acute angle by first proceeding past the switch onto a third track segment (which must form an obtuse angle with each of the segments creating the acute angle), and then reversing direction. Whether or not such a move is possible depends on whether a sufficient amount of track is available to pass the switch and make the reverse move, which, in turn, depends on the length of the train. For

Fig. 1 Illustration of a switch point that forms an acute angle



example, the switch point illustrated in Fig. 1 creates an acute angle such that a train traveling from point A to point B cannot simply turn at the switch point; instead, it must fully pass the switch point toward point C and then reverse in order to reach point B.

It will typically be the case that solving a shortest-route problem for each train independently and initiating each such route at time zero will not produce a feasible solution, as multiple trains may require traversing a common portion of the track or switch point simultaneously (leading to collisions). Thus, beyond accounting for the physical constraints on route selection for each train, our approach to formulating and solving the MTSR problem must introduce constraints that ensure that no two trains attempt to occupy a common point on the network at the same time. Given the fact that each train may occupy multiple track segments and switch points at the same time, this introduces substantial complexity beyond the single-train shortest-route special case of the problem.

Motivated by the need to frequently solve multiple-train repositioning problems in a railyard network, and the relatively high number of labor hours typically dedicated to this task, we propose approaches for formulating and solving the MTSR problem. We provide an exact approach that models the problem as an integer program with an objective of minimizing the total makespan required for train repositioning. We also propose a heuristic algorithm based on first determining the K -shortest routes for each train in the network, and then searching for a feasible combination from among each train's K -shortest routes.

The remainder of this paper is organized as follows. The following section discusses previous relevant work in the area of train routing and its relation to our proposed model and solution methods. Section 3 then provides a formal problem definition. In Sect. 4, we provide an \mathcal{NP} -Hardness proof for the MTSR problem, after which Sect. 5 discusses an exact model and a heuristic algorithm for efficiently determining a solution. Section 6 then characterizes the advantages and disadvantages of the two approaches via a set of computational tests, and we provide concluding remarks in Sect. 7.

2 Related Literature

The application of operations research techniques to railway scheduling and train routing problems has been investigated in numerous previous works, with a majority of these focused on passenger rail traffic between train stations within a city or

country. The associated rail networks typically do not directly connect each pair of stations and, as a result, problems that consider routing passenger trains through the network of stations often arise. Some applications also exist that consider scheduling train movements within the network of track segments at a station. Lusby et al. [1] provide a comprehensive classification of the literature in this area. They categorize the models and methods at the strategic, tactical, and operational planning levels, as the scheduling time horizon decreases from a lengthy planning horizon to real-time scheduling.

In scheduling train timetables for various stations connected by tracks, Higgins et al. [2] consider a single track line that passes through different stations. Their objective is to minimize the total makespan or completion time for all trains to reach their destination. They developed a mixed-integer nonlinear programming (MINLP) formulation and applied a branch-and-bound algorithm to generate an integer feasible solution. Their branching scheme determines train priority sequencing based on current and future estimated delays, forcing one train to wait for another higher priority train to pass along a shared track segment. Carey and Lockwood [3], on the other hand, determine the priority order of conflicting trains within their proposed MIP formulation. However, the combinatorial difficulty of the resulting problem necessitated the use of heuristic approaches for fixing some non-overlapping train timetable variables, reducing the size of the formulation. In each of these approaches, parallel tracks are available at stations or crossing points where more than one train can stop at a time and allow another to pass.

Liu and Kozan [4] consider the train timetabling problem for a single-line track using a job shop scheduling approach where trains and track sections are analogous to jobs and machines, respectively. They propose an MIP formulation as well as a constructive heuristic to minimize the total makespan. In a recent paper, Lange and Werner [5] revisit this problem with fixed train routes with associated desired departure and arrival times. The authors provide an MIP formulation to minimize total tardiness, both for cases where parallel tracks are accounted for directly in the formulation, and when the consideration of parallel tracks is handled in a post-processing stage.

In addition to the single-line track train timetabling problem, other more general networks have been considered in the literature, where stations correspond to nodes, and the links between stations serve as connecting edges, producing a network that usually is not a complete graph. For these problems, an aggregated rail topology is usually defined in which the various possible routing decisions within a node (station) are ignored for simplicity, and the arcs represent the connections between these aggregated nodes [1]. The detailed network layout associated with each node is then explicitly considered in order to verify the feasibility of the aggregated schedule's inbound/outbound times (this step is usually referred to as the train platforming problem).

With this network setup, Caprara et al. [6] propose a multi-commodity flow formulation for the train timetabling problem over a network of stations using binary variables associated with each arc, flow constraints for arcs, and constraints ensuring that multiple trains avoid conflicting arcs. The authors then suggest using Lagrangian relaxation for conflicting arc constraints. A follow-up work also included constraints that account for station throughput capacity, although the test instance sizes

necessarily became smaller. Instead of associating binary variables with the selection of an arc, Cacchiani et al. [7] use such variables to select a train path, where the model's constraints prevent the selection of conflicting paths. The authors propose a branch-and-cut-and-price algorithm and a heuristic solution approach to solve the resulting problem. Their results demonstrate that the solution time becomes appreciably shorter for the path-based formulation than the arc-based one.

As noted earlier, a plan resulting from the problem's solution over an aggregated network topology must be validated at the station level for tactical verification of the proposed timetable. Lusby et al. [1] pose the intra-station level scheduling problem as a general junction train routing problem, which permits defining a general network that may include station tracks, non-station tracks (the latter of which correspond to the links under the aggregated topology), or both. As a result, the general junction train routing problem can be defined in sufficiently general terms that allow its use in addressing problems at each of the strategic, tactical, and operational levels.

Zwaneveld et al. [8] formulate an intra-station scheduling problem using a node packing formulation over a conflict graph, where nodes correspond to paths and arcs between nodes indicate pairs of conflicting paths. Their setup permits defining a conflict as a result of a shared track segment between two paths, as a result of an operational restriction (e.g., restrictions forced by maintenance schedules), or through the assignment of multiple routes to a single train. The objective of the model in [8] maximizes the total weighted benefit of the selected paths; when all path weights are equal, this corresponds to the maximum cardinality of the set of selected paths (which corresponds to the maximum throughput in the network). Their solution mechanisms include the use of valid inequalities to improve the model formulation as well as a branch-and-cut algorithm for solving the node packing problem. Their formulation selects from among a set of predetermined paths for each train rather than selecting track sections within a junction network, and thus searches for set of non-conflicting nodes (paths).

The junction train routing problem for intra-station scheduling can also be defined using a multi-commodity network flow formulation. Fuchsberger and Lüthi [9] introduce a resource tree conflict graph, where nodes correspond to resource elements (such as switches), and arcs correspond to tracks connecting the resources. Binary variables are associated with selecting each arc in the network, which, together with a travel time parameter, leads to an objective of minimizing the total travel time for trains on the network. To achieve a conflict-free solution, the authors include additional constraints that restrict the assignment of the set of conflicting arcs to at most one train.

Unlike the conflict graph, a so-called alternative graph, based on a disjunctive graph approach, can be applied in routing multiple trains. Instead of searching for non-conflicting paths, the alternative graph attempts to determine the order of operations whenever two activities are prone to conflict. Thus, the alternative graph has nodes that correspond to operations and arcs that determine the sequence of consecutive operations. This graph contains pairs of extra (alternative) arcs that determine the order of operations that would otherwise block a track segment simultaneously or that simply may not be executed at the same time for any reason. Mascis

and Pacciarelli [10] propose an alternative graph formulation to tackle the job shop problem with blocking and no-wait constraints. D'Ariano et al. [11] subsequently mapped their method to a real-time train scheduling problem. By allowing only one arc of each pair of the set of alternative arc pairs, the model determines the best sequence of trains at each conflict point.

A variant of the job shop scheduling with blocking approach for junction scheduling is proposed by Rodriguez [12], who considers real-time routing through a junction. In this paper the author defines the concept of a track circuit, which corresponds to the presence of an electrical signal on the track segments occupied by a train at any given time. A train must be delayed on its current track circuit if its subsequent track circuit contains another train. Rodriguez [12] uses a constraint programming approach to determine a set of routes without conflicting circuits with an objective of minimizing total delay.

As noted earlier, the problem of validating the feasibility of a timetabling solution at the station level is referred to as the train platforming problem, where the goal is to assign available platforms to the scheduled inbound trains. Considering each train as a node and generating the associated conflict graph that contains edges between pairs of trains that cannot be assigned to the same platform, a graph coloring approach can be used such that no adjacent nodes have the same color, and each color represents a platform. Cornelsen and Di Stefano [13] use such an approach as a feasibility check for a passenger railway aggregated inter-station routing solution. For real-time problems, when infeasibilities have been detected and require fast resolution, Pellegrini et al. [14] define a real-time traffic management problem, for which they provide a mixed-integer linear programming formulation and consider the objectives of minimizing maximum and total delay. Samà et al. [15] discuss graph-based disjunctive programming approaches for this class of problems, with an objective of minimizing the maximum delay (see also [16]).

While the approaches we have discussed can generally be tackled using branch-and-bound methods (Lusby et al. [1]), in many cases, the problem's size and complexity can restrict the likelihood that an exact method is able to produce a good solution in acceptable time. To help in addressing this problem, some research has focused on improving the quality of the linear relaxation of the mixed-integer programming formulation. For example, Fischer [17] proposes a set of ordering inequalities that eliminate commonly found fractional solutions. In addition, as an alternative to exact solution methods, numerous heuristic solution approaches have been developed to address problems of practical size. These methods include, but are not limited to, local search, Tabu search, and adaptive search algorithms.

The literature discussed thus far considered route planning and execution at the aggregate level (via the network connecting multiple stations), as well as intra-station routing and platform assignment. While these problems are often solved in the passenger-train problem domain, relevant work also exists that addresses problems unique to freight train operations. Ahuja et al. [18] introduced very large-scale neighborhood search methods for solving so-called blocking problems in freight transportation, where a block consists of a group of cars with a common destination. The creation of such blocks reduces the handling requirements in railyards, although the problem of assigning cars to common blocks is challenging due to its combinatorial nature.

The dissertation work of Schlechte [19] (and several related works, e.g., [20–22], and [23]) considered a number of large-scale optimization problems involving both freight train and passenger train routing and allocation of trains to tracks throughout a network of stations covering a geographical region. This work considers aggregated, macro-level mixed-integer optimization problems that are developed based on the results of simulating detailed micro-level operations. These works use a rail corridor between Switzerland and Italy through the Alps as a test bed to validate their solution approaches.

Bohlin et al. [24] consider a specially structured yard known as a classification yard (also known as a *hump* or *shunting* yard), in which cars on a set of parallel arrival tracks feed a set of parallel classification tracks (via a hump, using gravity to roll into the classification yard), which subsequently feed outbound tracks from which assembled trains depart. An engine pulls multiple cars onto the hump, the cars are decoupled, and then gravity and switching are used to feed the cars onto selected classification tracks. The problem they consider requires determining the assignment of classification tracks to outbound trains in order to meet a desired departure schedule. A broader discussion and overview of this class of shunting yard track assignment and engine sequencing problems can be found in [25], while more recently, Minbashi et al. [26] applied machine learning techniques to address this class of problems under uncertainty.

Several of these works predetermined a set of feasible paths for each train, while those that considered the actual station network structure explicitly did not address the complexities introduced by switch points, or the difficulties associated with repositioning trains within a railyard network containing a general structure. While these issues typically do not affect passenger train routing through stations, they become important factors to account for in freight rail train assembly and disassembly operations. Thus, to our knowledge, the most related works are perhaps Zwaneveld et al. [8], Rodriguez [12], and D'Ariano et al. [11], which consider routing trains through a network while accounting for collision avoidance and the delays introduced when trains must wait for one another to pass on commonly used track segments. However, these prior works do not account for the complexity introduced when simultaneously repositioning freight trains within a railyard via switch points. In the remainder of this paper, we address these additional challenges via the formulation and solution of the MTSR problem.

3 Problem Setting and Definition

In this section we describe a typical railyard network structure and the constraints this network imposes on train travel within the network. A railyard network consists of a set of track *segments*, where each segment either provides a connection between switches, or is part of a ramp track or support track, connected to a main line via lead track(s). A ramp track, also known as a production track, is used for loading containers onto railcars as well as unloading containers from railcars. Thus, ramp tracks must be accessible to overhead cranes used for loading and unloading operations. Support tracks, also known as storage tracks, are used to store railcars,

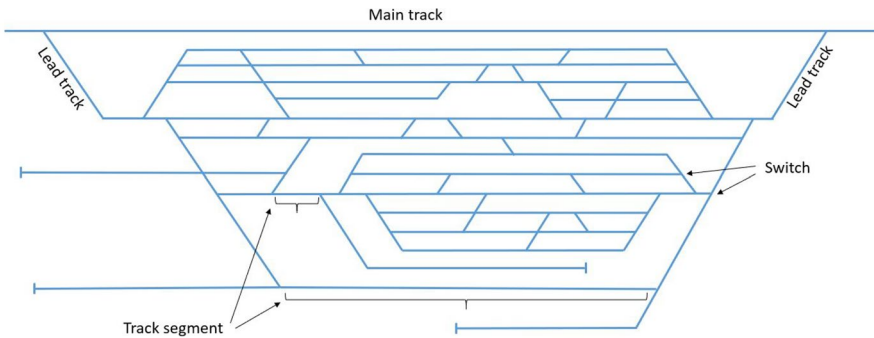


Fig. 2 Illustration of a sample railyard, with tracks and switches

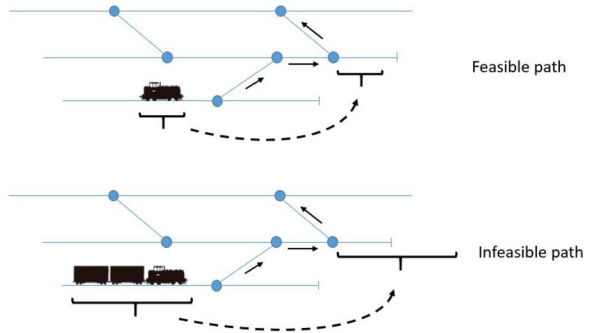
e.g., empty railcars, assembled outbound trains waiting to depart, or newly arriving inbound trains that have not yet been scheduled for unloading.

Track segments are interconnected via switch points, where more than two track segments meet, permitting travel between segments. We assume without loss of generality that at most one train may occupy a track segment at a time. If a particular section of track is sufficiently long so as to permit containing two trains at different locations on the section, we divide the section into smaller segments. We also assume without loss of generality (as in [27]) that a switch point corresponds to a point where exactly three track segments meet (if more than three segments meet at a switch point, we decompose the switch into multiple switch points separated by zero distance). At each switch point, two pairs of converging track segments form obtuse angles, and the third pair forms an acute angle. Train travel along an obtuse angle is possible without reversing direction, while traversing an acute angle requires proceeding past the switch and reversing direction (a so-called double-back move). Our network model contains a node for every switch point as well as an edge corresponding to each track segment.

Figure 2 illustrates a sample layout of a railyard with various track segments and switch nodes. We will refer to any set of railcars that remain connected during travel on the network from origin location to destination location generically as a train. Thus, a train may correspond to a single railcar or multiple connected railcars. As a result, a train may span multiple edges/nodes of the network at any point in time. This leads to complications that differentiate train routing problems from classical network routing problems. As illustrated in Fig. 3, if a train's length is smaller than the shortest track segment in the network, then a simple node-to-node path from origin to destination (which accounts for the time required for direction changes as a result of double-back moves) can be used to determine a shortest route. On the other hand, if the train's length exceeds the length of some track segments, the same path might not be feasible.

The problem we consider requires the movement of multiple trains on a rail network. It is therefore critical to track each train's end-to-end position throughout its movement in order to ensure that no two trains attempt to occupy a common track segment or switch simultaneously. This will often result in the need

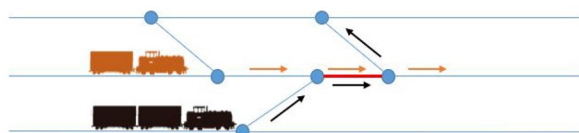
Fig. 3 The train length effect on feasibility of a path



to determine not only which path each train will take, but also points in time at which a train must wait for another train to pass before proceeding along its path. Figure 4 illustrates a case in which two trains require use of the edge highlighted in red and where one train must wait for the other in order to avoid a collision. For instance, if we allow the black train to traverse the red track segment first, the orange train must experience a delay before it can proceed onto the red segment.

We define the MTSR problem as follows. We are given the structure of a railyard network and the initial position of each of a set of trains on the network, as well as the desired destination location for each train, where a train is defined as a sequence of cars that must remain attached to one another. For each train, we seek a path in the network from its initial position to its destination position, as well as the location of the train at each point in time between its departure time and arrival at its destination. The goal of the MTSR problem is to minimize the time required to ensure that all trains are in their destination locations. To do so, we need to determine a route for each train such that the last train’s arrival time at its destination is minimized. Because of the low and consistent speeds used when repositioning cars within a railyard (e.g., typically less than 15km/hr), our model will assume that all trains move at a single speed equal to the average speed traveled within the railyard. Observe that the possibility of delays necessary for collision avoidance implies that the route chosen for each train in such a solution may or may not correspond to its shortest route in the network.

Fig. 4 Illustration of delays needed to avoid collisions



4 Problem Complexity

In this section, we show that the MTSR problem when trains may have any non-negative train lengths is \mathcal{NP} -Hard. We utilize results from Peis et al. [28], who demonstrate the strong \mathcal{NP} -Hardness of a store-and-forward packet routing problem on planar graphs. The store-and-forward packet routing problem requires routing packets of data from packet-specific origin points to destination points. In traveling along a path in the network, a packet may be delayed at an intermediate node while awaiting transmission along its path, as each link is able to transmit at most one packet per time unit. The goal is to minimize the total time required to transmit a set of packets from origin to destination points.

To demonstrate the \mathcal{NP} -Hardness of the packet routing problem, Peis et al. [28] use a reduction from the 3-BOUNDED-3-SAT problem. The 3-BOUNDED 3-SAT problem contains a set of m clauses $C_1 \dots C_m$, where the truth of each clause depends on the values of three binary variables (we can think of these as true/false variables). For example, the clause $(x_i \vee x_j \vee x_k)$ is true if any one of the variables x_i , x_j , or x_k , equals 1. Each of the variables appears in at most three clauses, and the problem is satisfiable if an assignment of values to each of the variables can be made that results in all of the clauses being true. The problem thus requires determining whether an assignment of true/false to each of the variables exists such that each clause is true. The 3-BOUNDED 3-SAT problem is strongly \mathcal{NP} -Complete.

In the proof given in Peis et al. [28], given an instance of 3-BOUNDED 3-SAT, they construct a packet routing network problem instance with a polynomial number of network vertices and packets that require routing, corresponding to each variable and clause in the 3-SAT problem instance. They also create a polynomial number of packets corresponding to each variable in the 3-SAT problem and, for each clause, a required route that must be followed through the generated vertices. They then show that a truth assignment exists for the 3-BOUNDED 3-SAT problem instance if and only if the makespan on the corresponding packet network routing problem is at most six for a planar network. Although each packet's path is predetermined in their proof, they then show that even if the routes are not predetermined for each packet, all routes other than the predetermined ones will be inferior to the prescribed route. Therefore, if routing decisions are allowed, the prescribed routes will be selected in an optimal solution. Thus, the complexity result continues to hold when routes are not predetermined. We can transform their packet network into a corresponding special case of our railway network in polynomial time, such that an ability to solve this special case of the MTSR problem in polynomial time would imply an ability to solve the 3-BOUNDED 3-SAT problem in polynomial time, which cannot be possible unless $\mathcal{P} = \mathcal{NP}$.

The solution to the corresponding special case of the MTSR problem on the rail network in the transformed problem solves any underlying instance of the corresponding packet routing problem. In the store-and-forward packet routing problem, a node can store multiple packets at the same time, although an arc can transmit at most one packet at a time (one unit of time is required to transmit a

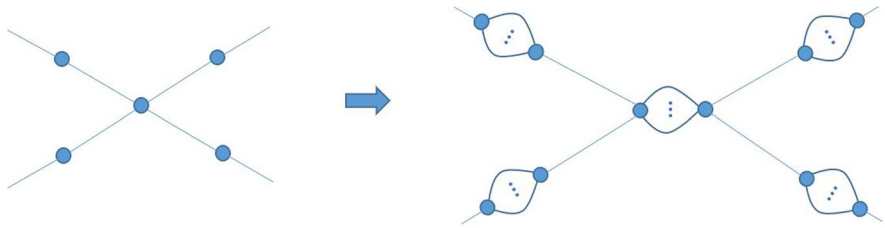


Fig. 5 Creating parallel tracks to represent a node of unlimited packet storage capacity

packet from one node to any other node that is connected to it via an arc). Conversely, in the railyard routing problem, both nodes and arcs can accommodate at most one train at any point in time. To address this issue in the problem transformation, for each node in their packet network, we create a set of n parallel tracks connecting two common outer nodes, where n is the total number of packets (see Fig. 5). Thus, a train that is located on a parallel arc in the rail network corresponds to a packet stored at the corresponding node in the packet network, and we assume that a train may travel from one node to another node via a connecting edge in one unit of time. For each packet in the packet network, we create a train in the rail network with origin and destination locations corresponding to those of the corresponding packet. Note that the network remains planar under this transformation.

Observe that packets do not have length, and we thus assume that in our corresponding rail network, all trains may be arbitrarily small in length. Because of this, although there is a possibility of conflicts at nodes after introducing parallel arcs, the time required for trains to cross one another at the nodes (i.e., going to and from the parallel edges) is negligible for trains of arbitrarily small length relative to the time it takes to travel from one node to another. In fact, when train lengths may take any nonnegative values, then conflicts at nodes are no longer relevant because the special case in which all trains have zero length eliminates the possibility of conflicts.

After this polynomial transformation from their packet network to the railyard network, we can conclude that the minimum makespan in the packet network is at most six if and only if the minimum makespan in the rail network is at most six, and both of these can occur if and only if the 3-BOUNDED 3-SAT instance is satisfiable. This implies that the MTSR problem is at least as hard as the 3-SAT problem and is thus in the class of \mathcal{NP} -hard problems.

5 Solution Approach

Efficient operations execution and high throughput levels at intermodal terminals necessitate an effective tool for planning railcar moves in the network. Thus, given an initial location of each of a set of trains on the network, as well as a desired destination location for each train, we wish to determine a sequence of feasible moves for each train that minimizes the time required until all trains are in their

final destination positions. Section 5.1 first provides a base mathematical model for this problem by formulating it as a large-scale 0-1 integer program. Section 5.2 generalizes this base formulation to ensure that a train may only move if it has an attached locomotive. While the resulting model is useful for solving small problem instances and permits providing a formal mathematical problem statement, the problem's complexity (and the required number of binary variables) implies that it is unlikely that it can be used to obtain solutions to problems of practical size in acceptable computing time. Because of this, Sect. 5.3 proposes a heuristic solution method based on identifying a feasible combination of solutions to the K -shortest routes problem (along with any required delays for each train's departure to ensure feasibility) for each train. We later provide the results of a set of computational tests in Sect. 6 that demonstrate the effectiveness of this heuristic method in providing high-quality solutions in acceptable computing time.

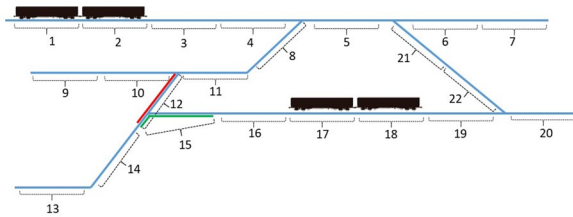
5.1 Integer Programming Model

This section presents a large-scale 0-1 integer programming (IP) model for the MTSR problem. Our IP modeling approach discretizes both space and time in order to provide a model representation of the problem and a benchmark against which we can compare heuristic solution methods for small to medium problem sizes. The input to this problem consists of a railyard network layout and a set of trains, each with origin and destination locations and an associated train length. We wish to determine a set of collision-free routes that minimizes the total time required for all trains to reach their destination locations.

As noted earlier, a railyard consists of a set of connected track segments. Our modeling approach decomposes each track segment into a discrete set of slots, i.e., each track segment is divided into a set of non-overlapping subsections called slots. This approach is needed to account for the location of each train at all points in time. Our IP model assumes a single standard slot size, corresponding to a standard car length, such that each slot can accommodate at most a single railcar at any point in time. Note that the most commonly used freight railcars in the US are 50 to 60 feet, while standard 53 foot well cars are commonly used in intermodal transportation to carry trailers that can also be transported via truck. Thus, using a 60-foot standard slot length tends to provide a good approximation for the majority of railyard operations (when accounting for the average distance between cars on a single train). In general, using a slot length equal to the largest car length on the network will lead to overestimates of train lengths on the network (for example, our model will assume that a sequence of ten 55-foot cars occupies ten sixty-foot slots, and is therefore 50 feet longer than the actual train).

Figure 6 illustrates a network representation of track segments in a railyard into slots that can accommodate railcars. We create an index number for each slot, thus characterizing the railyard as a set of connected slots of equal length. Observe that a track segment with a total length that is not equal to an integer multiple of the standard slot size will necessarily have track portions that are not covered by indexed slot numbers. Using our prior example, any track segment that is not an integer multiple

Fig. 6 Breaking a track into numbered slots



of 60 feet in length will have some portion uncovered by slots. If the set of indexed slots on the segment is equidistant from both ends of the segment, then there will be less than 30 feet of un-slotted track on either end of a segment of track. Because such track portions are quite small relative to the average segment length, we assume that the extra time required for a train to traverse such uncovered segments is negligible relative to its total travel time.

We next specify how slots are interconnected. For each slot number, we determine the set of slot(s) directly accessible to the right of the slot, and the set of slot(s) directly accessible to the left. In Fig. 6, for example, the slots accessible to the left of slot 11 are {10, 12}, while the slot to the right of slot 11 is {8}. Using this approach characterizes the feasible set of adjacent slots to which a train may move from any given slot during a unit of time. As Fig. 6 illustrates, it is possible for two slots to cover a common track portion next to a switch and then separate on either side of the acute angle at the switch. In particular, in the figure, slots 12 (marked in red) and 15 (marked in green) overlap to the left of a switch. As a result, slots 12 and 15 are both immediately accessible to the right of slot 14, while the slots to the right of 12 and 15 are different and equal to 11 and 16, respectively. This effectively corresponds to a virtual movement of the switch to the left end points of slots 12 and 15, and permits considerable flexibility in creating and assigning slot numbers to a railyard network.

For ease of exposition, we assume a network topology that permits establishing “right” and “left” travel directions. This is possible for a railyard network without any acute-angle free cycles by using the following conventions.¹ Starting from a node i , we move in a direction that we establish as the “right” direction by convention. We designate this direction to all slots that we can traverse directly (without using any double-back moves). We do the same for the opposite direction, calling this the “left” direction. As we move along a given direction, any time we hit a node, if we can continue traversing an arc without double-backing, we then apply the same

¹ A cycle in a rail network that does not require a train to traverse an acute angle using a double-back move is called an acute-angle-free (AAF) cycle. Such AAF cycles are sometimes used in a rail network to reverse a train’s orientation on the network. We assume that the rail network on which we solve the MSTR problem does not contain such AAF cycles. Thus, any existing such cycles in the network can be partially blocked to prevent AAF cycle traversal when solving an instance of the MSTR problem. Note that any route taken by a train that uses an AAF cycle without reversing orientation is dominated by a path that does not traverse the cycle. If a train on the network requires reversing its orientation, we assume the train travels to an AAF cycle and completes its orientation reversal prior to solving the MSTR problem instance in a preprocessing routine.

direction convention to the slots adjacent to the node; otherwise, we must reverse the direction if a double-back move is required. Note that associated with any right direction move from node i to node $i + 1$ is a corresponding left direction move from node $i + 1$ to node i . Moreover, any two arcs that form an acute angle in the network receive the same direction convention with respect to the node at which they meet. This convention determines the relative direction of adjacent slots to each node.

It is not uncommon that some track segments within a railyard are temporarily blocked and cannot be used during a planned set of train moves. For example, during a planned set of moves, some tracks may be blocked off for loading or unloading operations, or for maintenance operations. Such track segments and associated slots are thus excluded when defining the network layout and the set of adjacent slots available at a node. To characterize the position of each train in the network at all points in time, we create an index number for each car in the train, and define a train by the set of numbered cars it contains. We next define the variables and parameters we use to model the MSTR problem. Then we formulate the problem as an integer program and provide a characterization of the objective function and associated constraints.

To define a problem instance, we first define the network as a set S of standard-length slots indexed by s , and let R_s (L_s) denote the set of adjacent slots to the right (left) of slot s , for all $s \in S$. After indexing cars, suppose we have a set C of cars on the network that are part of a train that requires movement to a new location in the network. Given a train containing n connected cars, note that each of the first $n - 1$ of these cars (going from left to right) has an associated car connected to its right side (i.e., such that if the car is in slot s , the car connected to its right side must always occupy a slot in the set R_s throughout any sequence of car movements). Let CC denote the set of all cars in the network such that another car remains connected to its right side throughout all movements. Observe that if there are N trains requiring moves, then $|CC| = |C| - N$, i.e., for each of the N trains, exactly one car (the right-most car in the train) is not in the set CC . For each car c , let s_c^o and s_c^d denote the car's original and destination slots, respectively, and for each $c \in CC$, let RH_c denote the car that remains connected to its immediate right-hand side.

We use a discrete time model, assuming that any car within a slot can move to an adjacent slot in a single time unit. Let T denote a finite time horizon length that serves as an upper bound on the total number of time units required to complete all required moves (for example, if, for each train requiring a move, we determine the shortest route from origin to destination on the network after blocking off all track segment containing other trains, then we can set T equal to the sum of all associated train shortest route lengths). Our mathematical formulation requires defining decision variables that keep track of each car's location at all time points, the movements of cars between slots, and whether or not each car has reached its final destination location. We therefore let

- $X_{cst} = 1$ if car c is in slot s at time t , and 0 otherwise, for $c \in C, s \in S, t = 0, \dots, T$;
- $Z_t = 1$ if the process is complete at time t (i.e., no additional moves are required), and 0 otherwise, for $t = 1, \dots, T$;

- $GR_{st} = 1$ if a car in slot s moves to an adjacent slot to its right at time t , and 0 otherwise, for $s \in S, t = 0, \dots, T$; and
- $GL_{st} = 1$ if a car in slot s moves to an adjacent slot to its left at time t , and 0 otherwise, for $s \in S, t = 0, \dots, T$.

Using these definitions, we next formulate our integer programming model. Recall that the objective is to minimize the makespan required for all train moves, i.e., the earliest time at which all railcars occupy their destination locations.

$$\text{Minimize } \sum_{t=1}^T tZ_t \tag{1}$$

Subject to:

$$X_{cs^0} = 1, \quad c \in C, \tag{2}$$

$$X_{cs,t+1} \leq \sum_{i \in L_s} X_{cit} + X_{cst} + \sum_{i \in R_s} X_{cit}, \quad c \in C, s \in S, t = 0, \dots, T - 1, \tag{3}$$

$$\sum_{s \in S} X_{cst} = 1, \quad c \in C, t = 1, \dots, T, \tag{4}$$

$$\sum_{c \in C} X_{cst} \leq 1, \quad s \in S, t = 1, \dots, T, \tag{5}$$

$$Z_t \leq X_{cs^d t}, \quad c \in C, t = 1, \dots, T, \tag{6}$$

$$\sum_{t=1}^T Z_t \geq 1, \tag{7}$$

$$X_{cst} \leq \sum_{i \in R_s} X_{RH_{cit}} \leq 2 - X_{cst}, \quad c \in CC, s \in S, t = 0, \dots, T, \tag{8}$$

$$\sum_{i \in R_s} X_{ci,t+1} + X_{cst} \leq GR_{st} + 1, \quad c \in C, s \in S, t = 0, \dots, T - 1, \tag{9}$$

$$\sum_{i \in L_s} X_{ci,t+1} + X_{cst} \leq GL_{st} + 1, \quad c \in C, s \in S, t = 0, \dots, T - 1, \tag{10}$$

$$GR_{st} + \sum_{i \in R_s} GL_{it} \leq 1 \quad s \in S, t = 0, \dots, T, \tag{11}$$

$$GL_{st} + \sum_{i \in L_s} GR_{it} \leq 1 \quad s \in S, t = 0, \dots, T, \quad (12)$$

$$X_{cst} \in \{0, 1\}, \quad c \in C, s \in S, t = 0, \dots, T, \quad (13)$$

$$Z_t \in \{0, 1\}, \quad t = 1, \dots, T, \quad (14)$$

$$GR_{st} \in \{0, 1\}, \quad s \in S, t = 0, \dots, T, \quad (15)$$

$$GL_{st} \in \{0, 1\}, \quad s \in S, t = 0, \dots, T. \quad (16)$$

The objective function (1) minimizes the earliest time at which each car occupies its destination slot, while Constraint set (2) sets the initial locations of the cars and locomotives. Constraint set (3) ensures that each car either moves to or from an adjacent slot at each time step, or remains in its previous location. Constraint set (4) requires that each car occupies exactly one slot at each time period, while Constraint set (5) allows at most one car to occupy a slot in each period. Constraint set (6) allows Z_t to equal one when every railcar is in its destination location, and Constraint (7) ensures that the final state is achieved at some point in time. Constraint (8) requires that all right-connected cars remain connected throughout all time periods. Constraints (9)–(12) together keep track of car moves to adjacent slots, and ensure that if a move to the right (left) occurs for a car in a given slot, no car in a right-adjacent (left-adjacent) may move left (right). Finally, constraints (13)–(16) define the binary variables.

The above integer programming model for the MSTR problem provides a solution that determines a period-by-period location of each railcar, along with the minimum time at which all railcars occupy their destination locations. The solution ensures a collision-free plan of movement for railcars because at most one railcar may occupy a slot in every time period. In the following, we provide an extension to this model that specifically accounts for locomotive requirements.

5.2 Integer Programming Model — Locomotive Extension

The integer programming model in the previous section considered trains and their locations on the railyard network. A train cannot move, however, without having a locomotive attached in order to push or pull the train. In other words, in the previous mathematical model, we implicitly assumed that each train has a locomotive already attached to it to pull or push the railcars. However, in many cases, a set of railcars may exist on the network without an attached locomotive. In fact, the number of locomotives that perform within-railyard operations is typically limited, and a key consideration requires assigning locomotives to trains that require movement before they can be moved. If the movement and assignment of locomotives must be considered within the multi-train repositioning model, additional definitions and constraints are needed for the integer programming model. In the following, we first describe additional parameters and

variables needed to account for this model extension. Later, we discuss how the previous mathematical model can be used to account for locomotive requirements.

In addition to the parameters defined in the previous section, we define *Loco* as the set of locomotives (i.e., the set of railcars that are locomotives). We also want to prevent the movement of any non-locomotive railcar(s) if no locomotive attached to the car(s); therefore, we let:

- $E_{st} = 1$ if slot s is accessible to a locomotive from the right, 0 otherwise, for $s \in S, t = 0, \dots, T$, and;
- $W_{st} = 1$ if slot s is accessible to a locomotive from the left, 0 otherwise, for $s \in S, t = 0, \dots, T$.

Note that a locomotive is *accessible* from a slot on the network at a given time t if a connected sequence of railcars exists between the slot and a locomotive at time t . Using these new definitions, we reformulate our integer programming model. Note that the objective of minimizing the makespan required for all train moves is the same as in the previous model.

$$\text{Minimize } \sum_{t=0}^T tZ_t \tag{17}$$

Subject to: (2)-(16)

$$E_{st} \leq \sum_{c \in C} X_{cst}, \quad s \in S, t = 0, \dots, T, \tag{18}$$

$$W_{st} \leq \sum_{c \in C} X_{cst}, \quad s \in S, t = 0, \dots, T, \tag{19}$$

$$E_{st} \leq \sum_{c \in Loco} X_{cst} + \sum_{i \in R_s} E_{it}, \quad s \in S, t = 0, \dots, T, \tag{20}$$

$$W_{st} \leq \sum_{c \in Loco} X_{cst} + \sum_{i \in L_s} W_{it}, \quad s \in S, t = 0, \dots, T, \tag{21}$$

$$GR_{st} + GL_{st} \leq E_{st} + W_{st}, \quad s \in S, t = 0, \dots, T, \tag{22}$$

$$2GR_{st} \leq E_{st} + W_{st} + \sum_{i \in R_s} (E_{i,t+1} + W_{i,t+1}), \quad s \in S, t = 0, \dots, T - 1, \tag{23}$$

$$2GL_{st} \leq E_{st} + W_{st} + \sum_{i \in L_s} (E_{i,t+1} + W_{i,t+1}), \quad s \in S, t = 0, \dots, T - 1, \tag{24}$$

$$\sum_{c \in C} (X_{cs,t+1} - X_{cst}) \leq E_{s,t+1} + W_{s,t+1}, \quad s \in S, t = 0, \dots, T - 1, \tag{25}$$

$$E_{st} \in \{0, 1\}, \quad s \in S, t = 0, \dots, T, \tag{26}$$

$$W_{st} \in \{0, 1\}, \quad s \in S, t = 0, \dots, T. \quad (27)$$

Constraints (18)–(21) determine whether or not each slot has an accessible locomotive. A slot is accessible to a locomotive, and therefore available to carry out a move, if either the immediate slot to its right or left is accessible to a locomotive, or a locomotive occupies that slot. Constraint (22) ensures that a move to the right or left of a slot is possible if and only if that slot is accessible to a locomotive either from the right or left. Constraints (23)–(25) require locomotive accessibility during the entire duration of a move. In other words, if a given railcar moves at some time step, its initial and subsequent locations' slots must remain accessible to a locomotive.

This mathematical model is more general because, in addition to tracking period-by-period train (and railcar) locations while ensuring collision-free routing, this integer programming model considers the role of locomotives as well. However, this extension makes the model even more complicated and larger in terms of the number of variables and constraints. Alternatively, we might use the mathematical model proposed in Sect. 5.1 to handle this locomotive extension heuristically. To see this, suppose we have k locomotives and wish to move $N > k$ trains to new positions. We first assign the k locomotives to a subset of k trains that will be moved in the first iteration. By determining the initial and final location of locomotives (this is subject to manual determination), we can solve an instance of the model proposed in Sect. 5.1 to determine the minimum time required to move the k locomotives to the k trains. Thereafter, we can use the same model to solve a multi-train repositioning problem for the k trains with the attached locomotives, because we have already ensured that those trains have a locomotive attached. In other words, at each iteration, we decompose the mathematical model into two steps: first attaching locomotives to trains, and then moving the trains with locomotives attached. After one iteration, we then have $N - k$ trains that still require moves, and we run another iteration of the two-step procedure, first moving locomotives to trains, and then moving the trains with the locomotives attached. This procedure is repeated as needed until all of the N trains have been moved.

Both integer programming formulations are useful for solving small problem instances and providing a mathematical model of the problem we wish to solve. However, the large number of binary variables required in the formulation makes its use in solving problems of practical size unlikely. The scale of the problem grows with the number of cars, the number of slots, and the time horizon. Because of this, the following section proposes a heuristic solution method that enables obtaining solutions for real-world problem sizes in acceptable computing time.

5.3 Heuristic Approach

A naïve heuristic approach might begin by determining the shortest route for each individual train on the subnetwork obtained by blocking off all track segments occupied by

other trains' origin and destination locations (we will refer to the resulting network as a train's associated blocked subnetwork). If a feasible route exists for each train on its associated blocked subnetwork, then a feasible solution for the MTSR problem is obtained by sequentially moving each train along its shortest route in its associated blocked subnetwork. While the corresponding solution provides an upper bound on the minimum makespan equal to the sum of all trains' shortest route lengths on the associated blocked subnetwork, this solution is likely to be far from optimal, as this solution does not allow simultaneous movement of multiple trains. While, in the most general case, we cannot *a priori* guarantee that a feasible route exists for each train on its associated blocked subnetwork, we will assume for tractability that at least one feasible route exists for each train in its associated blocked subnetwork (thus the network is not so populated with trains such that gridlock may arise, which is typical of railyard networks in practice).

Our goal is to execute a set of conflict-free simultaneous routes, rather than routing trains sequentially. A conflict arises when two or more trains seek to traverse a common track segment simultaneously. That is, because each train's route consists of a sequence of track segments, two trains whose routes contain a common segment have a potential for conflict. It may be the case, however, that this shared track segment is not required by both trains at the same time. Therefore, we need to keep track of the sequence of track segments contained within a train's planned route, as well as the time during which the train occupies each segment. Note that conflicts may be resolved by delaying one or more trains until the conflicting track segment is available.

Our heuristic solution method applies the following logic, assuming we wish to provide a solution to the MSTR problem with m trains, indexed from 1 to m . We first determine the shortest route for the first train on its associated blocked subnetwork, and execute this route beginning at time zero. We next determine the K shortest routes for each of the remaining $m - 1$ trains in the network for some integer $K > 0$. For the second train, among its K shortest routes, we select the one that minimizes its arrival time at its destination location when initiating the route at time zero, and delaying train 2 when conflicts arise with train 1. We then do the same for the third train, selecting the route that minimizes its arrival time at its destination location when initiating the route at time zero, and delaying train 3 when conflicts arise with train 1 or 2. We proceed in this manner for all remain trains in sequence, delaying a train that has a conflict with any lower indexed train number.

Because the routes are determined sequentially in train index order, the sequence in which trains are indexed may have a substantial impact on the quality of the solution obtained. And, because $m!$ index sequences are possible, it is computationally impractical to evaluate all possible index sequences. As a result, we require a strategy for prioritizing railcars in order to determine an effective index sequence. One such potential strategy would be to sequence trains based on their shortest route lengths in nonincreasing order. The idea behind this sequencing approach is to ensure that those trains with longer routes are less likely to experience delays. Another potential strategy would be to consider the reverse order, i.e., sequencing trains based on shortest route lengths in nondecreasing order. This approach would take advantage of the fact that those trains with shorter routes are less likely to create conflicts with other trains scheduled after them. Another strategy that may be applied in practice would establish train sequencing priorities based on subsequently

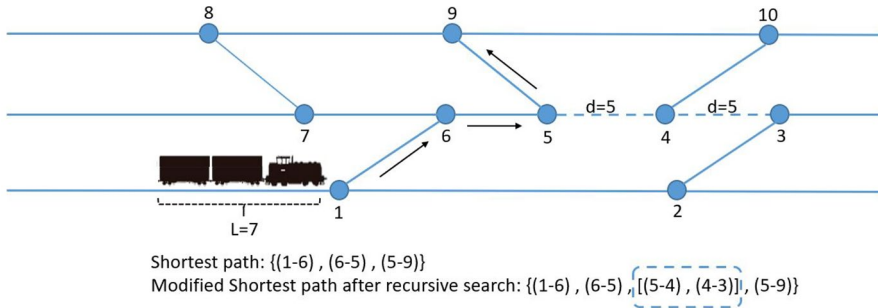


Fig. 7 Adjusting shortest path to account for arcs utilized for double-back moves

scheduled operations, e.g., those trains with earlier scheduled departures and/or promised due dates would be given higher priority.

Determining the K shortest routes for each train on its associated blocked subnetwork requires an ability to determine the shortest train route in a railyard network. To do this, we apply the shortest train route algorithm proposed in Aliakbari et al. [27], which runs in $\mathcal{O}(n^2)$ time, where n corresponds to the number of switch nodes in the network. With this method to determine the shortest route, we then apply the method of Yen [29] to determine the K shortest routes for each train.

Recall that when a train performs a double-back move, the length of the train moves past a node and then reverses direction in order to traverse an acute angle. In performing such a move, the track segments occupied by the train immediately after passing the node and before reversing are not strictly contained within the node path between the train’s origin and destination. In addition, there may be several choices for the set of track segments that may be used to accommodate the double-back move. Furthermore, preventing track-segment conflicts requires accounting for each track segment the train occupies at all times. Hence, at each switch point along a train’s shortest route where a double-back move is performed, we apply a recursive function (e.g., depth-first search) to determine the set of track segments the train will occupy while performing the double-back move. We refer to these edges in the network as double-back edges. These edges must be accounted for in the train’s route in order to account for all occupied edges at all times. Figure 7 illustrates this concept. Suppose that the train shown (of length 7) will execute a route containing nodes $1 \rightarrow 6 \rightarrow 5 \rightarrow 9$. This sequence requires a double-back move at node 5. This move is feasible because the network contains an acute-angle-free path to the right of node 5 that is at least 7 units in length (in fact, it contains two such paths). Thus, in addition to the route occupying edges (1, 6), (6, 5), and (5, 9), the train will need to occupy edge (5, 4) (which has length 5) and either edge (4, 3) or (4, 10) in order to double-back at node 5 (the figure illustrates the choice of edge (4, 3) instead of (4, 10); thus edges (5, 4) and (4, 3) are added to the train’s route).

Accounting for track segment conflicts requires associating a time interval with each train-segment pair. We assume all trains travel at the same speed, i.e., each train can move one unit of distance per unit time. Therefore, if we know a track segment’s length, the time a train begins travel on the segment, and the train’s length, we can

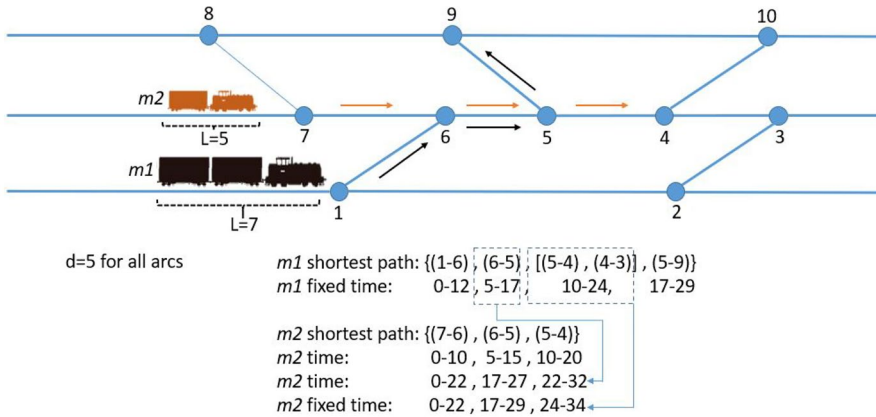


Fig. 8 Adjusting path time based on required delay

easily determine the period of time during which the train occupies the segment. This information is required in order to determine the amount of delay time introduced when a train approaches a track segment on its route that is occupied by some other train. In addition to the edges that are traversed by a train along its path, we account for the time during which double-back edges are occupied by considering each such edge as occupied during the time required to complete the double-back move.

To illustrate this idea, consider the two trains m_1 and m_2 in Fig. 8 (assume for convenience that m_1 has priority by virtue of its lower index number). Suppose the shortest route for train m_1 consists of $1 \rightarrow 6 \rightarrow 5 \rightarrow 9$, while that for train m_2 consists of $7 \rightarrow 6 \rightarrow 5 \rightarrow 4$. Because m_1 must double-back at node 5, it also uses edge (5, 4) to make this move. Thus, m_1 and m_2 both require using edges (6, 5) and (5, 4), creating a potential conflict. If m_1 departs at time 0, it will occupy edge (6, 5) from time 5 to time 17, and will occupy edge (5, 4) from time 10 to time 24. If m_2 departs at time 0, it will occupy edge (6, 5) from time 5 to time 15, which conflicts with m_1 's use of this edge. We thus delay the time m_2 begins on this edge by 12 time units until time 17 (when m_1 completes traversing this edge). After incorporating this delay, m_2 arrives at edge (5, 4) at time 22. However, this edge is not completed by m_1 until time 24, introducing an additional delay of 2 time units for m_2 before it can proceed along this edge. Train m_2 thus experiences a total delay of 14 time units along its path in order to allow m_1 to complete its usage of conflicting edges. We update the times m_2 occupies the edges along its route as well as the total time associated with the route accordingly. The resulting adjustments to the times during which m_2 occupies each edge along its route are shown in Fig. 8.

This route-delay procedure is then applied to each of the K shortest routes for m_2 . After adjusting each of the K shortest route lengths for m_2 , we select the route with the earliest completion time and choose this as the fixed route for m_2 . This same procedure is then applied to each additional train in index order, introducing delays for each route as necessary, and selecting the shortest of the K routes for the train after accounting for necessary delays within each route. Observe that our approach begins each train's route as early as possible, only delaying a train as needed, i.e.,

when it arrives at a conflicting edge. An alternative approach would delay the start time of the train by the total amount of delay required, resulting in the same destination arrival time for the train. In our implementation, we chose the former convention, using the non-delay scheduling approach discussed in Kanet [30], although neither approach is guaranteed to dominate the other in general.

Our heuristic approach, therefore, transforms the railyard into a shortest path network and first determines the K shortest routes for each train on its associated blocked subnetwork, accounting for edges occupied during double-back moves along each route. For each train in index sequence, and for each of the K shortest routes, we adjust each route length by introducing delays needed to resolve conflicts with higher indexed trains, and choose the route with the smallest destination arrival time. The pseudo-code in Algorithms 1 and 2 below encodes our heuristic approach, where M denotes the set of trains, K denotes the number of shortest routes considered for each train, and $FinalRoutes$ contains the list of finalized train routes. Observe that the time required to solve the K shortest paths problems for $|M|$ trains is no worse than $\mathcal{O}(K|M|n^2)$, while the route-delay conflict resolution phase of our heuristic algorithm requires $\mathcal{O}(K|M|^2)$ time. Thus, the heuristic time is bounded by $\mathcal{O}K|M| \max\{n^2, |M|\}$. Because we typically expect $n^2 \gg |M|$, we write this as $\mathcal{O}(K|M|n^2)$. Thus, the heuristic approach contains low-order polynomial-time complexity and, therefore, can be expected to scale seamlessly to problems of very large size.

Algorithm 1 FindFinalRoutes(M, K)

```

for train  $m \in M$  do
  Find  $K$  shortest routes and sort in nondecreasing order of length:  $P^m = \{P_1^m \dots P_K^m\}$ 
  for path  $P_i^m \in P^m$  do
    for edge  $c \in P_i^m$  do
      if train  $m$  performs double-back after  $c$  then
        Use Depth-First-Search to determine edges train  $m$  uses in double-back move
        Add double-back edges to  $P_i^m$  after edge  $c$ 
      end if
    end for
    Let train  $m$  start time equal zero
    for edge  $c \in P_i^m$  do
      Determine begin and end time for train  $m$  on edge  $c$ 
    end for
  end for
end for
Add  $P_1^1$  to  $FinalRoutes$ 
for (train  $m \in M$  and  $m \neq 1$ ) do
  for route  $P_i^m \in P^m$  do
    call ResolveConflict( $P_i^m, FinalRoutes$ )
  end for
  Let  $P_{BestRoute}^m$  be the shortest route in  $P^m$ 
  Add  $P_{BestRoute}^m$  to  $FinalRoutes$ 
end for

```

Algorithm 2 ResolveConflict($P_i^m, FinalRoutes$)

```

for route  $p \in FinalRoutes$  do
  for edge  $c$  in route  $P_i^m$  do
    if the time interval for edge  $c$  in  $p$  and  $P_i^m$  overlaps then
      Calculate  $delay$  necessary to resolve conflict
      Adjust occupancy time of edge  $c$  and its successor edges in  $P_i^m$ 
    by adding  $delay$ 
      Extend occupancy time of edges spanned during delay time by
       $delay$  units
    end if
  end for
end for
end for
    
```

We can determine a *worst-case* bound on the performance of the application of this heuristic to a problem with $|M|$ trains as follows. Let P_1^m denote the shortest path for train $m \in M$ and assume that a feasible solution exists in which each train traverses its shortest path sequentially, i.e., one at a time in sequence. Letting $j = \arg \max_{m \in M} \{P_1^m\}$, observe that P_1^j provides a lower bound on the time required for all trains to reach their final destinations. Then a bound on the optimality gap can be expressed as $\frac{\sum_{m \in M} P_1^m - P_1^j}{P_1^j} \times 100\% = \frac{\sum_{m \in M, m \neq j} P_1^m}{P_1^j} \times 100\%$. For the special case in which all trains' shortest path lengths are identical, this bound equals $(|M| - 1) \times 100\%$. More generally, let $f = \frac{P_1^j}{\sum_{m \in M} P_1^m}$ (which implies $1 - f = \frac{\sum_{m \in M, m \neq j} P_1^m}{\sum_{m \in M} P_1^m}$). Then the optimality gap bound can be expressed as equal to $\frac{1-f}{f} \times 100\% = \left(\frac{1}{f} - 1\right) \times 100\%$.

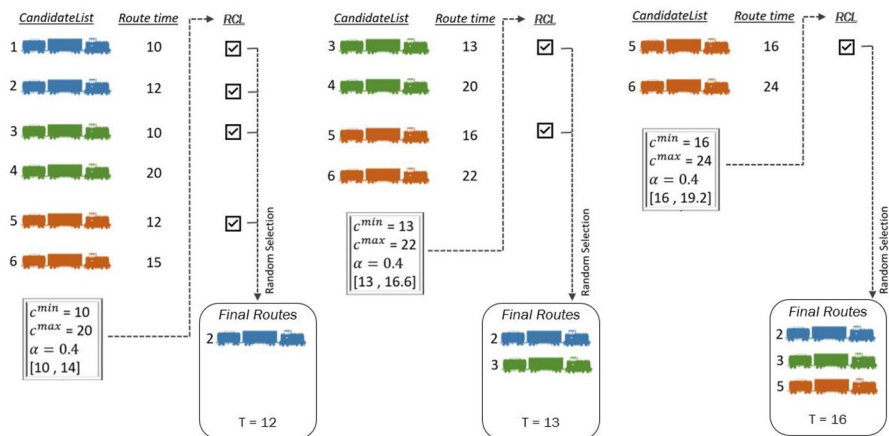


Fig. 9 Greedy randomized selection of routes

5.4 GRASP Extension

This section discusses the GRASP metaheuristic approach and how it can be incorporated into our heuristic approach. GRASP (Greedy Randomized Adaptive Search Procedure) is a multi-start metaheuristic (i.e., the procedure is initiated at a number of different starting points). At each iteration, an initial feasible solution is identified and a local minimum is found by exploration of a defined neighborhood of solutions. The best solution among all locally optimal solutions found using different starting points is then retained as the heuristic solution. Resende and Ribeiro [31] explain the basics of implementing GRASP and how to link it to other metaheuristics for an improved result. In the following, we describe our adoption of the GRASP concept for improving our heuristic approach using a greedy randomized approach.

Resende and Ribeiro [31] define a restricted candidate list (RCL) as a list of elements (routes) whose incorporation within a partial solution at each step results in the smallest increase in the partial solution's objective function value. We adopt their value-based procedure to form such an RCL. They define $c(e)$ as the incremental cost of each element. An element in our case corresponds to a particular route for a given train, and we define cost as the route's total time (considering delays and double-backs). At each iteration, routes with incremental cost within the selection interval $[c^{min}, c^{min} + \alpha(c^{max} - c^{min})]$ are selected to form the RCL, where $\alpha \in [0, 1]$. A route from the RCL is then randomly selected and incorporated within the partial solution. In other words, this interval determines the routes with the smallest increase in the partial objective function. As α grows, an increasingly randomized selection occurs, whereas with smaller values of α we restrict the RCL to the best routes in terms of total time. Figure 9 depicts the basic steps of this approach.

Note that in this algorithm, at each partial solution, the selection interval is updated. The advantage of this approach is that it permits consideration of routes that do not locally minimize total time, but which may provide the possibility of a lower final objective function value. In this way, the shortest route is not necessarily selected at each step. However, as we discussed before, if we set α equal to zero, the route with minimum total time is indeed selected at each step. In our experiments we set $\alpha = 0.33$. Algorithm 3 provides the pseudo-code for this approach, including a multi-start step that solves the problem multiple times and reports the best overall solution.

Algorithm 3 Greedy Randomized Heuristic

```

CandidateList  $\leftarrow \emptyset$ 
for train  $t \in T$  do
  Find  $K$  shortest routes for  $t$ :  $P^t = \{P_1^t \dots P_K^t\}$ 
  for path  $P_i^t \in P^t$  do
    for edge  $c \in P_i^t$  do
      if train  $t$  performs double-back after  $c$  then
        Use Depth-First-Search to determine edges train  $t$  uses in
        double-back move
        Add double-back edges to  $P_i^t$  after edge  $c$ 
      end if
    end for
    Add  $P_i^t$  to CandidateList
  end for
end for
BestSolution  $\leftarrow \emptyset$ 
BestValue  $\leftarrow \infty$ 
for  $m = 1, \dots, \text{MaxIterations}$  do
  FinalRoutes  $\leftarrow \emptyset$ 
  CandidateList $m$   $\leftarrow$  CandidateList
  while CandidateList $m$   $\neq \emptyset$  do
    for route  $r$  in CandidateList $m$  do
      Determine begin and end time for  $r$  on all of its edges
      call ResolveConflict( $r$ , FinalRoutes)
      set  $c(r)$  as the end time of last arc of  $r$ 
    end for
    find  $c^{\min}$  and  $c^{\max}$ 
    calculate interval  $\leftarrow [c^{\min}, c^{\min} + \alpha(c^{\max} - c^{\min})]$ 
    define RCL as the set of routes whose cost lie within the interval
    selectedRoute  $\leftarrow$  randomly select a route from RCL
    add selectedRoute to FinalRoutes
    update CandidateList $m$  by removing the  $K$  routes associated to
    selectedRoute's respective train
  end while
  if maximum time among routes in FinalRoutes is less than BestValue
  then
    BestSolution  $\leftarrow$  FinalRoutes
    BestValue  $\leftarrow$  maximum time among routes in FinalRoutes
  end if
end for

```

6 Computational Results

This section presents computational results for randomly generated sample railyard networks and examines the relative performance of the optimization model solution and that provided using the GRASP Extended (GE) heuristic, with a total of 100 restarts per problem instance and $K = 4$ and $\alpha = 0.33$. Instances of the optimization model were solved using CPLEX callable libraries in JAVA, while the heuristic solution approach was coded in JAVA. All solutions were run on a machine with an Intel Core i7 processor at 2.50 GHz, and with 8GB of RAM. The solver's parallel processing allowed up to four threads, which was determined by the computer's logical processors.

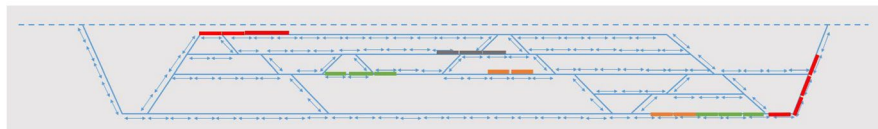


Fig. 10 A test instance sample network. The colored railcars illustrate a set of train origins and destinations; in this instance, three trains must move from their origins to destinations, while one train (shown in gray) requires no movement. The main line is represented by a dashed line and is not generally considered for in-railyard moves and, therefore, does not require slot assignments

A sample instance of a network representation for the exact approach is presented in Fig. 10, which shows the slots and sample origin and destination locations for railcars. For each sample network representation developed for testing, multiple origin and destination pairs were considered for railcars of various lengths. The number of slots needed for these sample networks ranged from as few as 25 to as many as 132. Our test instances considered a total number of cars and/or trains on the network ranging from 4 to 15, while the required time horizon ranged from 25 to 80. Note that in practice, in order to determine the time horizon required for the exact method, we can run the heuristic and use the resulting number of periods required as an upper bound on the objective function for the exact approach. However, in these tests, our goal was to have an independent and unbiased comparison between the two approaches. Therefore, we set the time horizon without knowledge of the bound given by the heuristic solution, and solely based on the visual representation of the network instance. Our test instances contained at least two trains (two sets of railcars) such that their routes contains a potential conflict. In other words, although the number of railcars, trains, and network structures were randomly created, the origins and destinations for railcars were assigned to ensure shared paths or track segments for different trains. This ensures instances that may result in either train delays or the use of parallel tracks, such that a train may use an alternative path where available (i.e., a path that is not the train's shortest path from origin to destination).

Our goal in creating randomly generated problem instances was to benchmark the heuristic solution methods against a known optimal solution where possible. As discussed in Sect. 5.3, the heuristic method we propose contains low-order polynomial-time complexity, and can therefore be implemented to provide fast solutions to large problems in practice. The exact approach, on the other hand, as our computational tests show, has difficulty in solving the comparably small-size test cases we considered. Note that although the network size of the problem instances we considered may not be as large as those found in real-sized railyards, the density of these problem instances is comparable and representative of problem density in practice. In other words, the number of simultaneously moving railcars at any given time in our instances is less than 20% of the slots, and attempts to simultaneously move a higher percentage of trains in practice would be unrealistic, either due to a limited number of available locomotives, because of the high potential for gridlock, or because doing so would require a greater number of cars per train than is typically found in practice.

Table 1 Performance Comparison and Computational Results

Exact Solution Compared to Heuristic Solution	# of Samples	% of Samples	Avg. Exact Model Solution Time (sec)	Avg. Heuristic Model Solution Time (sec)
Exact = Heuristic	53	42%	505	35
Exact > Heuristic	17	14%	3600	36
Exact < Heuristic	55	44%	792	35
Total	125	100%	1052	35

Recall that the optimization model requires dividing the railyard network into discrete slots, and its solution provides a period-by-period location of each railcar, while the GE heuristic approach transforms the railyard into a shortest route network. Both approaches result in collision-free routes that account for the complexity imposed by switches and double-back moves while allowing simultaneous moves. Comparative results for 125 randomly generated problem instances of comparably small size (relative to practice, with less than 140 track slots) are reported in Table 1. Note that the table only reports comparative results for problem instances such that we were able to determine a feasible solution within an hour using the exact model. (As the table shows, the exact approach was not able to determine an optimal solution in 17 of the 125 instances, which shows a solution time of 3600 s.)

Figure 11 shows a solution comparison between the exact approach and the GE heuristic approach. Observe that when the GE heuristic outperforms the exact approach (because the exact approach did not complete within 1 h, and the results reported correspond to the best solution found within an hour), the blue line lies above the purple line. These points are associated with the 17 instances where the exact approach was not able to find the optimal solution. Other than these 17 instances, the chart provides a representation of the proximity of the solutions achieved by the two approaches.

Figure 12 displays a performance comparison for the exact and heuristic approaches in terms of absolute gap and relative gap, for the 108 out of 125 problems that were solved optimally using the exact approach. Note that a difference of

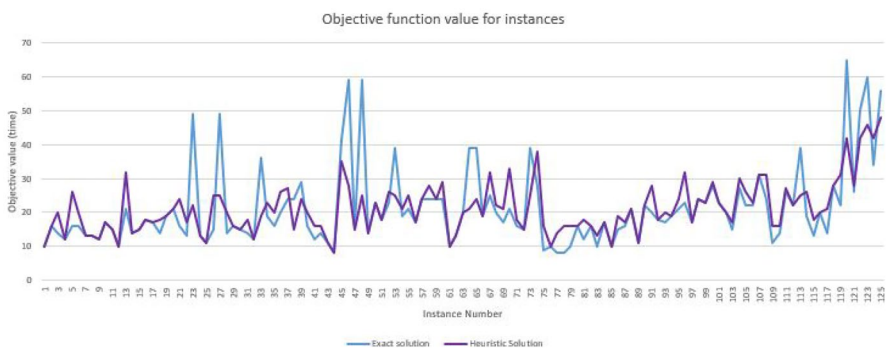


Fig. 11 Objective function values gained by exact and heuristic approaches

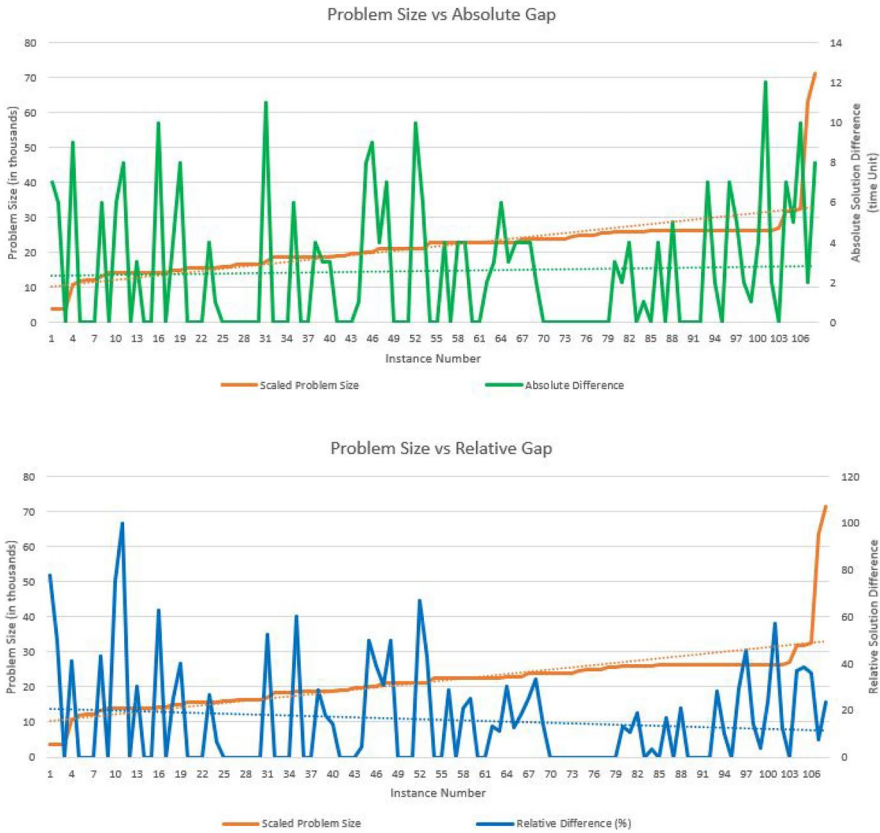


Fig. 12 Relative performance of the exact and heuristic approach

zero is, of course, desirable and implies that the heuristic found an optimal solution (which occurred in 42% of the cases). To illustrate problem instance sizes within the graphs, we defined a measure of problem size by multiplying the number of railcars, number of track slots, and the time horizon length.

The average absolute and relative gaps across these 108 instances were 2.6 and 15.9%, respectively. Confining these measures to the 108 problem instances in which the exact approach finds an optimal solution provides a biased view of these performance measures that favors the exact approach, which failed to find an optimal solution within an hour for 14% of the problem instances. We note that although the relative gap is as high as 100% in some instances, the absolute difference is generally flat as the problem size increases, while the percentage gap decreases (as shown by the dashed trend lines). That is, for very small problem instances, a one unit deviation from optimality may correspond to a high percentage deviation. This favors the use of the exact approach for small problem sizes, while applying the heuristic approach to problems of larger size, both with respect to relative performance

(optimality gap) and solution time (the heuristic solution time was around a half minute, regardless of problem instance size in these cases).

It is important to note that 35% of the instances are solved within 5 min by exact approach. However, Table 1 shows that the heuristic algorithm's run-time outperforms the exact approach regardless of problem size. Because of this, although the exact approach generates an optimal solution within 5 min for a relatively small percentage of cases, the heuristic approach is scalable to practical size instances due to its polynomial-time complexity, while the exact approach is often unreliable, even for this set of problem instances, which are smaller than found in practice. In practice, for example, decision makers often solve a multiple-train routing problem every hour or so, which requires a faster solution time and greater dependability than the exact approach is able to provide.

7 Discussion and Conclusion

This work considered the complex problem of simultaneously moving trains from a set of initial locations to destination locations in a railyard network. Our modeling and solution approaches account for track section and switch node capacity limits, allowing at most one train on a track segment at a time. We provided an exact problem formulation via a large-scale 0-1 optimization model and showed that the associated problem falls into the class of \mathcal{NP} -hard optimization problems. We also proposed a K -shortest-paths-based heuristic approach to minimize the total time required for trains to reach their destinations while accounting for collision avoidance. The solution methods proposed for this problem can serve as valuable input to a simulator or a decision support system for hub managers who wish to automate railcar repositioning decisions or evaluate different routing scenarios and options.

Our implementation did not consider the problem of attaching locomotives to trains. That is, the implementation of the exact approach assumed that every train requiring a move has a railcar attached, i.e., a locomotive is considered as another railcar within a set of connected railcars. As we discussed in some detail in Sect. 5.2, including locomotive moves within the mathematical model results in an additional layer of complexity, requiring a nontrivial additional number of variables and constraints. However, a multi-phase problem solution approach similar to the one we described in Sect. 5.2 allows for the reapplication of the heuristic in order to account for required locomotive moves. This requires a semi-automated multi-step heuristic procedure, where an initial problem is solved that moves locomotives from a set of initial positions to trains. In the second step, each train with an attached locomotive can now be moved by applying our heuristic algorithm. This same procedure can be repeated for as many locomotive-to-train assignments as is needed to move all trains on the network. Future work may include the development of improved methods for solving the locomotive attachment problem, along with methods to improve the mixed-integer programming formulation, and an in-depth analysis of the computational tradeoffs with respect to additional GRASP heuristic search time.

Data Availability The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of Interest The authors declare no competing interests.

References

1. Lusby RM, Larsen J, Ehrgott M, Ryan D (2011) Railway track allocation: Models and methods. *OR Spectrum* 33(4):843–883
2. Higgins A, Kozan E, Ferreira L (1996) Optimal scheduling of trains on a single line track. *Transportation Research Part B: Methodological* 30(2):147–161
3. Carey M, Lockwood D (1995) A model, algorithms and strategy for train pathing. *J Oper Res Soc* 46(8):988–1005
4. Liu SQ, Kozan E (2011) Scheduling trains with priorities: A no-wait blocking parallel-machine job-shop scheduling model. *Transp Sci* 45(2):175–198
5. Lange J, Werner F (2018) Approaches to modelling train scheduling problems as job-shop problems with blocking constraints. *J Sched* 21(2):191–207
6. Caprara A, Fischetti M, Toth P (2002) Modeling and solving the train timetabling problem. *Oper Res* 50(5):851–861
7. Cacchiani V, Caprara A, Toth P (2008) A column generation approach to train timetabling on a corridor. *4OR* 6(2):125–142
8. Zwaneveld P, Kroon L, Romeijn H, Salomon M (1996) Routing trains through railway stations: Model formulation and algorithms. *Transp Sci* 30(3):181–194
9. Fuchsberger M, Lüthi PDHJ (2007) Solving the train scheduling problem in a main station area via a resource constrained space-time integer multi-commodity flow. Institute for Operations Research ETH Zurich
10. Mascis A, Pacciarelli D (2002) Job-shop scheduling with blocking and no-wait constraints. *Eur J Oper Res* 143(3):498–517
11. D’ariano A, Pacciarelli D, Pranzo M, (2007) A branch-and-bound algorithm for scheduling trains in a railway network. *Eur J Oper Res* 183(2):643–657
12. Rodriguez J (2007) A constraint programming model for real-time train scheduling at junctions. *Transp Res B Methodol* 41(2):231–245
13. Cornelsen S, Di Stefano G (2007) Track assignment. *J Discrete Algorithms* 5(2):250–261
14. Pellegrini P, Marlière G, Rodriguez J (2014) Optimal train routing and scheduling for managing traffic perturbations in complex junctions. *Transp Res B Methodol* 59:58–80
15. Samá M, Pellegrini P, d’Ariano A, Rodriguez J, Pacciarelli D (2017) The potential of the routing selection problem in real-time railway traffic management. In: 7th International Conference on Railway Operations Modelling and Analysis-Rail Lille, p 19p
16. Samá M, Pellegrini P, D’Ariano A, Rodriguez J, Pacciarelli D (2016) Ant colony optimization for the real-time train routing selection problem. *Transp Res B Methodol* 85:89–108
17. Fischer F (2015) Ordering constraints in time expanded networks for train timetabling problems. In: 15th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2015), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
18. Ahuja R, Jha K, Liu J (2007) Solving real-life railroad blocking problems. *Interfaces* 37(5):404–419
19. Schlechte T (2012) Railway track allocation: Models and algorithms. PhD thesis, Technischen Universität Berlin
20. Borndörfer R, Erol B, Graffagnino T, Schlechte T, Swarat E (2014) Optimizing the simplon railway corridor. *Ann Oper Res* 218(1):93–106
21. Borndörfer R, Klug T, Schlechte T, Fügenschuh A, Schang T, Schülldorf H (2016) The freight train routing problem for congested railway networks with mixed traffic. *Transp Sci* 50(2):408–423
22. Schlechte T, Borndörfer R, Erol B, Graffagnino T, Swarat E (2011) Micro-macro transformation of railway networks. *J Rail Transp Plann Manage* 1(1):38–48

23. Borndörfer R, Klug T, Lamorgese L, Mannino C, Reuther M, Schlechte T (2017) Recent success stories on integrated optimization of railway systems. *Transportation Research Part C: Emerging Technologies* 74:196–211
24. Bohlin M, Flier H, Maue J, Mihalák M (2011) Track allocation in freight-train classification with mixed tracks. In: 11th workshop on algorithmic approaches for transportation modelling, optimization, and systems, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
25. Bohlin M, Hansmann R, Zimmermann UT (2018) Optimization of railway freight shunting. In: *Handbook of Optimization in the Railway Industry*, Springer, pp 181–212
26. Minbashi N, Bohlin M, Palmqvist CW, Kordnejad B (2021) The application of tree-based algorithms on classifying shunting yard departure status. *J Adv Transp* 2021
27. Aliakbari M, Geunes J, Sullivan K (2021) The single train shortest path problem in a railyard. *Optim Lett* 15:2577–2595
28. Peis B, Skutella M, Wiese A (2009) Packet routing: Complexity and algorithms. *International Workshop on Approximation and Online Algorithms* pp 217–228 Springer, Berlin, Heidelberg
29. Yen JY (1971) Finding the k shortest loopless paths in a network. *Manage Sci* 17(11):712–716
30. Kanet JJ (1986) Tactically delayed versus non-delay scheduling: An experimental investigation. *Eur J Oper Res* 24(1):99–105
31. Resende M, Ribeiro C (2014) GRASP: Greedy Randomized Adaptive Search Procedures. *Search Methodologies - Introductory Tutorials in Optimization and Decision Support Systems* (2nd edition):287–312

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.