# OPTIMIZATION MODELS FOR TRAIN ROUTING IN RAILYARD NETWORKS

A Dissertation

by

MINA ALIAKBARI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2021

Major Subject: Industrial & Systems Engineering

## ABSTRACT

In this document, I will explain three research topics that comprise my dissertation. The three topics presented in three separate chapters correspond to problems that arise in the railway industry, in the areas of freight train repositioning operations, routing, and scheduling.

The first research subject deals with a single train routing problem within a railyard network. A railyard, sometimes called a rail hub, is a large set of connected rail tracks that function as a station for freight trains. These facilities are where the main operations of assembly and disassembly of railcars occur in order to create outbound trains, as well as load and unload railcars. A fundamental problem is to determine the shortest route for a train given its desired initial and final locations on the railyard network. This network has geometrical limitations imposed by switch nodes, as well as constraints imposed by track and train lengths. In other words, depending on train length and the geometrical structure of the railyard network, we wish to find the shortest path for a predetermined start and end location. To address this problem, we propose two related approaches that transform the railyard network into an expanded graph. A shortest path algorithm can be applied on the expanded graph to find the shortest route for a single train from an initial to a final location. Our polynomial-time algorithms are able to deal with the unique geometric structure of the railyard networks. They also account for locomotive orientation (i.e., whether the locomotive is pulling or pushing the train during each move). In addition, our approach does not restrict the route to start from a specific node, rather it allows a train to span multiple nodes. This problem and our proposed algorithms provide a foundation for the subsequent research topic.

The second research topic considers the simultaneous movement of multiple trains over the railyard network. As discussed, the main operation within railyards is the assembly and disassembly of railcars to form the outbound trains. Such operations can sometimes be performed simultaneously, especially if there are multiple trains being prepared for departure in the upcoming hours. In this case, multiple movements of the railcars over different tracks of the railyard network are necessary. If these repositionings are done simultaneously, this will save time compared to performing single train routing one at a time. In presenting this problem, we propose an integer programming model and a heuristic approach for solving a routing problem involving multiple trains. The key is to generate collision-free routes while accounting for the unique structure of railyard networks. Our constructive heuristic approach works based on the assumption of having a ranked list of trains that are going to be repositioned. We relax the requirement of having to strictly prioritize trains by using a GRASP metaheuristic over the constructive heuristic we propose. Our integer programming model generates an optimal solution for many smaller-size instances of the problem. However, for practical size problems, the heuristic approach is favorable because of its fast solution time.

The third topic addresses train departure scheduling over a finite time horizon. In this problem, a set of containers is given with their predetermined destinations. We consider these containers as units that must be loaded on trains to get shipped to their destinations. Therefore, units assigned to the same train should have a common destination. However, the departure time of the train depends on the units assigned to the train because each unit has a time interval within which it must be shipped. The unit's shipping interval is predetermined along with its destination. The interval of each unit is different from the other units, although they may overlap. This enables a train to load a batch of units that have the same destination and share a specific departure time that lies within each of their shipping intervals. Therefore, scheduling departures of outbound trains requires determining train departure times and destinations, along with the assignment of individual units to trains. We call this the Train Assignment Planning problem. The objective is to build an outbound train schedule that satisfies restrictions imposed by units' destinations and shipping intervals. Different performance measures such as the degree of on-time performance, as well as schedule workload balance are investigated. We provide mixed integer programming models and constructive heuristic approaches for distinct objective functions. We also propose a Lagrangian Relaxation approach for maximizing on-time performance and we use it to provide a solution upper bound and a means to evaluate the quality of the approaches. Comparing the solution time for the integer programming approach and the heuristic approach, we show that the heuristic performs much faster and provides a solution within seconds without sacrificing solution quality.

The three topics briefly explained above are related as they all provide useful tools for railyard facility systems, whether it is necessary to find the shortest route, investigate the possibility of simultaneous repositioning of trains, or scheduling and assigning shipment units to outbound trains. In this document that I present as my dissertation, I present these three topics in detail.

DEDICATION

To my family.

## ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Joseph Geunes, who guided me through my Ph.D. career. I feel lucky for having the chance to be his student. He was not only a great instructor for my academic success but also an excellent mentor. I also would like to thank Dr. Amirali Ghahari and Dr. Mike Prince for their superb help and for assisting me in understanding the real-world problems related to my research. I take this opportunity to appreciate all faculty and staff in the department of Industrial and Systems Engineering at Texas A&M University. My academic development and personal growth are all due to being part of this community. I should not underestimate the role of my friends who made this experience more pleasant. Last but not least, I would like to say thank you to my family for all their support, hope, and love.

# CONTRIBUTORS AND FUNDING SOURCES

## Contributors

This work was supported by a thesis (or) dissertation committee consisting of Professors Geunes, Butenko, and Garcia of the Department of Industrial & Systems Engineering and Professor Shetty of the Department of Information and Operations Management.

All data used in computational testing were generated randomly by the student independently.

All other work conducted for the thesis (or) dissertation was completed by the student independently.

## Funding Sources

Graduate study was supported by an assistantship from Texas A&M University and a research grant from BNSF Railway.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# 1. THE SINGLE TRAIN SHORTEST ROUTE PROBLEM IN A RAILYARD NETWORK

## 1.1 Introduction

A typical freight rail network consists of a set of rail hubs, typically in large metropolitan areas, connected to one another via main line tracks that may span hundreds of miles between hubs. A hub facility acts as an origin point, destination point, and sorting facility for various freight rail cars. Routine daily operations at a rail hub include receiving trucks that drop off intermodal freight bound for other cities via rail, offloading rail cars from arriving trains for pickup by trucks for regional intermodal distribution, disassembling rail cars from arriving trains from other hubs, and assembling rail cars into outbound trains destined for other hubs.

The combined activities of disassembling incoming trains and assembling outbound trains require frequently moving and repositioning subsets of connected cars within the railyard using a locomotive to either push or pull the cars. We will refer to any set of connected cars that requires movement via a locomotive generically as a train. Efficient disassembly and reassembly of cars requires moving trains within the railyard network as quickly as possible. Thus, given a subset of cars that requires repositioning in the railyard network, we wish to find the shortest route from an origin location to a destination location in order to minimize total repositioning time. A fast solution procedure for these problems is necessary since repositioning cars occurs frequently.

The associated shortest route problems contain features that make them fundamentally different from classical shortest path problems on a network. These differences arise from the structure of the railyard network, the length and orientation of a train, and restrictions on movements trains face in the network. For example, a train approaching a node in the network via a given edge cannot necessarily proceed directly onto any other edge connected to the node (where we define a node in the network as a point at which three or more sections of track meet). In particular, if the section of track from which the train enters the node forms an acute angle with another section of track connected to the node, then the train cannot proceed directly onto the latter edge from the former.

It may be possible, however, for the train to proceed onto a section of track connected to the node that forms an obtuse angle with the train's direction of entry, travel entirely past the node, and then to reverse direction onto the edge forming an acute angle with the train's original direction of entry to the node. Note that this is only possible if a sufficient amount of track exists to permit the train to fully pass the node and then reverse direction.

In addition to the complications resulting from acute angles in the railyard network, we also encounter the possibility that the length of the train may span multiple nodes in the network at the same time due its length. This requires a different characterization of the train's position in the network than is typically used in classical shortest path problems. Moreover, a train's original and final desired orientation in the network may differ. We define a train's orientation based on the end of the train at which the locomotive is attached, as well as whether the locomotive is pushing or pulling the train. Thus, if we denote the end of the train with a locomotive as the positive (+) end (and the other end as the negative ($-$) end), when the locomotive is pulling the rail cars, the (+) end is leading the direction of motion. This complication sometimes arises as a result of which end of the train a locomotive can more easily access to initiate movement, as well as the ease with which the locomotive can be disconnected from the train after moving to the train's final layout. If, for example, the train's final layout falls on a dead-end track segment, then it is often necessary to ensure that the locomotive pushes the train onto the dead-end segment (with the ($-$) end leading) so that it can easily disconnect from the train and move to another track segment without being blocked.

Motivated by the need to frequently reposition trains in the railyard network, we consider the shortest route problem for a train in a railyard network, given a train's origin and destination positions in the network, as well as its initial and final orientation. We provide an intuitive approach for solving this problem, which results in an algorithm that is relatively easy to implement and to describe. The worst-case performance of this algorithmic approach is no worse than Dijkstra's classical shortest path solution method of $\mathcal{O}(n^2)$ for a network containing $n$ nodes (Dijkstra [2]). This bound is typically quite loose for railyard networks, which are often sparse, containing a number of edges that is much smaller than $n^2$. We note that Enayati Ahangar, Sullivan, Spanton, and Wang [1] recently developed an $\mathcal{O}(n^2)$ algorithm for solving problems in this class. While the worst-case bound of our approach is the same when expressed as a function of the number of nodes $n$, we demonstrate that our shortest route approach requires fewer total edges, which reduces the computational requirements for any instance. Our approach also permits a relatively straightforward and compact description, which facilitates ease of implementation while accounting for the fact that train position can span multiple nodes.

The remainder of this paper is organized as follows. The following section discusses related literature in the area of train routing. Section 1.3 then formalizes the problem definition, after which Section 1.4 discusses two algorithmic approaches for its solution. Section 1.5 then characterizes the complexity associated with our solution approaches, while Section 1.6 provides concluding remarks.

## 1.2 Related Literature

A large volume of literature exists dealing with the application of operations research and analytics techniques to railroad freight scheduling and routing problems. Cordeau, Toth, and Vigo [3] provide an excellent overview and classification of work in this area. The majority of this work focuses on the flow of rail traffic between and through the various railyards comprising a nationwide or international rail network. A railcar entering the system at a railyard, or terminal, is typically classified into a *block*, implying that it is grouped with a set of cars that are assigned a common destination. A block's designated destination may not correspond to the final destination of each car in the block. If the block's destination does not correspond to a car's destination, then the car must be reclassified at the block destination for further transport. A nonlinear mixed-integer optimization model and solution method for determining a yard's blocking policy is provided in Bodin, Golden, Schuster, and Romig [4]. This multicommodity flow model specifies a final-destination-dependent classification block for any car at a given railyard. The combined problem of scheduling train departures at each yard and assigning blocks to scheduled trains is considered in Jha, Ahuja, and Şahin [5]. This work models the multicommodity flow problem on an expanded time-space network and applies integer programming and heuristic techniques to obtain high-quality solutions. While this prior stream of work on blocking problems emphasizes larger tactical planning problems of moving freight from origin to destination via multiple railyards, we focus on operations within a railyard.

A significant body of prior work considers models that measure railyard operations performance measures, intended to characterize throughput times of cars or blocks through a railyard. Petersen [6] provides a thorough description of typical railyard activities and develops a queuing model to predict train throughput times. Turnquist and Daskin [7] refined this queuing analysis to account for individual car-level processing times and considered the impacts of different dispatching approaches. Daganzo, Dowling, and Hall [8] further considered the way in which different block sorting strategies influence average yard performance measures. A common approach in practice for analyzing the way in which different policies and strategies affect railyard throughput performance is to create a simulation model that accounts for random train arrivals, block sorting strategies, and departing train dispatch rules (Cordeau, Toth, and Vigo [3]). Our focus in this paper lies in addressing a problem that significantly contributes to railyard throughput performance, i.e., the most efficient movement of a connected set of cars through the railyard from an origin to a destination.

Several past works have considered the problem of simultaneously routing multiple trains through a common set of tracks and platforms such that no collisions occur and some measure of service performance is achieved (e.g., each passenger train has an arrival time and a desired departure time, and the objective is to ensure that each train departs before its desired departure time; see, e.g., Zwaneveld, Kroon, Romeijn, and Salomon [9]). Sama, Pellegrini, D'Ariano, Rodriguez, and Pacciarelli [10] provide a mixed-integer optimization formulation and a metaheuristic solution approach for real-time selection of routes when different trains may desire the same track at the same time. They use an objective of minimizing targeted departure delays. For each train, their approach determines a subset of routes selected from among the existing routing alternatives available for each train. Lange and Werner [11] take a different approach by modeling train path assignments as a job-shop problem. Each track or track section is equivalent to a machine, while each train corresponds to a job; as a result, each route corresponds to an ordered set of operations performed on a set of machines. Conflicts are therefore eliminated by assigning one job (train) at a time to a machine (track).

Despite the high volume of literature on block-to-train assignment and scheduling in rail freight, as well as in the area of routing multiple passenger trains through common tracks to adhere to a schedule, relatively little work exists that explicitly considers routing decisions within a railyard while accounting for the unique network topology and train travel restrictions inherent in a freight railyard network. Two relatively recent notable exceptions exist. Riezebos and Wezel [12] consider the problem of determining the $k$ shortest paths for a train in a shunt yard with an objective of minimizing the number of crossovers and direction changes associated with each path. Their model treats each section of track as a node in the network with an associated direction of travel and does not account for the fact that a freight train may span multiple track sections at any given time (the application context on which they focus deals with overnight repositioning of self-propelled coach cars). The arc costs in the associated network account for distances between track segments as well as crossover and direction change penalties. The most closely related work to ours is that of Enayati Ahangar, et al. [1], who consider the same shortest freight-train route problem in a railyard network. We rely on a preprocessing stage they developed as our initial step, which determines, for a given train length and railyard network topology, which acute-angle turns in the network are feasible. We then propose a shortest route solution approach based on the solution of an equivalent shortest path problem, which differs from their approach. While the worst-case complexity of our approach is the same as theirs when expressed as a function of the number of nodes in the network, our approach requires fewer edges, which determines the number of computations required using common shortest path methods, such as Dijkstra's algorithm [2]. Our proposed approach arguably lends itself to a less complex description, which facilitates easier implementation. We next define this problem and describe our proposed solution approach.

## 1.3 Problem Definition

This section defines the structure of the railyard networks we consider, as well as the constraints this network imposes on train travel. A railyard network consists of a set of tracks that are connected to one or more main tracks via lead tracks (where the main tracks lead to other railyards). Track length can span over ten thousand feet, and numerous sections of tracks are connected to one another via switch points. A switch point corresponds to a point at which three or more track sections meet, enabling an approaching train to depart the switch point via multiple directions. Figure 1.1 illustrates a typical railyard layout.

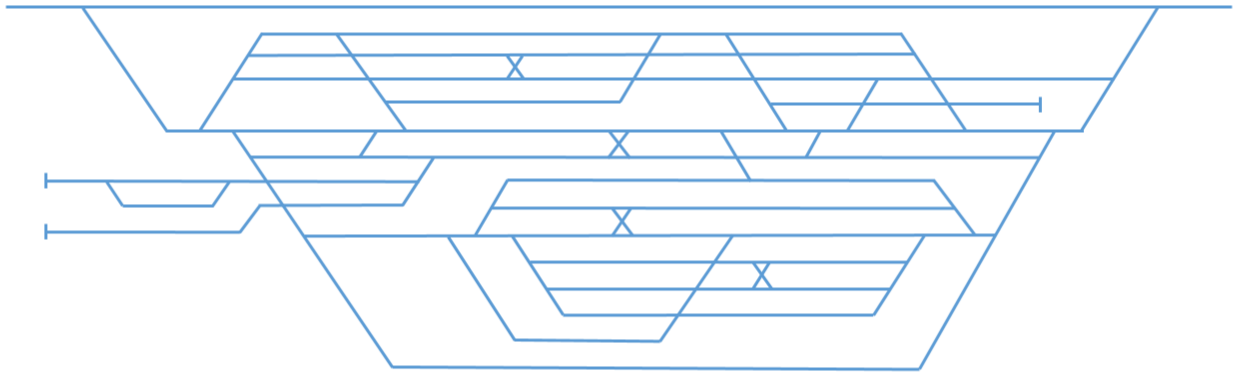


Figure 1.1: Illustration of a sample railyard.

A railyard switch point permits a train to travel onto two different tracks. Hence, a typical switch point involves three track segments converging at a point such that the angle formed by one of the pairs of track segments is acute, while the angles formed by the other two pairs are obtuse. A crossover is essentially a pair of switches that facilitate connections between two parallel tracks. A double crossover corresponds to a pair of crossovers which permit travel between two parallel tracks regardess of travel direction (crossovers are depicted with an $\times$ in Figure 1.1.)

Physical constraints restrict a train to only depart a switch point via a track segment that forms an obtuse angle with the track segment on which it approaches the switch. In our network representation of the railyard, each switch point defines a node in the network. We assume without loss of generality that exactly three track segments meet at any node, where a track segment defines an edge in the network. If, for example, four or more edges meet at a node, we can create copies of the node with zero distance between the node and its copies, where the original node and each copy are incident to three edges (see Figure 1.2). After making any required adjustments to ensure that exactly three edges meet at any node, each such node may be viewed as a connection point between two in-line track segments and another segment that forms an acute angle with this line; equivalently, each node creates two supplementary angles, one of which is acute and the other obtuse.

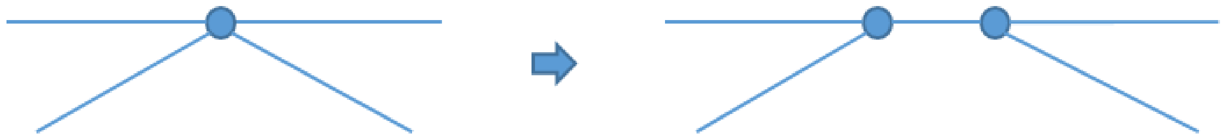


Figure 1.2: Creation of degree-three nodes.

We define a train as a car or set of cars attached to one another on the network, which must move as a single entity on the network. Given a rail network on which a train lies, we can describe the train's layout on the network with a series of nodes that form a sub-path in the network (see Figure 1.3), which characterize the train's position in the network.

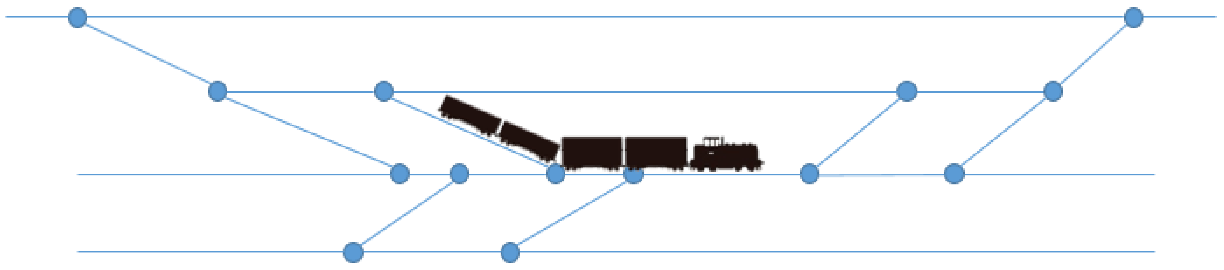


Figure 1.3: A train's layout at any given time may span multiple nodes.

The goal of the shortest train route problem is to find the shortest distance (or time) required to move a train in the network from some initial layout to some final layout (Figure 1.4 (a) and (b) illustrate two sample networks with initial and final locations of a train indicated).

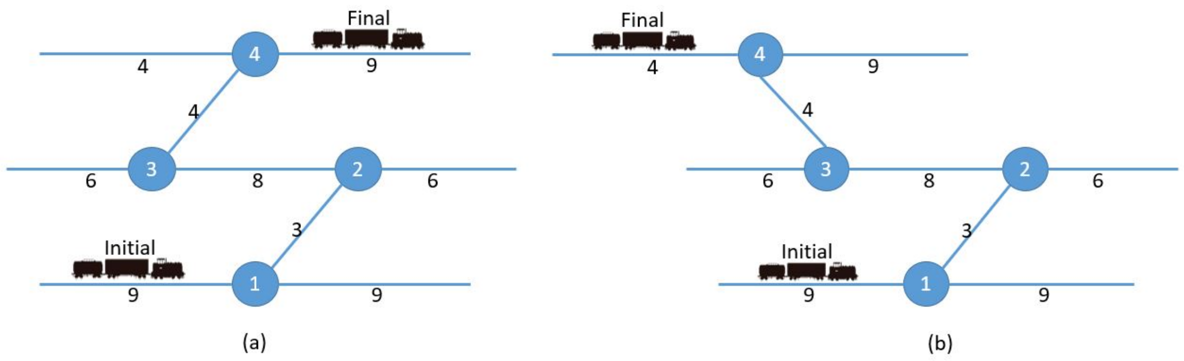


Figure 1.4: Example problem instance with track lengths.

Recall that a train approaching a switch node may continue travelling without delay onto the track segment with which its current track segment forms an obtuse angle. Whether or not it can travel back onto the other acute-angle leg depends on the train's length and whether a sufficient amount of available track space exists to allow the entire train to pass the switch node and then reverse direction. If we let $L$ denote the train's length, then the railyard network must be able to accommodate a length of $L$ on the other side of the switch node via the obtuse angle leg in order for the train to entirely pass the node, reverse direction, and then proceed onto the other acute-angle leg. We say that a switch node is *backable* by a train of length $L$ if the train is able to traverse its acute angle, and we refer to such a move as a *double-back*. Figure 1.5 illustrates a sequence of moves in which a train performs a double-back move on an acute angle.

In [1], the authors define acute-angle-free cycles in a network. An acute-angle-free (AAF) cycle is the one that starts at a node and returns back to it without traversing an acute angle. Figure 1.6 demonstrates a sample AAF cycle. For example, if a train travels from node 6 to node 5, it can then visit nodes 1, 2, and 3, in that order, and return to node 5 without having traversed

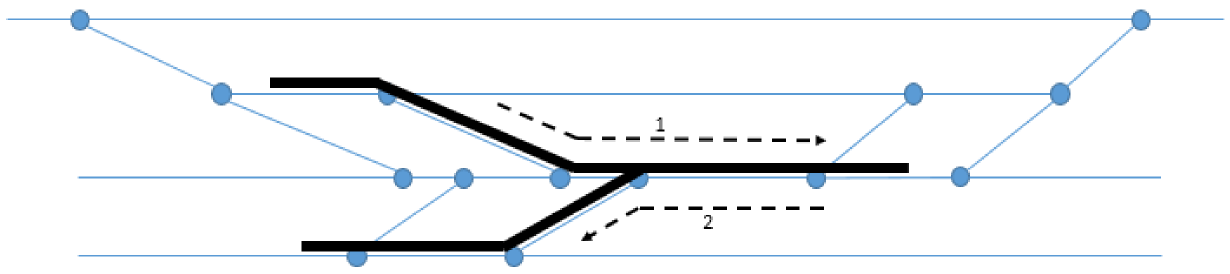


Figure 1.5: A double-back move at a switch requiring a direction change.

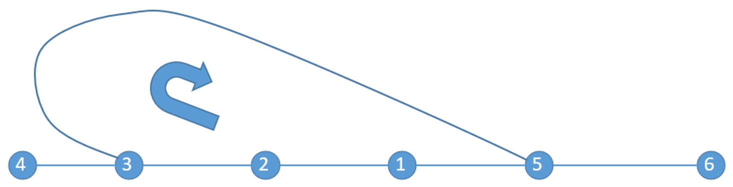


Figure 1.6: An example acute-angle-free cycle is 5-1-2-3-5.

any acute angles. (Observe that the ability to traverse an AAF cycle without any acute angle turns may depend on the train's direction of travel, i.e., the same cycle in Figure 1.6 cannot be traversed without an acute angle turn when visiting nodes in the sequence 1-5-3-2-1.) For networks such that all AAF cycles have length of at least $L$ (the train's length), Enyaty Ahangar et al. [1] provide a polynomial-time preprocessing routine that determines whether each node is backable (for networks that permit AAF cycles of any nonnegative length, they show that this preprocessing problem is $\mathcal{NP}$-Hard). We confine our analysis to networks that do not possess such AAF cycles of length less than $L$, assuming therefore that we can use the algorithm in [1] to determine whether each angle in the network is backable in polynomial time for a given train of length $L$. The output of this preprocessing step, i.e., whether or not each node is backable, will serve as an input to our solution approach discussed in Section 1.4.

We will first present each of our solution approaches without regard for the train's orientation, i.e., assuming only the initial and final locations of the train are specified, independent of the location of the (+) and (−) ends within the layout. Later, we present modifications needed to account for a desired orientation of the train within its final layout position. Using this approach, a train's desired final layout on the network can be characterized by those nodes on top of which the train will lie, plus two outer nodes that bound the outer-most cars of the train (these "outer nodes" of a layout correspond to the first node the lead car of the train will reach by traveling in either direction). Suppose that in the final layout, the train covers a set of $\eta$ nodes. For example, $\eta$ equals two in Figure 1.3. This creates at most $\eta + 2$ candidate "entry nodes" (the $\eta$ nodes spanned by the train plus the two outer nodes) via which the train may reach its final layout from some node not among these final layout nodes. Given a candidate entry node and whether or not it is backable, we can efficiently determine whether the train's entry to its final layout via this node is feasible.

Our solution approaches will require differentiating between the different edges incident to a given node $i$, and we adopt the convention used in Enayati Ahangar, et al. [1] by referring to the pair of edges that form an acute angle at node $i$ as edges $2(i)$ and $3(i)$, while the remaining edge corresponds to edge $1(i)$ (see Figure 1.7).

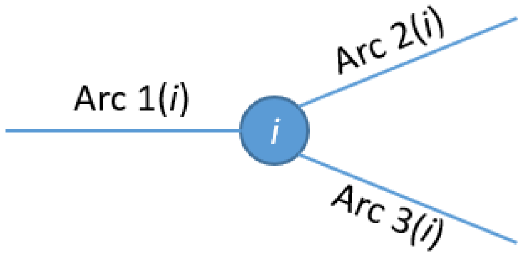


Figure 1.7: Arc naming convention.

Let $\mathcal{F}$ denote a set of feasible entry nodes. We will refer to an entry node $i$ as an $O$ (one) entry node ($OEN$) if it is in $\mathcal{F}$ and the train must achieve its final layout by entering the node via either $2(i)$ and $3(i)$ from some node not in $\mathcal{F}$ and subsequently exiting the node via edge $1(i)$; otherwise the entry node is a $T$ (two/three) entry node ($TEN$).

Similarly to the train's final layout, we assume without loss of generality that the train's initial layout is defined by a set of $m$ nodes spanned by the train, plus the two outer nodes corresponding to the first node reached by traveling in either direction. This creates at most $m+2$ candidate "start nodes" via which the train may begin its route. If the train must be moved into the start node $i$ by approaching via edge $2(i)$ or $3(i)$ and exiting via edge $1(i)$, then we will refer to that start node as an $O$ start node ($OSN$); otherwise, the start node is a $T$ start node ($TSN$).

Observe that any train route through the railyard network must consist of a set of alternating directed AAF walks, adjoined by backable nodes at which double-back moves are made for direction changes. As we move along a given direction, any time we reach a node, if we continue traversing an arc without double-backing (i.e., from $1(i)$ to one of $2(i)$ or $3(i)$ or vice versa), then we are continuing an AAF walk in the same direction; otherwise, we switch to the opposite direction when a double-back move is made.

The problem therefore is to find the shortest route between an initial and final layout that consists of a feasible sequence of alternating and adjoining directed AAF walks in the network.

## 1.4 Solution Approach

This section proposes two methods for obtaining the shortest route from an initial to final layout for a single train on a railyard network. We differentiate between a train's shortest route through the rail network, which may contain double-back moves, and the classical simple shortest path problem in a network, which is characterized by a sequence of nodes and associated edges that provide the shortest distance between an origin node and a destination node in the network. Although the all-shortest-paths approach described in Section 1.4.1 requires greater computational time in the worst case, it is very easy to describe, understand, and implement, and its presentation lays a foundation for describing the improved method presented in Section 1.4.2.

### 1.4.1 All-Shortest-Paths Approach

Section 1.4.1.1 first presents the approach without considering the train's orientation, i.e., when orientation is unimportant, while Section 1.4.1.2 later presents modifications needed to account for the train's orientation.

#### *1.4.1.1 Orientation-free approach*

This approach uses network information on how nodes are connected to one another in the network, as well as whether direction changes are required using double-back moves at any given node. A node $j$ is $O$-reachable ($OR$) from node $i$ if a path exists starting at $i$ and exiting via $1(i)$, which leads to node $j$ without making any acute angle turns, and without completely traversing any AAF cycles. Similarly, a node $j$ is $T$-reachable ($TR$) from $i$ if a simple path exists beginning at $i$ and exiting via $2(i)$ or $3(i)$, which leads to node $j$ without any acute angle turns, and without completely traversing any AAF cycles. Observe that it is possible for a node to be both $O$-reachable and $T$-reachable from a given node when AAF cycles exist in the network. Given any node, it is straightforward (using a recursive algorithm, such as depth-first or breadth-first search) to construct the subnetwork whose node set consists of all $OR$ nodes from the node, along with all edges with both ends incident to nodes in this node set; let $OR(i)$ denote this subnetwork. It is similarly straightforward, for any given node, to construct the subnetwork with node set consisting of all $TR$ nodes from the node, along with all edges with both ends incident to nodes in this node set; let $TR(i)$ denote this subnetwork. Examples of these subnetworks are illustrated in Figure 1.8, which depicts a set of subnetworks associated with the example network shown in Figure 1.4(b).

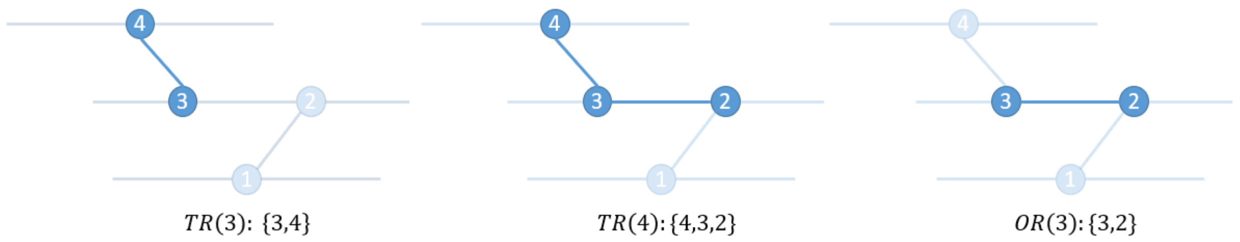


Figure 1.8: Illustration of O-reachable and T-reachable subnetworks.

Some additional discussion on the construction of subnetworks $OR(i)$ and $TR(i)$ is warranted when AAF cycles exist in the railyard network. Such cycles are not overly common in railyard networks, and typically exist for the purpose of temporary railcar storage. Despite this, we still account for the possibility of their presence in constructing a solution approach. Note that it is easy to show that if we are indifferent to the train's orientation on the network, an optimal shortest train route always exists that does not completely traverse an AAF loop. Because of this, when train orientation is not a concern, in the search for $O$-reachable or $T$-reachable nodes from some node $i$, if we reach some node $k$ for a second time, we can discard the second instance of the node from the subnetwork and discontinue further exploration of $OR(i)$ or $TR(i)$ from such a node. (When orientation is important, however, as we later discuss in Section 1.4.1.2, we cannot discard a second instance of a node encountered in the subnetwork and discontinue further exploration.)

Observe that if node $j$ is $O$-reachable from node $i$, then finding the shortest route of the train in $OR(i)$ from a layout in which the train exits node $i$ via $1(i)$ and arrives at node $j$ is equivalent to finding a simple shortest path from node $i$ to $j$ in the subnetwork $OR(i)$. Conversely, if node $j$ is $T$-reachable from $i$, then finding the shortest train route in $TR(i)$ from a layout in which the train starts at node $i$ and moves along $2(i)$ or $3(i)$ and ends at $j$ is equivalent to finding a simple shortest path from node $i$ to $j$ in the subnetwork $TR(i)$.

Suppose then that we first solve an all shortest paths problem without any double-back moves, i.e., for each node $i$ (for a total of $n$ nodes), we find the shortest path to each $OR$ node in $OR(i)$ and to each $TR$ node in $TR(i)$ and retain this information. That is, for each node in the original network, we make a list of all $OR$ and $TR$ nodes and their associated shortest path and distance. Using Dijkstra's algorithm, in the worst case, this requires $\mathcal{O}(n^3)$ computations, where $n$ is the number of nodes in the network; however, the total number of edges for a railyard network is typically much fewer than $n^2$, which suggests a better computational complexity bound if a special implementation of Dijkstra's algorithm is utilized.

We next define a new shortest path graph. This graph is built based on the idea that every feasible path in the railyard network corresponds to a sequence of alternating double-back moves from the train's initial location to its final destination. This new shortest path graph contains two layers; layer $T$ and layer $O$, with a copy of each node in the network in each layer. For each node

$i$ we refer to these copies as nodes $iT$ and $iO$. Traversing node $iT$ means the train exits node $i$ via edge $2(i)$ or $3(i)$, while traversing $iO$ implies that the train exits node $i$ via edge $1(i)$.

The following provides a more explicit definition of the five types of arcs needed to connect nodes in the resulting shortest path graph. Figure 1.9 illustrates the shortest path network's layers and arcs using the example shown in Figure 1.4(b).

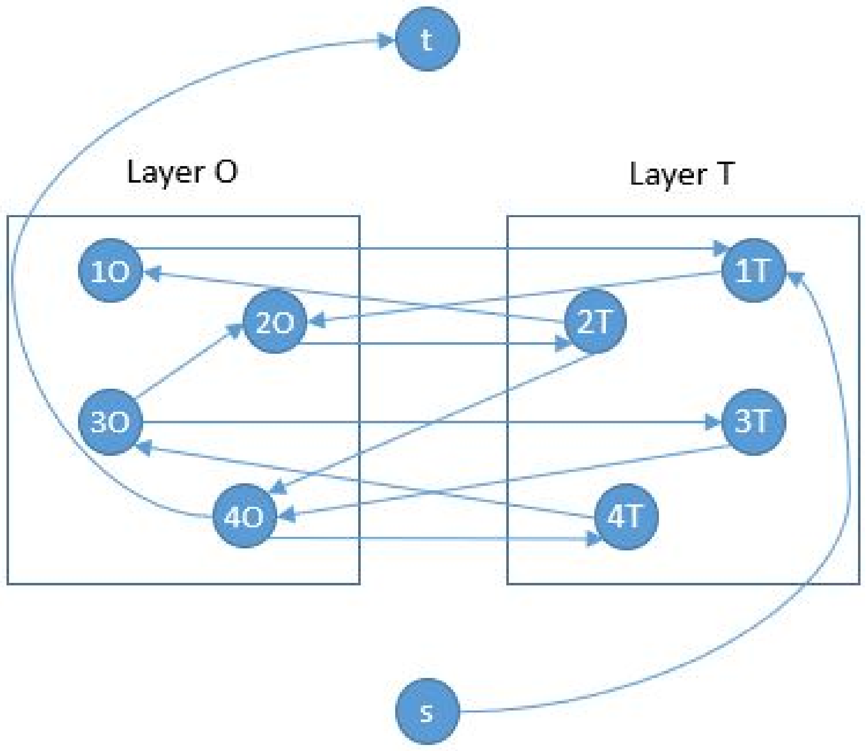


Figure 1.9: Illustration of O-T Layers shortest path graph.

1. **Source Arcs:** Recall that we begin with at most $m + 2$ feasible start nodes (consisting of the $m$ nodes covered by the initial layout plus the outer nodes bounding the train) associated with an initial path. Thus, we create a source node $s$ that connects to each valid start node, with an arc of length equal to the distance required to move the train from its initial position to start node. If start node $i$ is an $OSN$ (i.e., the train enters node $i$ on edge $2(i)$ or $3(i)$ and

exits via $1(i)$), then the source connects to the start node in layer $O$; if the start node is a $TSN$ (i.e., the train enters node $i$ on edge $1(i)$ and exits via $2(i)$ or $3(i)$), the source connects to the start node in layer $T$.

2. **Sink Arcs:** In a similar fashion to the source, we create a dummy sink node $t$, and for each of at most $\eta + 2$ feasible entry nodes, we create an arc from layer $O$ to the dummy sink node if the entry node $i$ is an $OEN$ (i.e., the train enters node $i$ on edge $2(i)$ or $3(i)$ and exits via $1(i)$); otherwise, the entry node connects to the sink node from layer $T$. The associated arc length is the distance required for the train to reach its final position from each entry node to the final train layout.

3. ***T* Exit Arcs:** Consider a node $i$ in layer $T$; if the node is backable, we create an arc from $iT$ to each node $j \neq i$ in layer $O$ that is in $TR(i)$ and has the potential to accommodate a subsequent double-back move, i.e., any node $j$ in $TR(i)$ such that the shortest path from node $i$ to $j$ in $TR(i)$ enters node $j$ via $2(j)$ or $3(j)$ (in other words, $j \in TR(i)$ and $i \in TR(j)$). The corresponding arc length equals the shortest path length in $TR(i)$. If an entry node $j$ to the final layout is in $TR(i)$ and cannot be used for a subsequent double-back move after leaving node $i$ on $2(i)$ or $3(i)$, then we connect node $iT$ to node $jT$ (providing a path to the sink node; observe that if node $j$ is an entry node that is not backable, then it cannot be an $OEN$ and must be a $TEN$). Doing this for all $i$ in layer $T$ completes the set of arcs leaving nodes in layer $T$. Note that an arc $(i, j)$ within layer $T$ is only created if node $j$ is a final entry node that is not backable in $TR(i)$ after leaving node $i$, and is reachable from node $i$ after exiting on $2(i)$ or $3(i)$.

4. ***O* Exit Arcs:** Consider a node $i$ in layer $O$; if node $i$ is an $OSN$, we create an outgoing arc from $iO$ to any node $j \neq i$ in layer $O$ that is in $OR(i)$ and can be used for a subsequent double-back move (i.e., such that we enter node $j$ via $2(j)$ or $3(j)$ in $OR(i)$ after exiting node $i$). The associated arc length equals the shortest path length in $OR(i)$. If an entry node $j$ to the final layout is in $OR(i)$ and cannot be used for subsequent double-back move after

leaving node $i$, we connect $iO$ to node $jT$ (providing access to the sink node).

5. **Double-Back Arcs:** If node $i$ is backable, we create an arc with length $L$ (the train length) from $iO$ to node $iT$. This corresponds to the added distance required along edge $1(i)$ for making a double-back move. Traversing these arcs corresponds to a double-back move. In other words, we first exit node $i$ via $1(i)$ and then immediately reverse and exit node $i$ via $2(i)$ or $3(i)$. Observe that if a node is backable, it is always possible to fully pass such a node and then reverse direction. Therefore, in the final shortest path graph, a given node can only have an arc leaving layer $O$ and connecting to the same node index in layer $T$ if it is indeed backable.

We call the resulting shortest path graph the $O$-$T$ Layered ($OTL$) graph. The shortest path from source node $s$ to sink node $t$ determines shortest train route over the original network considering the double-back moves. This solution consists of a start node, a sequence of nodes at which the train performs double-back moves, and a final entry node and, by construction, a path exists in the $OTL$ graph for each feasible path in the original train network. The length of each path in the $OTL$ network also correctly reflects the associated path length on the original network, plus the distance required for double-back moves.

The need to compute the shortest path in $TR(i)$ and $OR(i)$ for each node $i$ serves as the computational bottleneck. This requires solving $n$ shortest path problems, each of which requires $\mathcal{O}(e + n \log n)$ operations using a Fibonacci heap implementation of Dijkstra's algorithm ($\mathcal{O}(n^3)$ operations if we consider Dijkstra's algorithm). Thus, the total number of operations required for solving $n$ such problems is $\mathcal{O}(ne + n^2 \log n)$. The $OTL$ graph contains $\mathcal{O}(n)$ nodes and $\mathcal{O}(n^2)$ arcs and its solution, therefore, requires $\mathcal{O}(n^2 + n \log n) = \mathcal{O}(n^2)$ operations. As a result, the complexity of the overall solution approach is $\mathcal{O}(ne + n^2 \log n)$.

#### *1.4.1.2 Accounting for orientation*

We next consider how to extend the approach to account for the train's orientation. When considering orientation, the location of the locomotive relative to the train becomes important, i.e., whether the locomotive pulls or pushes the train into final location. This is particularly important in cases where the final layout lies on a dead-end of a track where it is undesirable to have the locomotive trapped behind the railcars. In Figure 1.10, for the train shown in the left figure to have reached its illustrated location and orientation, the locomotive may have either pulled the railcars through node 1, or it may have pushed them through node 2. However, for the train shown on the right, the locomotive would have needed to push the railcars through node 3 to prevent becoming trapped between the railcars and the track's dead-end. We next explain how we can manipulate the $OTL$ graph to ensure achieving not only a final train layout, but one with a desired orientation.

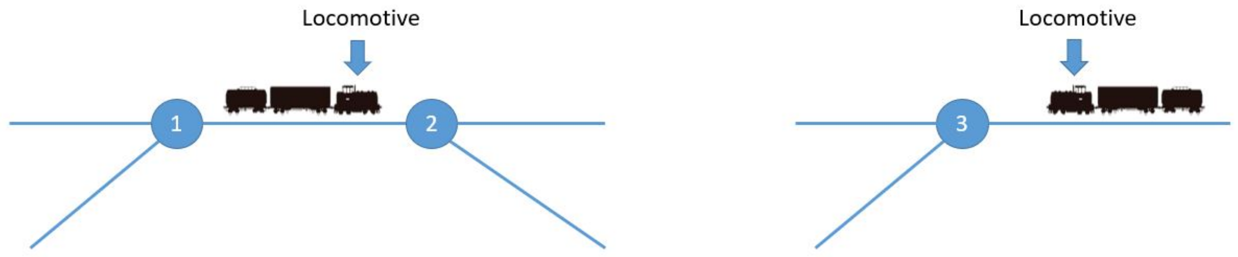


Figure 1.10: Orientation of the train and whether locomotive should pull or push.

Assume we have a locomotive on one end of the train designated as the (+) end, and suppose we create a duplicate of each node in layer $O$ and each node in layer $T$. That is, for each node $iO$ we create nodes $iO^+$ and $iO^-$, and for each node $iT$, we create nodes $iT^+$ and $iT^-$. Entry to $iO^+$ and $iT^+$ corresponds to node $i$ entry with the locomotive pulling, while entry to $iO^-$ and $iT^-$ corresponds to node $i$ entry with the locomotive pushing. A double-back move now corresponds to traversing arc ($iO^+$, $iT^-$) when the locomotive pulls into the double-back move node and pushes out of it, and to arc ($iO^-$, $iT^+$) when the locomotive pushes in the double-back move node and pulls out of it. All other arcs in the graph must go from a $(+)$ node to a $(+)$ node or from a $(-)$ node to a $(-)$ node. In addition, if node $i$ can subsequently be reached after departing node $i$ (by traversing an AAF cycle), such moves correspond to arcs of the form ($iO^+$, $iT^+$), ($iO^-$,

$iT^-$), ($iT^+$, $iO^+$), and ($iT^-$, $iO^-$). Traversing these arcs results in a change in direction relative to node $i$ while maintaining either a pushing or pulling orientation. For each node $i$ that is in $OR(i)$ or $TR(i)$, we determine the shortest path from the node back to itself in $OR(i)$ and $TR(i)$ to determine the length of these AAF loop arcs.

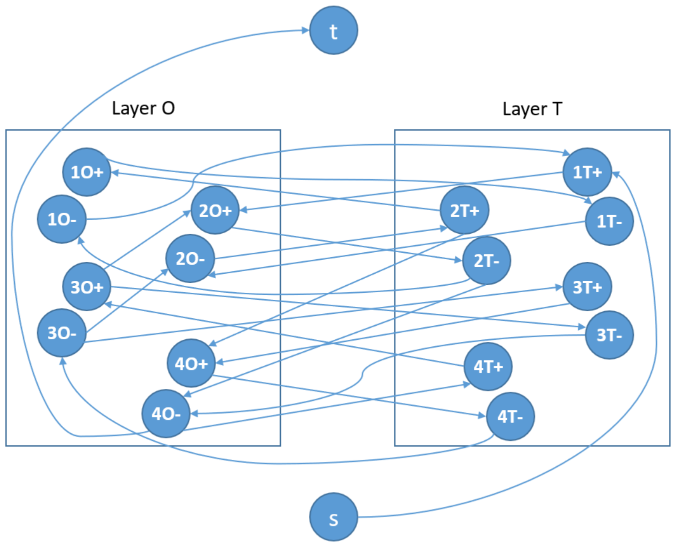


Figure 1.11: Illustration of O-T Layer graph considering train orientation.

For a final entry node $i$, we only connect to the sink node if the entry to the node has the proper orientation. For example, if node $i$ is a final entry node that must be approached using edge $1(i)$ and exited via $2(i)$ or $3(i)$ with the locomotive pulling, then only node $iT^+$ connects to the sink node. If node $i$ is a final entry node that must be approached using edge $2(i)$ or $3(i)$ and exited via $1(i)$ with the locomotive pulling, then only node $iO^+$ connects to the sink node. Similarly, if node $i$ is a final entry node that must be approached using edge $1(i)$ and exited via $2(i)$ or $3(i)$ with the locomotive pushing, then only node $iT^-$ connects to the sink node. And if node $i$ is a final entry node that must be approached using edge $2(i)$ or $3(i)$ and exited via $1(i)$ with the locomotive pushing, then only node $iO^-$ connects to the sink node.

Similarly, to connect the source node to start node(s), if leading with the $(+)$ end of the train (pulling) requires approaching start node $i$ using $1(i)$ ($2(i)$ or $3(i)$), then we connect the source node to $iT^+$ ($iO^+$) with the corresponding path length. If leading with the $(-)$ end of the train (pushing) requires approaching start node $i$ using $1(i)$ ($2(i)$ or $3(i)$), then we connect the source node to $iT^-$ ($iO^-$) with the corresponding path length. Figure 1.11 illustrates the orientation-based network's layers and arcs using the example shown in Figure 1.4(b). Compared to Figure 1.9, it is clear that the number of arcs increases to accommodate for train orientation.

This extension to permit accounting for the train's orientation results in an $OTL$ graph with twice the number of nodes and arcs, while maintaining the same worst-case complexity bound.

### 1.4.2 Improved Approach

We next describe an approach that eliminates the need for the all-shortest-paths solution and results in an improved worst-case complexity. We continue to assume without loss of generality that each node is incident to three edges, two of which form an acute angle. Given a node $i$, we refer to edges that form an acute angle as edges $2(i)$ and $3(i)$, and we call the remaining edge $1(i)$ (see Figure 1.7).

For any node traversed by the train, we can identify an edge of entry and an edge of departure. Observe that entry to node $i$ via edge $1(i)$ permits continuing along edge $2(i)$ or $3(i)$ without an acute-angle turn, while entry via either edge $2(i)$ or $3(i)$ permits continuing along edge $1(i)$ without an acute-angle turn. However, entry via $2(i)$ $(3(i))$ with a departure along edge $3(i)$ $(2(i))$ requires turning along the acute angle via a double-back move. Note that if edge $1(i)$ in Figure 1.7 does not connect to another node to its left, then it is not strictly necessary to define a leaf node at the end of this edge, because whether we can change direction at $i$ is only dependent on the back-ability of node $i$, which is determined in the preprocessing stage. Such edges can be used to accommodate double-back moves, however (in which case the train proceeds from $2(i)$ to $3(i)$, or vice versa).

We create two layers of the graph, which we will refer to as the Push and Pull layers. In general, these Push and Pull layers are independent of the direction of movement, and simply differentiate the orientation of the train. As illustrated in Figure 1.12, we create four copies of each node, with two in each of the Push and Pull layers of the graph. We now have upper and lower layers as well, where the copy of node $i$ in the upper layer corresponds to entry via edge $1(i)$ (call these $iU^+$ for pulling, and $iU^-$ for pushing), while the copy of node $i$ in the lower layer corresponds to entry via edge $2(i)$ or $3(i)$ (call these $iL^+$ for pulling and $iL^-$ for pushing). For instance, node $iU^+$ corresponds to the train entering node $i$ via $1(i)$ with locomotive pulling the railcars.

We next define four necessary arc types to create our shortest path graph (note that we will use the notation $2(i)/3(i)$ to correspond to either edge $2(i)$ or edge $3(i)$). Figure 1.12 illustrates the shortest path graph for the example problem illustrated in Figure 1.4(a), assuming all acute angles in the network are backable (the figure omits some arc directions for convenience).

1. **Source Arcs:** Recall that we consider at most $m + 2$ valid starting nodes. From the source node, if leading with the + end of the train (pulling railcars) requires approaching start node $i$ using $1(i)$ $(2(i)/3(i))$, then we connect the source node to $iU^+$ $(iL^+)$. Alternatively, if leading with the $-$ end of the train (pushing) requires approaching start node $i$ using $1(i)$ $(2(i)/3(i))$, then we connect the source node to $iU^-$ $(iL^-)$. The length of the edge equals the shortest distance required to move the train to the corresponding start node.

2. **Sink Arcs:** For each of at most $\eta + 2$ feasible entry nodes, if the final orientation of the train requires that the locomotive approaches the entry node with the (+) end of the train (pulling railcars), then we connect the entry node to the sink node only from the Pull layer. On the other hand, if the final layout must be achieved with locomotive pushing the railcars, then we connect the entry node to sink node from the Push layer. The associated edge length equals the distance required to move the train from the feasible entry node to its final position.

3. **Through Arcs:** For each edge $(i, j)$ in the rail network:

   - If edge $2(i)/3(i)$ equals edge $1(j)$, i.e., pulling into node $i$ via $1(i)$ and using edge $(i, j)$ leads to pulling into node $j$ via $1(j)$, we create an arc $(iU^+, jU^+)$. We also create arc $(iU^-, jU^-)$ corresponding to travel on edge $(i, j)$ from $i$ to $j$ with the locomotive pushing the railcars.

- If edge $2(i)/3(i)$ equals $2(j)/3(j)$, we create an arc $(iU^{+}, jL^{+})$, which corresponds to pulling into node $i$ via $1(i)$ and traveling directly to node $j$ via $2(j)/3(j)$. We also create arc $(iU^{-}, jL^{-})$ corresponding to traveling on edge $(i, j)$ from $i$ to $j$ with the locomotive pushing the railcars.
- If edge $1(i)$ equals edge $2(j)/3(j)$, we create an arc $(iL^{+}, jL^{+})$ corresponding to pulling into node $i$ via $2(i)/3(i)$ and traveling directly to node $j$ via $2(j)/3(j)$. We also create arc $(iL^{-}, jL^{-})$ corresponding to traveling on edge $(i, j)$ from $i$ to $j$ with the locomotive pushing the railcars
- If edge $1(i)$ equals $1(j)$, we create an arc $(iL^{+}, jU^{+})$ which represents pulling into node $i$ via $2(i)/3(i)$ and traveling directly to node $j$ via $1(j)$. We also create arc $(iL^{-}, jU^{-})$ corresponding to traveling on edge $(i, j)$ from $i$ to $j$ with the locomotive pushing the rail cars.

4. **Double-Back Arcs:** Cross-layer arcs connecting the Pull and Push layers enable double-back moves. Therefore, arc $(iL^{+}, jL^{-})$ indicates that the train pulls railcars into node $i$ via $2(i)/3(i)$, makes a double-back move resulting in an orientation change, and pushes the railcars into node $j$ via $2(j)/3(j)$. Similarly, arc $(iL^{-}, jL^{+})$ represents the move with the reverse orientation. On the other hand, arc $(iL^{+}, jU^{-})$ permits pulling railcars into node $i$ via $2(i)/3(i)$, making a double-back move, and continuing via pushing into $1(j)$. The arc $(iL^{-}, jU^{+})$ corresponds to the same move with the orientation reversed.

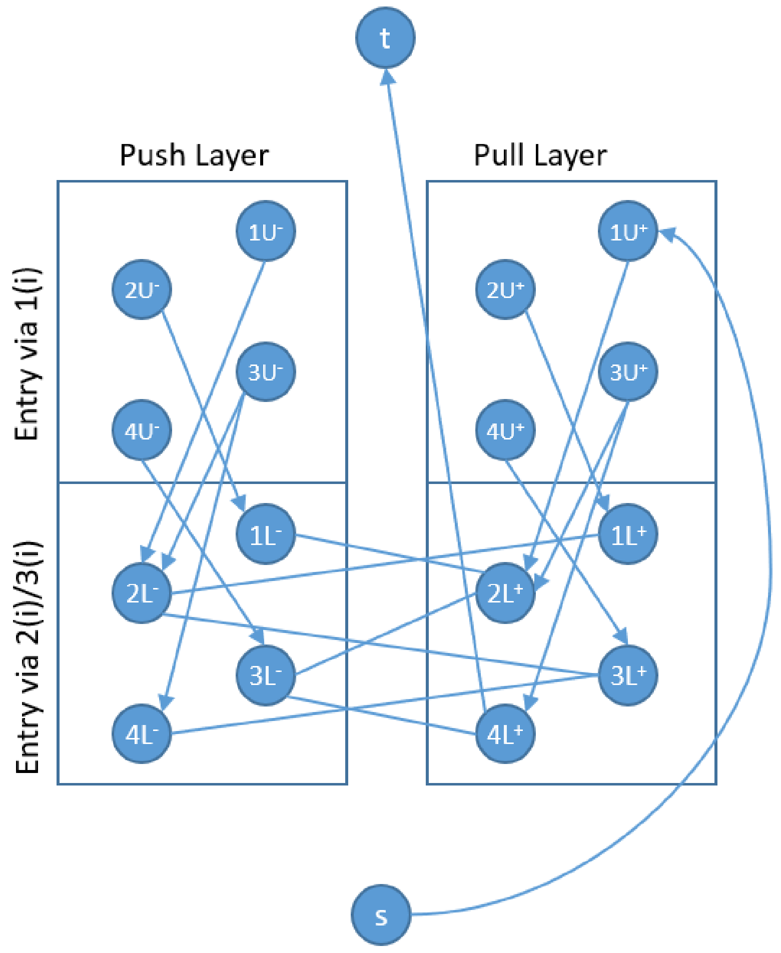


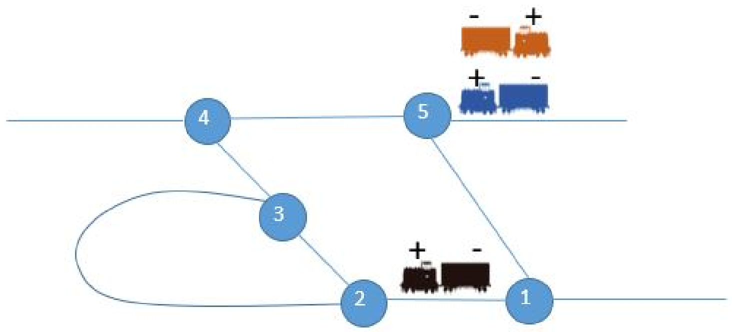
Figure 1.12: Expanded shortest path graph (some arc directions are omitted for convenience).

The arcs in the graph within the Push and Pull layers now represent the lengths of the corresponding edges in the original network, while the arc lengths between layers include the length of the edge plus the train's length $L$. An edge within the Pull (Push) layer corresponds to a simple move from one node to another without making any double-back move, while an edge between the two layers corresponds to an acute angle turn via a backable node. As a result, the between (Push and Pull) layer edges correspond to entering and leaving some node $j$ via edge $2(j)$ or $3(j)$ (and, therefore, such edges always depart the Push or Pull layer from the set of nodes in lower layers). Note that in this structure, the pull or push orientation does not change when train traverses an

AAF cycle and the associated cycle arcs remain within the pull or push layer for such cycles.

We call this graph the expanded shortest path graph. Each feasible train route in the original railyard network has a corresponding path in the expanded graph, and the shortest path from the source node to the sink node in the graph thus gives the shortest train route in the railyard network. Note that each time we jump from the left layer to the right (or vice versa), this implies that the train has changed orientation in terms of pulling or pushing, via a double-back move at a switch node. As discussed before, this improved approach does not require the all-shortest-paths solution for each node, and this results in better worst-case complexity.

To demonstrate how our improved approach addresses feasible routes and changes in orientation even when acute-angle-free cycles exist, we provide the solutions for the example shown in Figure 1.13. In this example, we want the train on the lower track (colored in black) to get to the upper right location. Both orientations (colored in blue and orange) are acceptable in this case.



| | |
|---|---|
| Start nodes: | 1L- and 2U+ |
| Entry nodes: | 5L+ and 5U- |
| Entry nodes: | 5L- and 5U+ |

Figure 1.13: Example network to demonstrate cycle and train orientation

We create the expanded network associated with this example. Figure 1.14 illustrates this network. The dashed lines are the source and sink arcs. For example, in order to have the train at its destination with the orientation colored in orange, the train may pull right into node $5$ via $1(5)$. This is associated with the dashed orange arc connecting $5U^+$ to $t$, or it may push toward node $5$ via $2(5)$, the lower arc. This is associated with the dashed orange arc connecting $5L^-$ to $t$. Similarly, related arcs are constructed for the train orientation colored in blue. In addition, we have the dashed black arcs for the initial location of the train. The rest of the arcs are constructed following the steps discussed before. The coloring of these arcs are only for better visibility of the network.

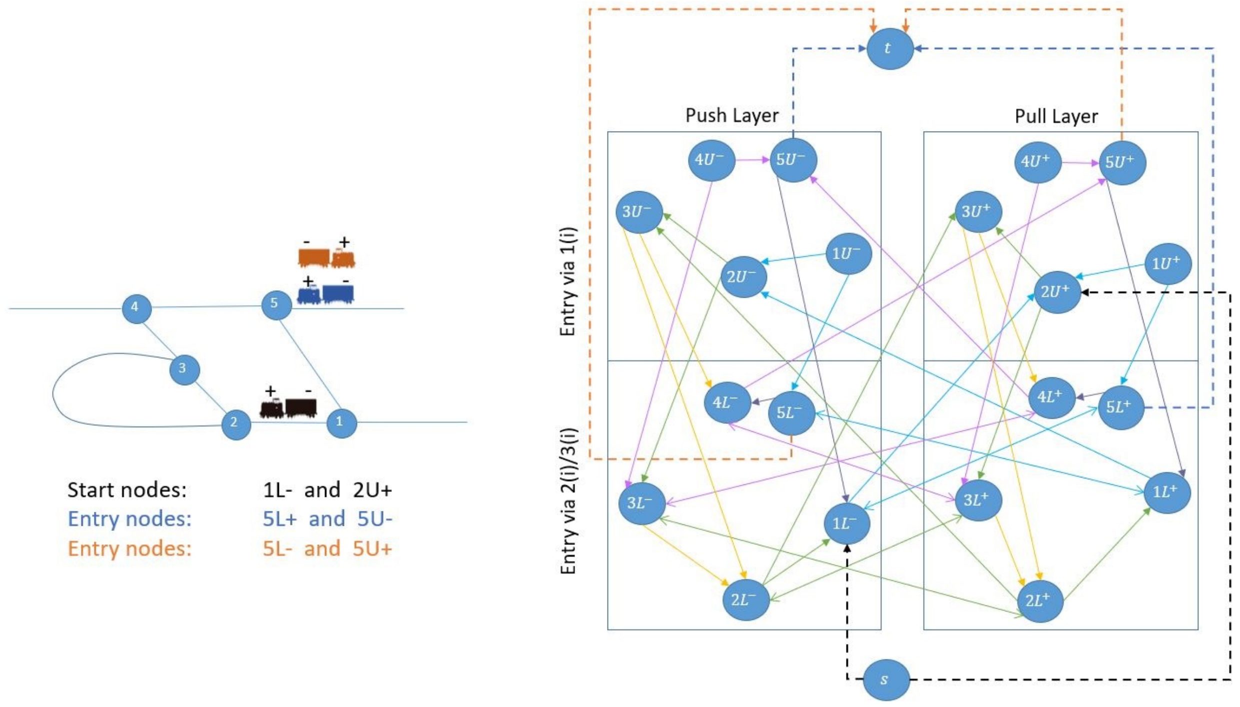


Figure 1.14: Expanded network for the example represented in Figure 1.13

Figures 1.15-1.18 demonstrate the four feasible routes for this example along with the expanded network's solution. The highlighted arcs show the node to node paths. Note that the first and second routes do not use the cycle, whereas the third and fourth routes utilize the cycle. Although the latter routes are not the shortest route for this instance, they are feasible and we present them to demonstrate the behavior of the train when acute-angle-free cycles are part of the routes.

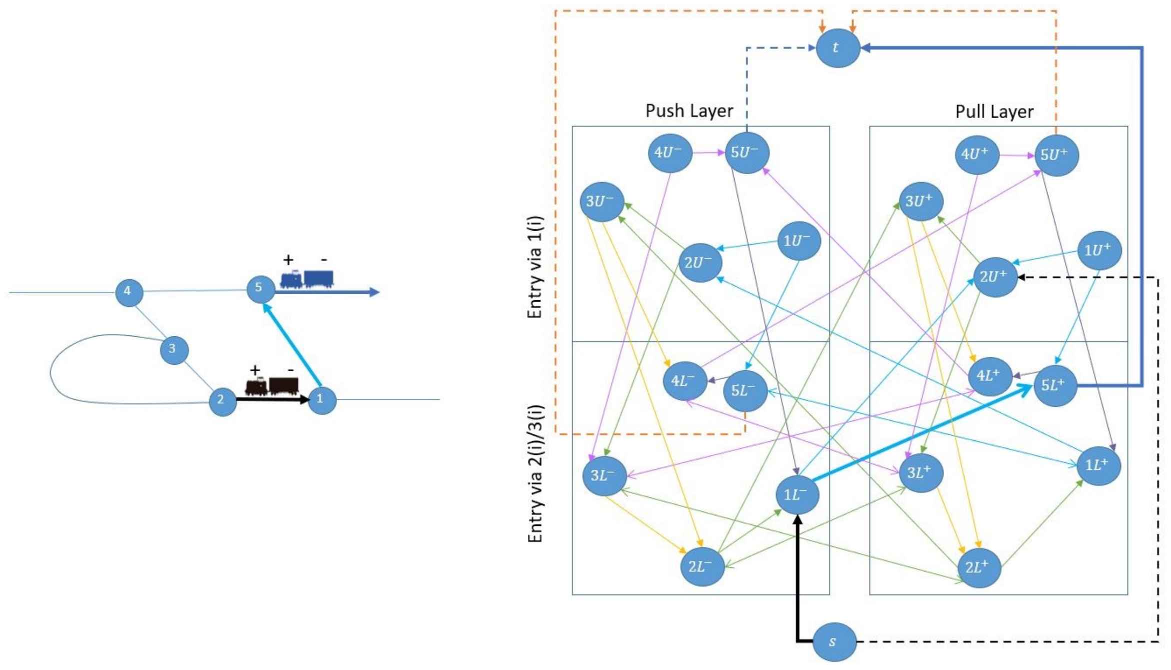


Figure 1.15: Solution 1 for the example represented in Figure 1.13

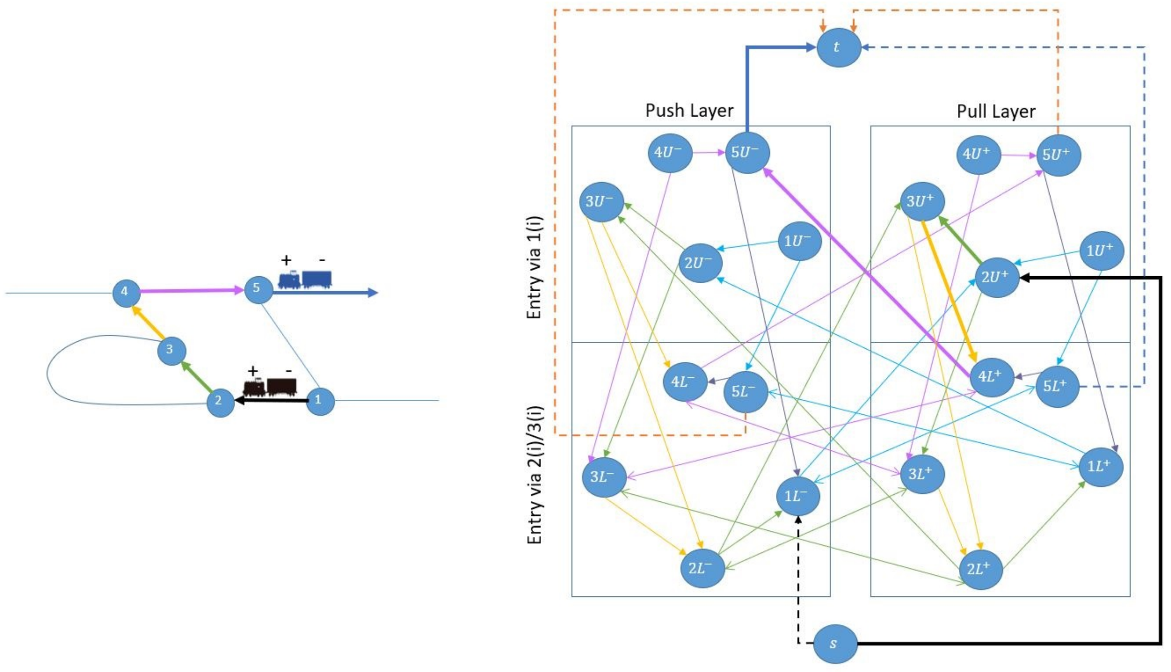


Figure 1.16: Solution 2 for the example represented in Figure 1.13

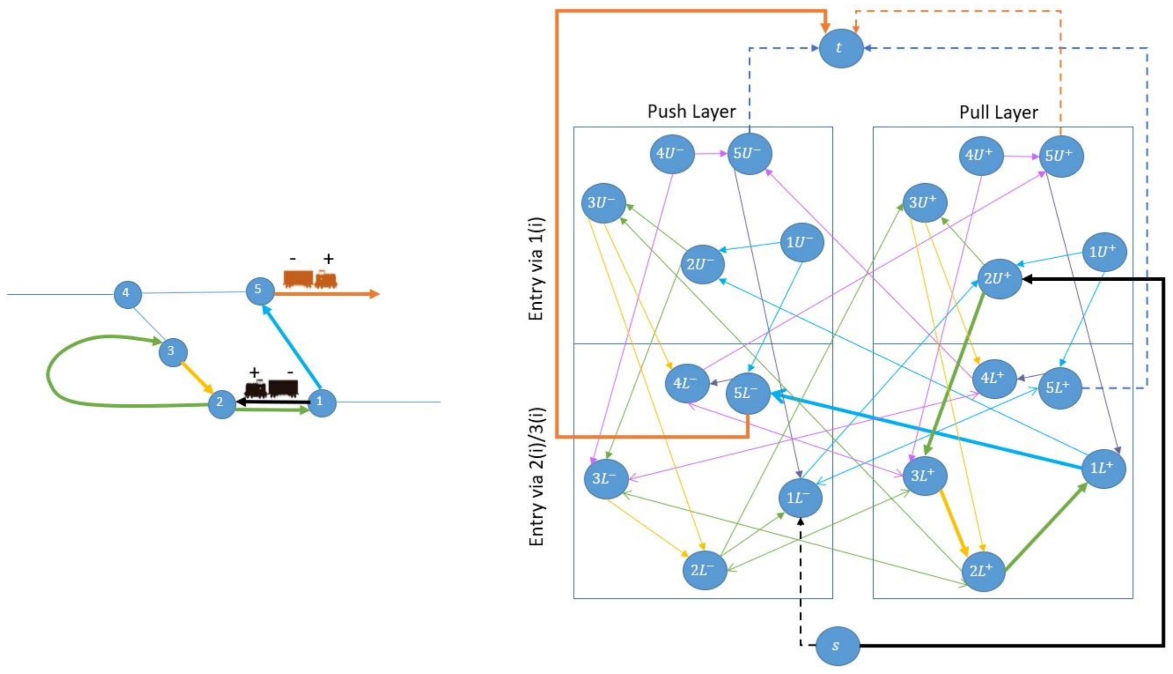


Figure 1.17: Solution 3 for the example represented in Figure 1.13

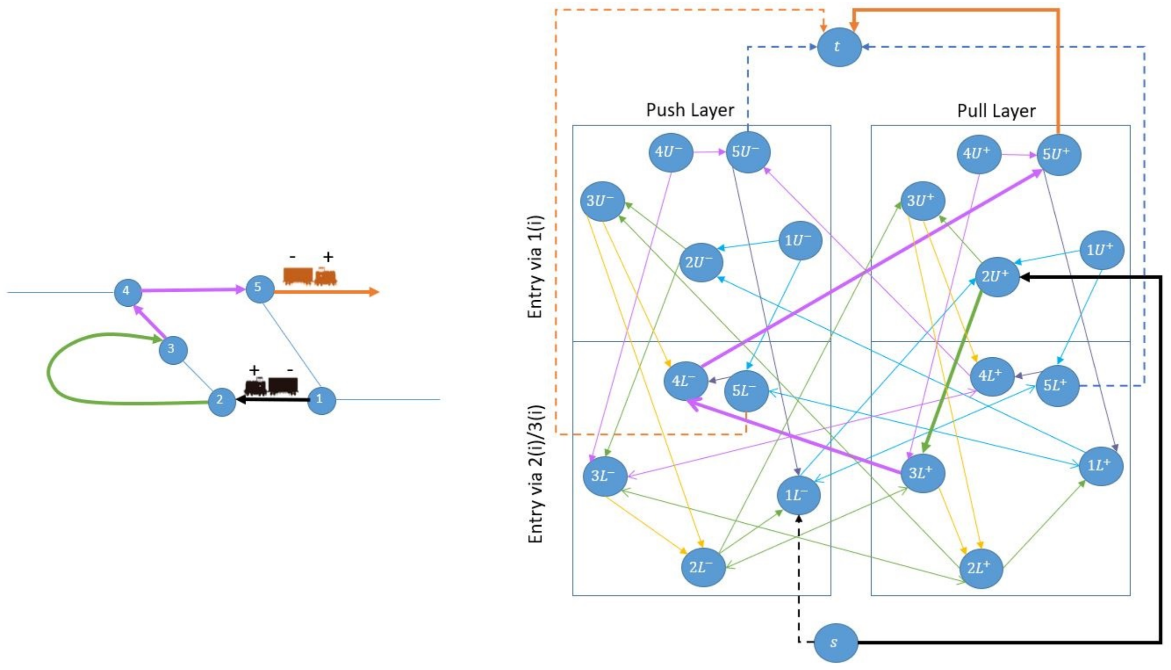


Figure 1.18: Solution 4 for the example represented in Figure 1.13

We followed the steps discussed earlier to generate the expanded network. Note that we presented a very small instance. Obviously for a larger network, the expanded network would become more dense. In the next section, we will discuss the complexity of the algorithms we proposed.

## 1.5 Complexity

The shortest path graph associated with the improved approach contains $4n + 2$ nodes and $4e + 2n$ arcs. Using a Fibonacci heap implementation of Dijkstra's algorithm to compare the two approaches, the solution of this shortest path problem thus has worst-case complexity of $O(e + n \log n)$ for the improved approach. Although the complexity of the algorithms we provided are $O(ne + n^2 \log n)$ and $O(e + n \log n)$ for the all-shortest-paths and improved approaches respectively, the required number of nodes and arcs for the final graph are as follows:

- All-shortest paths approach: The number of nodes is $4n + 2$, and the number of edges is at most $2n^2 + 2n$;
- Improved approach: The number of nodes is $4n + 2$, and the number of edges is at most $4e + 2n$,

where $n$ and $e$ denote the number of switch nodes and edges in the original network, respectively. Note that if we ignore the pulling and pushing requirements in the All-shortest-Paths approach, the associated numbers of nodes and edges become $2n + 2$ and $n^2 + 2n$, respectively. However, as discussed in section 1.4.1, the complexity of this approach is dominated by the need to compute shortest paths for each node's $T$-reachable and $O$-reachable sub-networks, which results in an overall complexity of $\mathcal{O}(ne + n^2 \log n)$ using Fibonacci heap implementation of Dijkstra's algorithm ($\mathcal{O}(n^3)$ if simple Dijkstra's algorithm is used).

The number of nodes and edges of the final graph in the approach proposed in [1] are $4m + 8n + 2$ and $4e + 12n + 4$ respectively, where $m$ is the total number of nodes (switch and leaf nodes). Thus, although our approach utilizes the preprocessing step introduced in [1], and both approaches solve a shortest path problem over a transformed network representation, it is apparent that our improved approach requires fewer edges in the final graph, which is the key to determining the required computational effort in shortest path algorithms.

Figure 1.19 shows the reduction in the number of nodes and arcs in the corresponding shortest path graph provided by the improved approach when compared with the method in [1] (the figure assumes $e = \frac{3n}{2}$, where $n$ is the number of switch nodes, and that the number of leaf nodes equals 10% of the number of switch nodes). The real-world instances tested in [1] contain around 4,600 nodes. Thus, the difference in computational time and effort can be non-trivial when needing to solve problems of this type repeatedly in practice.

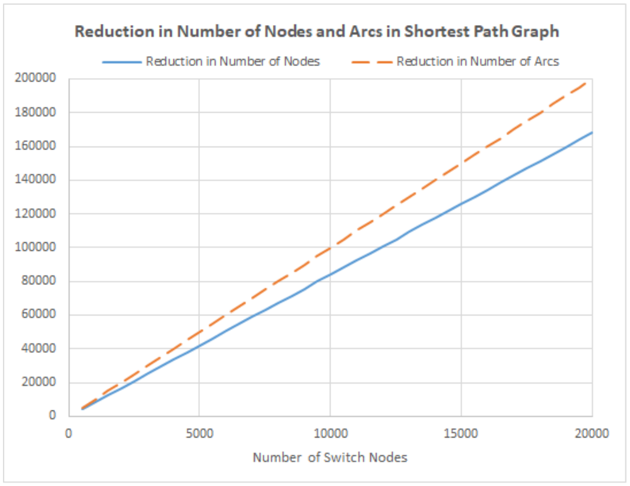


Figure 1.19: Reduction in shortest path graph size when compared with the approach in [1].

## 1.6 Discussion and Conclusion

This work proposed two related approaches for constructing an expanded version of a railyard network, over which any shortest path algorithm can be applied to find the shortest feasible route of a single train from an initial position to a final location on the network. We based our analysis on a Fibonacci heap implementation of Dijkstra's classical shortest path algorithm, although any shortest path algorithm can be utilized over the constructed networks to find the shortest feasible route solution. The proposed polynomial-time algorithms account for the geometric restrictions of the railyard network, with switch points serving as possible points for changing direction if the length of track connected to the switch node can accommodate the length of the train. We note that the proposed improved approach is easier to implement compared to previous works because it only requires accounting for the network's switch nodes and also uses a more compact network representation. Another major contribution of our approach is that it allows a train to span multiple nodes in the network at any point in time, while accounting for train orientation by distinguishing between pull and push moves and expanding the shortest path approach accordingly. Future work will include the problem of determining the combination of paths on the network for multiple trains that minimizes the time for all trains to reach their final positions without collisions.

# 2. MULTIPLE TRAIN REPOSITIONING IN A RAILYARD NETWORK

## 2.1 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 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 and rapid 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 for all trains to reach to their final destination positions on the network. We call this the multiple-train shortest-route (MTSR) problem.

Evidently, if we are able to determine the shortest route independently for each train, 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, which is extensively discussed in Chapter 1, is fundamentally different from classical shortest path problems, as it requires explicit consideration of the structure of the railyard network. 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 (see Chapter 1 for detail discussion on the single-train shortest route problem).

It will typically be the case that solving the 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 track segment 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 occupy a common track segment or switch point 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 simultaneously reposition trains in a railyard network, 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 2.3 then provides a formal problem definition. In Section 2.4, we provide an

$\mathcal{NP}$-Hardness proof for the MTSR problem, after which Section 2.5 discusses an exact model and a heuristic algorithm for efficiently determining a solution. Section 2.6 then characterizes the advantages and disadvantages of the two approaches via a set of computational tests, and we provide concluding remarks in section 2.7.

## 2.2 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 works focused on passenger rail traffic between train stations within a city or country. The associated railway 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, Larsen, Ehrgott, and Ryan [13] 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, Kozan, and Ferreira [14] 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 nonlinear mixed-integer programming (MIP) 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 with higher priority to pass along a shared track segment. Carey and Lockwood [15], 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 [16] 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 [11] revisit this problem with fixed train routes with associated desired departure and arrival times. The authors provide an MIP formulation to minimize total tardiness for the cases in which 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 (Lusby, et al. [13]). 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, Fischetti, and Toth [17] 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, Caprara, and Toth [18] 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. [13] 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 authors' 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, Kroon, Romeijn, and Salomon [9] 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 Zwaneveld, et al. [9] is to maximize 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 through the network). Their proposed 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. Note that 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 [19] 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 [20] propose an alternative graph formulation to tackle the job shop problem with blocking and no-wait constraints. D'Ariano, Pacciarelli, and Pranzo [21] 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 [22], who considers the real-time routing of trains 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 is occupied by another train. Rodriguez [22] 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 [23] use such an approach as a feasibility check for a passenger railway aggregated inter-station routing solution.

While the approaches we have discussed can generally be tackled using branch-and-bound methods (Lusby, et al. [13]), in many cases, the problem's size and complexity can restrict the likelihood that such an exact method is able to produce a good solution in acceptable time. As a result, numerous heuristic solutions 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 we have discussed considered route planning and execution operations at the aggregate level (via the network interconnecting different stations), as well as intra-station routing and platform assignment. Several of these works required predetermining 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 the station network. While these issues typically do not affect scheduled 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. [9], Rodriguez [22], and D'Ariano, et al. [21], which consider routing trains through a network while accounting for collision avoidance and the delays introduced when trains must vie for common track segments. However, these prior works do not account for the complexity introduced when simultaneously repositioning freight trains within the network via switch points. In the remainder of this paper, we address these additional challenges via the formulation and solution of the MTSR problem.

## 2.3 Problem 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 such segment can be characterized as a ramp track or support track, connected 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, e.g., empty railcars, assembled outbound trains waiting to depart, or newly arriving inbound trains that have not yet been scheduled for unit unloading.

Track segments are interconnected via switch points, where more than two track segments meet, permitting travel between track segments. We will assume without loss of generality (as also discussed in Chapter 1) that a switch point corresponds to a point at which exactly three track segments meets (if more than three segments meet at a switch point, we decompose the switch into multiple switch points separated by zero distance). At each such switch point, two pairs of converging track segments form obtuse angles, which 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 will contain a node for every switch point as well as an edge corresponding to each track segment.

Figure 2.1 illustrates a sample layout of a railyard with various track segments and switch nodes.

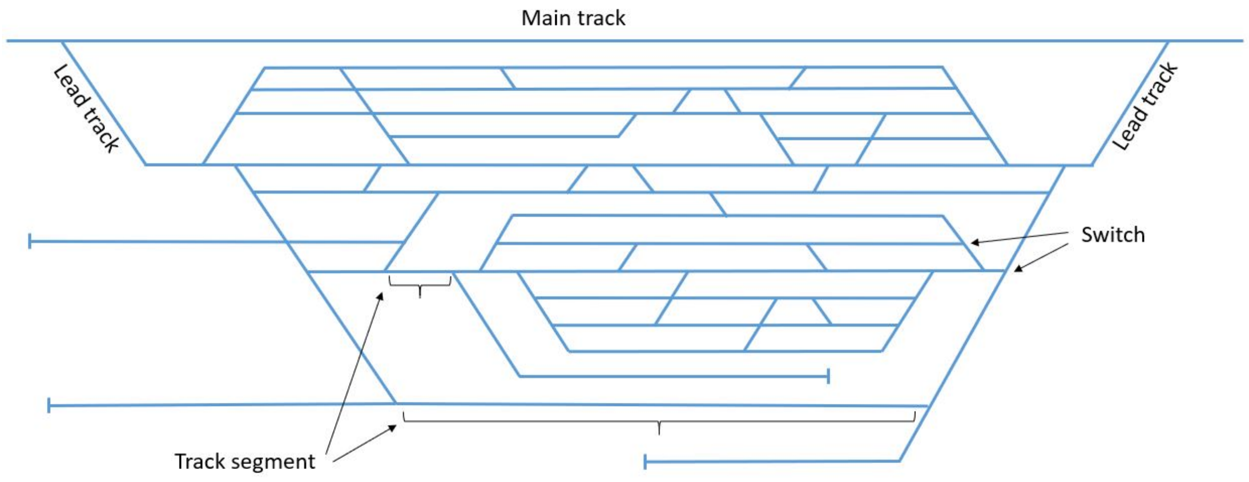


Figure 2.1: Illustration of a sample railyard, with tracks and switches

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 Figure 2.2, 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.

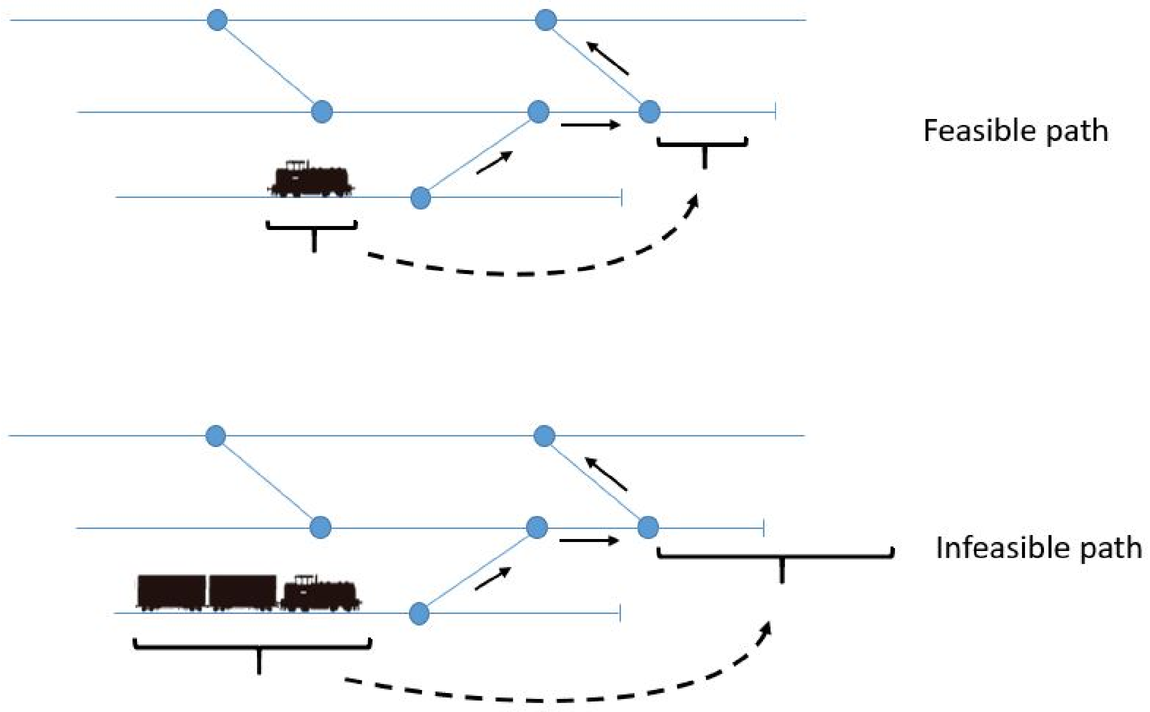


Figure 2.2: The train length effect on feasibility of a path

The problem we consider involves 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 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 2.3 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.

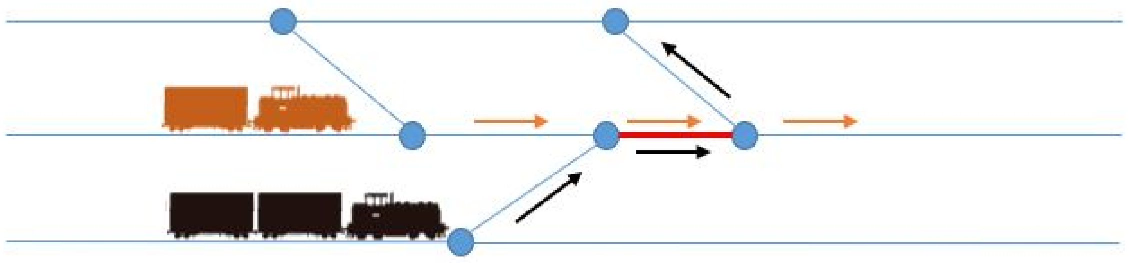


Figure 2.3: The delay concept and the importance of train layout at each time

The goal of the MTSR problem is to minimize the time required to move a set of trains from their origin locations to destination locations on the network. 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. 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.

## 2.4 Problem Complexity

In this section, we show that the MTSR problem when trains may have any nonnegative values of train length is $\mathcal{NP}$-Hard. We utilize results from Peis, Skutella, and Wiese [24], 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. [24] use a reduction from the 3-BOUNDED-3-SAT problem. The 3-BOUNDED 3-SAT problem contains a set of $m$ clauses $C_1 \ldots C_m$, 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. [24], given an instance of 3-BOUNDED 3-SAT, they construct a packet network with a polynomial number of vertices and packets for each variable and clause. They also create a polynomial number of packets corresponding to each variable and, for each clause, a route for each corresponding packet is determined that passes 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 6 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. In the following, we transform their packet network into a corresponding special case of our railyard 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 $\mathcal{P} = \mathcal{NP}$.

We can transform their planar packet network into a corresponding railyard network in polynomial time, where the solution to the corresponding special case of the MTSR problem on this rail network solves the associated 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 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, 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 Figure 2.4). Thus, a train located on one of the parallel arcs in the rail network corresponds to a packet 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, which is equivalent to effectively assuming they all have the same length, i.e., in our network, all trains can be of the same length, which can be arbitrarily small. Because of this, although there is a possibility of conflicts at nodes after introducing the 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 non-negative 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 6 if and only if the minimum makespan in the rail network is at most 6, and both of these 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.

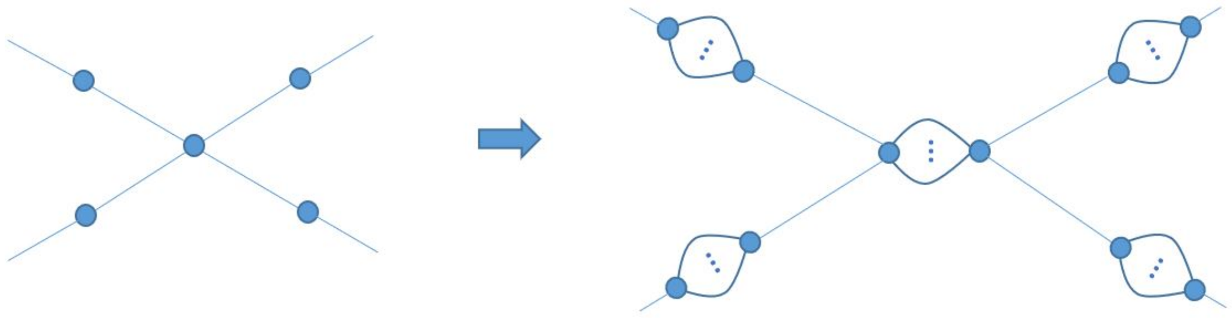


Figure 2.4: Creating parallel tracks to represent a node of unlimited capacity

## 2.5 Solution Approach

Efficient execution of operations and high throughput levels at intermodal terminals necessitate an effective tool for planning railcar moves within 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 for each train to reach its destination. Section 2.5.1 first provides a mathematical model for this problem by formulating it as a large-scale 0-1 integer program. While the resulting model is useful for solving small problem instances and permits providing a formal mathematical statement of the problem, 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, Section 2.5.3 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 Section 2.6 that demonstrate the effectiveness of this heuristic method in providing high-quality solutions in acceptable computing time.

### 2.5.1 Integer Programming Model

This section provides a large-scale 0-1 integer programming model for the MTSR problem, the input for which consists of a railyard network configuration and a set of trains, each with an origin and destination location and associated train length, to determine a set of collision-free shortest routes that minimizes the total time for each train to reach its destination location.

As noted earlier, a railyard consists of a set of connected track segments. In our modeling approach, we decompose each track segment into a discrete set of slots, i.e., each physical track segment is divided into a set of non-overlapping track subsections called slots. This approach permits accounting for the location of each train at all points in time. We assume that this slot size has a length at least as great as the length of the the longest railcar on the network. This ensures that any slot is capable of accommodating any railcar size at any point in time. Note that setting the slot size to a length equal to that of the longest railcar on the network may lead to an overestimate of the length on the network if railcar lengths vary significantly. On the other hand, when railcars have a standard length, or when each occupied slot accommodates multiple connected railcars equal to the slot's length, this overestimation does not occur (in the latter case, if multiple connected cars occupy a slot, we treat these cars as a single car throughout a sequence of moves). Figure 2.5 illustrates the decomposition of track segments into slots that can accommodate railcars. We then create a slot index number for each slot, thus characterizing the railyard as a a set of connected slots of equal length.

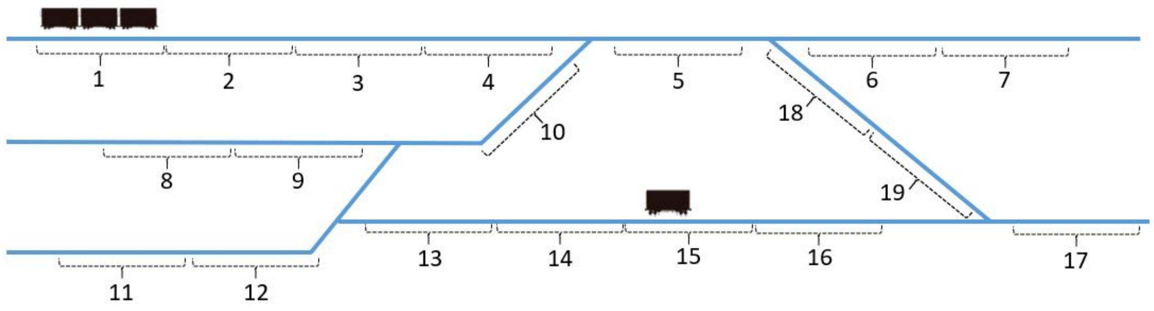


Figure 2.5: Breaking a track into numbered slots

We next specify how slots are interconnected. That is, for each slot number, we determine the set of slot(s) that are directly accessible to the right of the slot, and the set of slot(s) that are directly accessible to the left. In Figure 2.5, for example, the slots to the left of slot 10 are {9, 12}, while the slot to the right of slot 10 is $\{5\}$. 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. Observe that a network's layout and the definition of a slot's standard length may lead to slight differences in the distances from one slot to its adjacent slots. For example, in Figure 2.5, the distance from slot 10 to adjacent slot 12 is greater than the distance from slot 10 to slot adjacent slot 5. We assume that such differences in adjacent slot distances are negligible with respect to the time required for a car (or set of cars) to move from one slot to any adjacent slot.

For ease of exposition, we assume a network topology that permits establishing "right" and "left" travel directions, although this is not strictly necessary. This is possible for any railyard network by using the following conventions. 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 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 unit of time. 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 $c \in C$, $s \in S$, $t = 0, \ldots, T,$;
- $Z_t = 1$ if all cars are in their final slots at time $t$, and 0 otherwise, for $t = 1, \ldots, T,$;

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

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

$$\text{Minimize} \quad \sum_{t=1}^{T} tZ_t \tag{2.1}$$

$$\text{s.t.} \quad X_{cs_c^o0} = 1, \qquad c \in C, \tag{2.2}$$

$$X_{cs,t+1} \leq \sum_{i\in L_s} X_{cit} + X_{cst} + \sum_{i\in R_s} X_{cit}, \qquad c \in C, s \in S, t = 0, \ldots, T-1, \tag{2.3}$$

$$\sum_{s\in S} X_{cst} = 1, \qquad c \in C, t = 1, \ldots, T, \tag{2.4}$$

$$\sum_{c\in C} X_{cst} \leq 1, \qquad s \in S, t = 1, \ldots, T, \tag{2.5}$$

$$Z_t \leq X_{cs_c^dt}, \qquad c \in C, t = 1, \ldots, T, \tag{2.6}$$

$$\sum_{t=1}^{T} Z_t \geq 1, \tag{2.7}$$

$$X_{cst} \leq \sum_{i\in R_s} X_{RH_cit} \leq 2 - X_{cst}, \qquad c \in CC, s \in S, t = 0, \ldots, T, \tag{2.8}$$

$$\sum_{i\in R_s} X_{ci,t+1} + X_{cst} \leq GR_{st} + 1, \qquad c \in C, s \in S, t = 0, \ldots, T-1, \tag{2.9}$$

$$\sum_{i\in L_s} X_{ci,t+1} + X_{cst} \leq GL_{st} + 1, \qquad c \in C, s \in S, t = 0, \ldots, T-1, \tag{2.10}$$

$$GR_{st} + \sum_{i\in R_s} GL_{it} \leq 1 \qquad s \in S, t = 0, \ldots, T, \tag{2.11}$$

$$GL_{st} + \sum_{i\in L_s} GR_{it} \leq 1 \qquad s \in S, t = 0, \ldots, T, \tag{2.12}$$

$$X_{cst} \in \{0, 1\}, \qquad c \in C, s \in S, t = 0, \ldots, T, \tag{2.13}$$

$$Z_t \in \{0, 1\}, \qquad t = 1, \ldots, T, \tag{2.14}$$

$$GR_{st} \in \{0, 1\}, \qquad s \in S, t = 0, \ldots, T, \tag{2.15}$$

$$GL_{st} \in \{0, 1\}, \qquad s \in S, t = 0, \ldots, T. \tag{2.16}$$

The objective function (2.1) minimizes the total time required for each car to reach its destination slot, while Constraint set (2.2) sets the initial locations of the cars and locomotives. Constraint set (2.3) ensures that each car either moves to or from an adjacent slot at each time step, or remains in its previous location. Constraint set (2.4) requires that each car occupies exactly one slot at each time period, while Constraint set (2.5) allows at most one car to occupy a slot in each period. Constraint set (2.6) allows $Z_t$ to equal one when all railcars reach their destination locations, and Constraint (2.7) ensures that the final state is achieved at some point in time. Constraint (2.8) requires that all right connected cars remain connected throughout all time periods. Constraints (2.9)–(2.12) 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 (2.13)–(2.16) 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 total time required for all railcars to reach 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 also includes the specific nature of locomotives.

### 2.5.2 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 lie 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 that are 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 $s \in S$, $t = 0, \ldots, T$, and;
- $W_{st} = 1$ if slot $s$ is accessible to a locomotive from the left, 0 otherwise, for $s \in S$, $t = 0, \ldots, T$.

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.

$$\text{Minimize} \quad \sum_{t=0}^{T} tZ_t \tag{2.17}$$

$$\text{s.t.} \quad X_{cs_c^o 0} = 1, \qquad c \in C, \tag{2.18}$$

$$X_{cst+1} \leq \sum_{i \in L_s} X_{cit} + X_{cst} + \sum_{i \in R_s} X_{cit}, \quad c \in C, s \in S, t = 0, \ldots, T-1, \tag{2.19}$$

$$\sum_{s \in S} X_{cst} = 1, \qquad c \in C, t = 1, \ldots, T, \tag{2.20}$$

$$\sum_{c \in C} X_{cst} \leq 1, \qquad s \in S, t = 1, \ldots, T, \tag{2.21}$$

$$Z_t \leq X_{cs_c^d t}, \qquad c \in C, t = 1, \ldots, T, \tag{2.22}$$

$$\sum_{t=1}^{T} Z_t \geq 1, \tag{2.23}$$

$$X_{cst} \leq \sum_{i \in R_s} X_{RH_c it} \leq 2 - X_{cst}, \qquad c \in CC, s \in S, t = 0, \ldots, T, \tag{2.24}$$

$$E_{st} \leq \sum_{c \in C} X_{cst}, \qquad s \in S, t = 0, \ldots, T, \tag{2.25}$$

$$W_{st} \leq \sum_{c \in C} X_{cst}, \qquad s \in S, t = 0, \ldots, T, \tag{2.26}$$

$$E_{st} \leq \sum_{c \in Loco} X_{cst} + \sum_{i \in R_s} E_{it}, \qquad s \in S, t = 0, \dots, T, \tag{2.27}$$

$$W_{st} \leq \sum_{c \in Loco} X_{cst} + \sum_{i \in L_s} W_{it}, \qquad s \in S, t = 0, \dots, T, \tag{2.28}$$

$$\sum_{i \in R_s} X_{ci,t+1} + X_{cst} \leq GR_{st} + 1, \qquad c \in C, s \in S, t = 0, \dots, T-1, \tag{2.29}$$

$$\sum_{i \in L_s} X_{ci,t+1} + X_{cst} \leq GL_{st} + 1, \qquad c \in C, s \in S, t = 0, \dots, T-1, \tag{2.30}$$

$$GR_{st} + \sum_{i \in R_s} GL_{it} \leq 1, \qquad s \in S, t = 0, \dots, T, \tag{2.31}$$

$$GL_{st} + \sum_{i \in L_s} GR_{it} \leq 1, \qquad s \in S, t = 0, \dots, T, \tag{2.32}$$

$$GR_{st} + GL_{st} \leq E_{st} + W_{st}, \qquad s \in S, t = 0, \dots, T, \tag{2.33}$$

$$2 * GR_{st} \leq E_{st} + W_{st} + \sum_{i \in R_s} (E_{i,t+1} + W_{i,t+1}), \qquad s \in S, t = 0, \dots, T-1, \tag{2.34}$$

$$2 * GL_{st} \leq E_{st} + W_{st} + \sum_{i \in L_s} (E_{i,t+1} + W_{i,t+1}), \qquad s \in S, t = 0, \dots, T-1, \tag{2.35}$$

$$\sum_{c \in C} (X_{cs,t+1} - X_{cst}) \leq E_{s,t+1} + W_{s,t+1}, \qquad s \in S, t = 0, \dots, T-1, \tag{2.36}$$

$$X_{cst} \in \{0, 1\}, \qquad c \in C, s \in S, t = 0, \dots, T, \tag{2.37}$$

$$Z_t \in \{0, 1\}, \qquad t = 1, \dots, T, \tag{2.38}$$

$$GR_{st} \in \{0, 1\}, \qquad s \in S, t = 0, \dots, T, \tag{2.39}$$

$$GL_{st} \in \{0, 1\}, \qquad s \in S, t = 0, \dots, T, \tag{2.40}$$

$$E_{st} \in \{0, 1\}, \qquad s \in S, t = 0, \dots, T, \tag{2.41}$$

$$W_{st} \in \{0, 1\}, \qquad s \in S, t = 0, \dots, T. \tag{2.42}$$

Constraints (2.25)–(2.28) 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 (2.33) 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 (2.34)–(2.36) 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 location's 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 section 2.5.1 to handle this locomotive extension heuristically. To see this, first consider a problem where the goal is to only assign locomotives to the trains we desire to move. 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 section 2.5.1 to determine the minimum time required to move the locomotives to the trains. Thereafter, we can use the same model to solve a multi-train repositioning problem for those trains with the attached locomotives, because we have already ensured that those trains have a locomotive attached. In other words, we decompose the mathematical model into two steps: first attaching locomotives to trains, and then moving the trains with locomotives attached.

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.

### 2.5.3 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, 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 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 scheduled operations, e.g., those trains with earlier scheduled departures or loading/unloading operations 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 Chapter 1, 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 [25] 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 2.6 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).

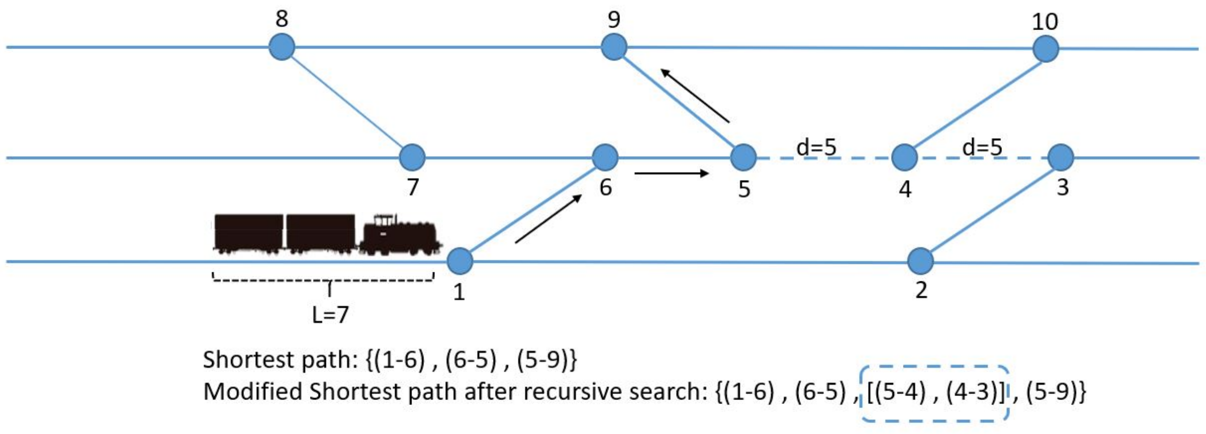

8
9
10
d=5
d=5
7
6
5
4
3
1
2
L=7
Shortest path: {(1-6) , (6-5) , (5-9)}
Modified Shortest path after recursive search: {(1-6) , (6-5) , [(5-4) , (4-3)] , (5-9)}

Figure 2.6: Adjusting shortest path to account for arcs utilized for double-back moves

Accounting for track segment conflicts requires associating a time interval with each train-segment pair. We assume all trains travel at the same speed and that this speed is proportional to distance traveled, 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 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 $m1$ and $m2$ in Figure 2.7 (assume for convenience that $m1$ has priority by virtue of its lower index number). Suppose the shortest route for train $m1$ consists of $1 \rightarrow 6 \rightarrow 5 \rightarrow 9$, while that for train $m2$ consists of $7 \rightarrow 6 \rightarrow 5 \rightarrow 4$. Because $m1$ must double-back at node 5, it also uses edge $(5, 4)$ to make this move. Thus, $m1$ and $m2$ both require using edges $(6, 5)$ and $(5, 4)$, creating a potential conflict. If $m1$ 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 $m2$ departs at time 0, it will occupy edge $(6, 5)$ from time 5 to time 15, which conflicts with $m1$'s use of this edge. We thus delay the time $m2$ begins on this edge by 12 time units until time 17 (when $m1$ completes traversing this edge). After incorporating this delay, $m2$ arrives at edge $(5, 4)$ at time 22. However, this edge is not completed by $m1$ until time 24, introducing an additional delay of 2 time units for $m2$ before it can proceed along this edge. $m2$ thus experiences a total delay of 12 time units along its path in order to allow $m1$ to complete its usage of conflicting edges. We update the times $m2$ 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 $m2$ occupies each edge along its route are shown below Figure 2.7.

This route-delay procedure is then applied to each of the $K$ shortest routes for $m2$. After adjusting each of the $K$ shortest route lengths for $m2$, we select the route with the earliest completion time and choose this as the fixed route for $m2$. 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 [26], although neither approach is guaranteed to dominate the other in general.

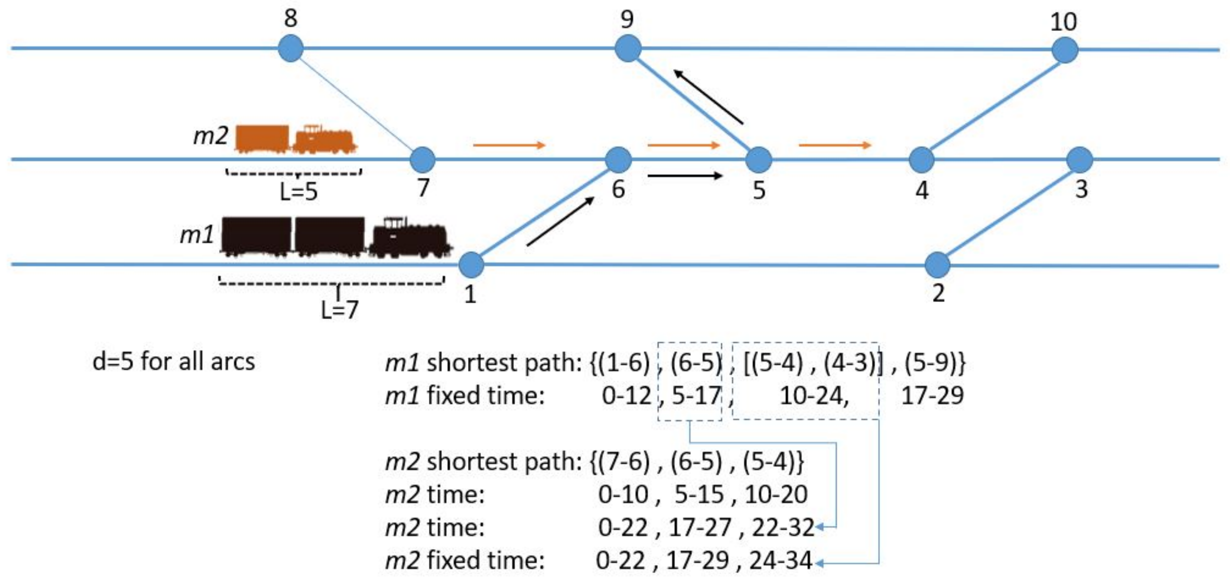

8
9
10
m2
L=5
7
6
5
4
3
m1
1
2
L=7
d=5 for all arcs
m1 shortest path: {(1-6) , (6-5) , [(5-4) , (4-3)] , (5-9)}
m1 fixed time: 0-12 , 5-17 , 10-24, 17-29
m2 shortest path: {(7-6) , (6-5) , (5-4)}
m2 time: 0-10 , 5-15 , 10-20
m2 time: 0-22 , 17-27 , 22-32
m2 fixed time: 0-22 , 17-29 , 24-34

Figure 2.7: Adjusting path time based on required delay

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.

We can determine a worst-case bound on the performance on 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

**Algorithm 1** FindFinalRoutes($M, K$)

```
for train $m \in M$ do
    Find $K$ shortest routes and sort in nondecreasing order of length: $P^m = \{P^m_1 \ldots P^m_K\}$
    for path $P^m_i \in P^m$ do
        for edge $c \in P^m_i$ 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 $P^m_i$ after edge $c$
            end if
        end for
        Let train $m$ start time equal zero
        for edge $c \in P^m_i$ 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 $P^m_i \in P^m$ do
        call ResolveConflict($P^m_i$, $FinalRoutes$)
    end for
    Let $P^m_{BestRoute}$ be the shortest route in $P^m$
    Add $P^m_{BestRoute}$ to $FinalRoutes$
end for
```

**Algorithm 2** ResolveConflict($P^m_i$,$FinalRoutes$)

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

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 $\frac{1-f}{f}\times 100\% = \left(\frac{1}{f}-1\right)\times 100\%$.

### 2.5.4 GRASP Extension

This section discusses a metaheuristic called GRASP 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, a starting 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 [27] explain the basics of implementing GRASP and how to link it to other metaheuristics for an improved result. In the following, we adopt the GRASP concept to improve our heuristic with a greedy randomized approach.

In the previous section, we prioritized trains and provided an ordered list of trains to begin with. For a pure greedy algorithm, we select the train with the minimum shortest path to begin with. Furthermore, at each iteration, we select a train route among the remaining trains that results in the least total time extension concerning the previously fixed routes. This is the same as what we discussed in the previous section, except that instead of an ordered list of trains, we pick the minimum route at each step.

Resende and Ribeiro [27] explain how to 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). Let $c^{min}$ be the minimum route total time (cost) among all candidate routes. Similarly let $c^{max}$ be the maximum total time among all candidate routes. At each iteration, routes with an incremental cost that lies 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 $\alpha$ grows, an increasingly randomized selection occurs, whereas with smaller values of $\alpha$ we restrict the RCL to the best routes in terms of total time. Figure 2.8 depicts the basic steps of this approach.

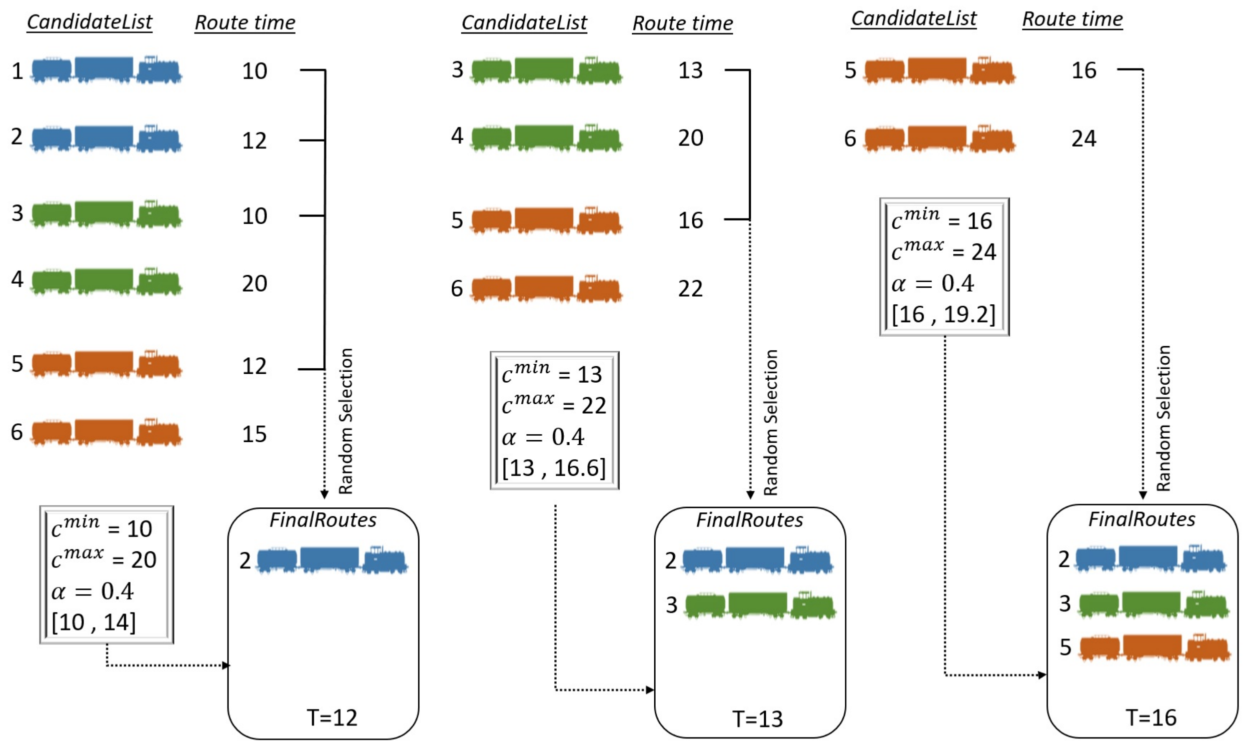


Figure 2.8: Greedy randomized selection of routes

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 $\alpha$ equal to zero, the route with minimum total time is indeed selected at each step. In our experiments we set $\alpha = 0.4$. Algorithm 3 provides the pseudocode for this approach, including a multi-start step that solves the problem multiple times and reports the best overall solution.

## 2.6 Computational Results

This section considers sample railyard networks and examines the optimization model's solution and the GRASP Extended (GE) heuristic approach. The optimization model segmentizes network tracks, and its solution provides a period-by-period location of each railcar, while our 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. Results for instances of relatively small size (with less than 100 track slots) are reported in Table 2.1. 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.

Table 2.1: Performance Comparison and Computational Results

| **Exact Model Solution Compared to Heuristic Model Solution** | **# of Samples** | **% of Samples** | **Avg. Exact Model Solution Time (sec)** | **Avg. Heuristic Model Solution Time (sec)** |
|---|---|---|---|---|
| **Exact = Heuristic** | 36 | 45% | 548 | 33 |
| **Exact >Heuristic** | 12 | 15% | 3600 | 36 |
| **Exact <Heuristic** | 32 | 40% | 825 | 35 |
| **Total** | 80 | 100% | 1117 | 34 |

Figure 2.9 shows the solution comparison gained via exact approach and heuristic approach. It can be observed that when heuristic solution outperforms the exact approach, the blue line lies above the purple line. These points are associated with the 12 instances where the exact approach was not able to find the optimal solution. Other than these twelve instances, the chart provides a representation of the proximity of the solutions achieved by the two approaches.

**Algorithm 3** Greedy Randomized Heuristic

```
$CandidateList \leftarrow \emptyset$
for train $m \in M$ do
    Find $K$ shortest routes for $m$: $P^m = \{P_1^m \dots P_K^m\}$
    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
        Add $P_i^m$ to $CandidateList$
    end for
end for
$BestSolution \leftarrow \emptyset$
$BestValue \leftarrow \infty$
for n = 1,...,MaxIterations do
    $FinalRoutes \leftarrow \emptyset$
    $CandidateList_n \leftarrow CandidateList$
    while $CandidateList_n \neq \emptyset$ do
        for route $r$ in $CandidateList_n$ 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 $CandidateList_n$ by removing the $K$ routes associated to $selectedRoute$'s re-
spective 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
```

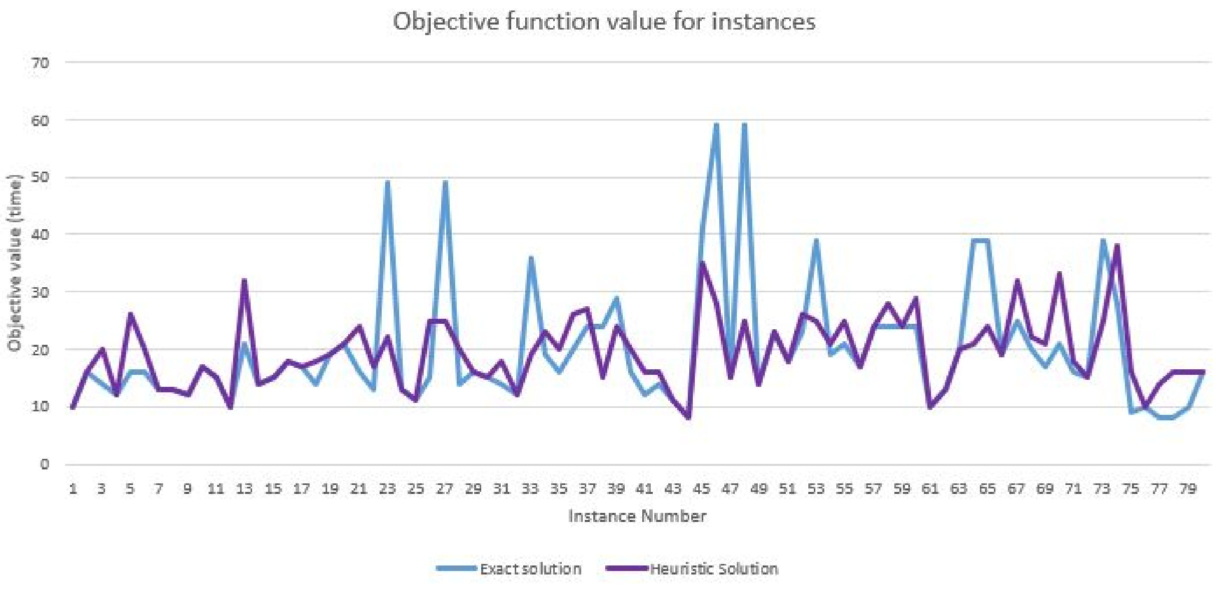
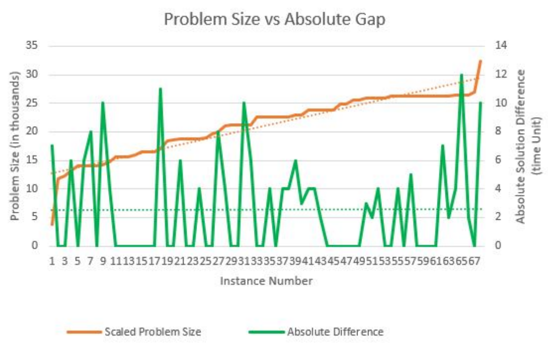


Figure 2.9: Objective function values gained by exact and heuristic approaches

Figure 2.10 illustrates a comparison between the performance of the two approaches in terms of the absolute gap and relative gap. The charts depict the difference only for the 68 instances where an optimal solution was found by the exact approach. Note that a difference of zero is desirable and means that the heuristic was able to find the optimal value. In order to show the instance size in these charts, we defined problem size by multiplying the number of cars, number of track slots, and the time horizon length.



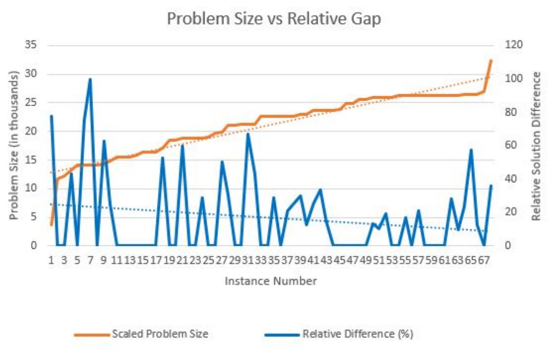


Figure 2.10: Relative performance of the exact and heuristic approach.

The average absolute and relative gap across these 68 instances are 2.6 and 16.9% respectively. Note that, although the relative gap is as high as 100% for some instance, it is rather due to magnitude of objective function than the poor quality of solution. In other words, for instances that have a small objective function (i.e. short time-span), even one unit of time deviation from optimal solution will result in high relative gap. The absolute gap chart depicts that the deviation of the heuristic solution from the optimal values tends to stay the same regardless of the problem size. With that said, It is apparent from these charts that the performance of the heuristic algorithm is not dependent on the instance size and the absolute and relative gap maintain their quality as the problem size grows.

Additionally, table 2.1 shows heuristic algorithm run-time outperforms the exact approach regardless of problem size. Due to this, although the exact approach generates an optimal solution for many small cases, the heuristic approach might be more favorable for practical size instances. In real-world, users solve a multiple train routing problem probably every hour or so, which would require a faster solution time than the exact approach can provide.

## 2.7 Discussion and Conclusion

In this work, we considered the problem of simultaneously moving trains from their initial locations to destination locations in a railyard network. We account for track section and switch node capacity limits allowing at most one train at a time to avoid collision when multiple trains utilize the same track segment. We investigated the problem via an integer programming model and a heuristic approach to minimize the total time required for trains to reach their destinations while accounting for collision avoidance.

The integer programming approach uses a network where track segments are decomposed into a discrete set of slots. This way, the location of each railcar over the railyard network's tracks can be determined at each time step. The mathematical model is also capable of accounting for railcars that must remain attached to each other during the repositioning operations. We also provide an extension to this integer programming model to account for the role of the locomotive. The resulting model would prevent the movement of railcars if there is no locomotive attached to them.

On the other hand, we provide a constructive heuristic combined with the GRASP metaheuristic that utilizes the proposed algorithm in Chapter 1 for the single-train shortest route problem. The heuristic considers multiple possible routes for each train that we desire to reposition. Then, at each iteration, a route is randomly selected from a set of routes that have the least total time. The selected route is fixed for the associated train, and the rest of the trains' routes are adjusted based on the conflicts they might have with the already fixed route.

Both integer programming and the heuristic approach provide collision-free routing options for the multiple-train shortest-route problem. Our approaches consider the geometric restrictions of a railyard network and accounts for the train length when determining the occupancy period of track segments. While the integer programming model is very accurate in positioning the railcars step-by-step, its practical use is limited to small instances only. The problem size grows as the network gets larger since this requires a greater number of slots, which translates into a larger number of constraints. However, the heuristic's performance is independent of the problem size in terms of solution time. This is because the heuristic does not require decomposing the tracks into slots. It only uses the shortest path algorithm over an expanded network followed by an adjustment of route time based on conflicts and delays.

The solution to this problem can serve as a 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.

# 3. TRAIN ASSIGNMENT PLANNING

## 3.1 Introduction

In a freight rail network, a set of geographically dispersed railyard facilities is interconnected via main lines. These main lines traverse hundreds of miles interconnecting railyard facilities, or hubs. These railyard facilities serve as origin and destination terminals for freight trains that carry containers. We will refer to a container of any type generically as a shipment unit, or a unit. Thus, within terminals, units with the same destination hub are assembled to form new outbound trains.

Each unit at a railyard has some known destination, due date, and a time before which it must leave the railyard in order to arrive at its destination on time. When and which units must go into assembly to create an outbound train heading to the associated destination is a decision that must be made with a goal of avoiding any possible delays or late deliveries. When assembling outbound trains, limitations exist on train lengths because, depending on the railyard structure, certain trains might not fit within the railyard tracks where assembly, loading, and unloading of railcars occurs. Depending on the railyard's capacity, the number of trains that may be assembled and depart a facility within a given time interval also serves to constrain the set of available solutions. The ultimate goal is then to assemble units together with common destinations into trains in order to maximize on-time performance (OTP).

At intermodal freight terminals, where containers are transferred to and from trucks and trailers, the assembly and disassembly of trains to create departure schedules is a significant part of the workload. There are often cases where temporarily high unit arrival rates carried from inbound trucks to railyard facilities can cause an unbalanced workload throughout a week. Considering workforce requirements, instead of matching the assembly workload to the arrival rate, facility managers often prefer a balanced schedule of outbound trains that also seeks to meet on-time departure goals. Thus, given a set of units and their destinations and departure intervals, we may seek a smooth train schedule throughout the week that also attempts to ensure that all units leave the railyard within their predetermined time intervals. We call this the Train Assignment Planning (TAP) problem.

Motivated by the need to have a weekly schedule of train departures within a railyard network, we propose approaches for solving the Train Assignment Planning problem. We provide an exact approach that models the problem as a mixed-integer program with different objectives, including maximizing on-time performance (OTP) and maximizing schedule "balance" (or smoothness) within a week.

The remainder of this paper is organized as follows. The following section discusses previous relevant work in the area of train scheduling. Section 3.3 then provides a formal problem definition. Section 3.4 discusses an exact mathematical model for the Train Assignment Planning problem. In Section 3.5, we provide an $\mathcal{NP}$-Hardness proof for this problem under the OTP objective. Finally, in Section 3.6 we propose heuristic algorithms for efficiently determining solutions and provide some insights on computational results in Section 3.7.

## 3.2 Related Literature

The ultimate goal of our problem is to achieve a train schedule (i.e. determining departure times and destinations) by assigning units to each train, while respecting each unit's allowed departure interval. In the following we explain the train scheduling problem, the timetabling problem, and the train unit assignment problem, since each of these attempts to schedule trains given some constraints. Note that the train scheduling problem is defined for freight transportation, whereas the timetabling and train unit assignment problems are primarily applied to passenger transportation.

Ahuja, Cunha, and Şahin [28] provide a comprehensive review of railroad planning problems. They address the train scheduling problem as a large-scale network optimization problem that determines the number of trains, their associated routes and arrival/departure times, and the assignment of blocks of cars to those trains, where a block can travel on multiple trains and several blocks can ride on one train. These decisions are usually made within two stages, first determining train routes and then assigning blocks to the formed trains. In addition to minimizing the number of trains, they consider other objectives such as minimizing the travel distance of cars and minimizing car swaps between trains. In this problem, although car blocks have origin and destination stations which determine which trains they may travel on, they do not have associated shipping time intervals for on-time departures. In our problem, the schedule of the trains is highly dependent on the units' allowed departure intervals.

The train timetabling problem is defined for passenger trains where passengers can make interchanges to reach their destinations. Therefore, train arrival and departure times are determined at the associated stations through which they pass throughout their routes. Timetables (train destinations and departure times) can be generated cyclically, meaning that they are repeated in a pattern, e.g., weekly or aperiodically. Wong, Yuen, Fung, and Leung [29] propose a mixed integer programming model for an aperiodic timetable, considering some operational constraints such as dwell time at stations. They consider the objective of passenger satisfaction in terms of interchange waiting times.

While the main objective of the train timetabling problem is to find a fixed train schedule that satisfies some operational constraints, some work has been done in an attempt to create more robust schedules. Fischetti, Salvagnin, and Zanette [30] introduce approaches to find solutions with increased robustness to handle more operational constraints that may cause delays in the network. In their approach, given an ideal train timetable, an actual train schedule with more robustness is desired with minimum variance from the ideal schedule.

Although the output of the train timetabling problem is a train schedule, it does not deal with the assignment of units to trains. In fact, because timetabling is mainly applied to passenger trains, trains are scheduled based on operational constraints (e.g. terminal capacity) rather than the requirements associated with individual freight units. Thus, the train timetabling does not consider the consequences of departure time on its respective units (i.e. passengers), whereas in our research problem, departure time determines the on time performance of each unit.

The train-unit assignment problem (TUAP) is another context investigated in the literature for passenger railways in which predetermined timetabled trips with associated departure times are considered along with their associated passenger requirements. In addition, trains with given seating capacity and cost are available to fulfill these requirements. The objective is basically to assign a set of trains to trips while satisfying a set of operational constraints. This problem class is sometimes referred to as railway rolling stock planning. Cacchiani, Caprara, and Toth [31] show that the TUAP is $\mathcal{NP}$-Hard and propose a heuristic to address this problem. They allow a maximum of two train units to serve passenger demand for a trip, where train unit types with different capacity and speed are considered. In later work, Cacchiani, Caprara, and Toth [32] address the same problem during a peak period within the planning horizon, when many trips overlap. Note that in the TUAP, trains are called train-units and this differs from the definition of a unit in our problem. We consider units as the containers being assigned to trains, but the TUAP considers units as trains being assigned to predetermined timetabled trips.

To the best of our knowledge, the train unit assignment problem is the closest topic to our research problem that has been addressed in prior literature. However, there are key characteristics that our problem uniquely considers, which makes it distinct from the TUAP. In the TUAP, trips have predetermined departure times. In contrast, our problem requires determining this train departure schedule. Also individual shipment units in our problem have departure intervals, i.e., these units may depart on one of several different trains during the timeline and still make it to their destination on time. In addition, we consider an objective function that balances departures over a time horizon, which differs from the objective function used in the TUAP. Thus, in our problem setting, the existence of unit departure intervals permits shifting units between different trains with a common destination that fall within the allowable departure interval in order to construct a smoother schedule. In the next section we provide a formal definition of our problem.

## 3.3 Problem Definition

This section describes the Train Assignment Planning (TAP) problem in detail. The goal of this problem is to assist railyard operations managers in determining an optimal outbound train schedule. An outbound train schedule consists of a set of trains (and assigned units) and their respective departure hours and destinations, which may repeat cyclically (for example, weekly).

Table 3.1 shows a typical weekly outbound train schedule. Departure hours are determined relative to a fixed point in the week, e.g., Monday at 12:00 am. This means that train #1 in the table, with destination Chicago, will depart at 7:00 am on Monday. In addition to destination and departure times, trains have two other characteristics: the maximum number of units (which is a limit on the number of units that can be loaded on a train), and the minimum number of units (which determines the lowest number of units that should be loaded on a train to make train travel cost effective across the railroad network).

Table 3.1: A Sample Outbound Train Schedule

| **Train #** | **Departure Time (hr)** | **Destination** |
|---|---|---|
| **1** | 7 | Chicago, IL |
| **2** | 17 | Dallas, TX |
| **3** | 51 | Chicago, IL |
| **4** | 79 | New York, NY |
| **5** | 100 | New York, NY |
| **6** | 125 | Phoenix, AZ |

The ultimate goal of the outbound train schedule is to transport units to their destination railyards on time. In other words, we are given a list of units, where we would like for each unit to arrive at its destination by a certain time, i.e., its goal time. If a unit arrives at its destination prior to its goal time, then the unit is regarded as on time. Therefore, each unit has three characteristics: a destination, an on-time departure interval, and a permissible delay departure interval. The on-time interval is the period during which the unit must depart in order to arrive at its destination on time. There is usually a grace period after the goal time such that destination facility will accept the arrival of a unit; however, units that are accepted after their goal time are labeled as late units. The so-called delay departure interval is then the interval during which a unit is allowed to leave the facility and will be accepted at its destination, but will not be considered as having arrived on time. Any time outside the union of the on-time interval and delay interval is prohibited for the assignment of a unit to a train, as the destination facility will not accept the arrival of such units.

To elaborate on this, see Figure 3.1. In Figure 3.1(a), four units are shown along with their on-time departure intervals. For example, the on-time interval for the first unit from the bottom is [2, 9]. Suppose all four units have the same destination. This means that if we create an outbound train schedule with a train at hour 7 and assign all four units to that train, it can transport them to their destination on time (see Figure 3.1(b)). In this example, we achieved 100% on-time performance (OTP), i.e., all units are shipped within their respective on-time departure interval. Now assume that the delay departure interval for the topmost unit is [8, 11]. For an outbound train schedule with a train departing at time 8, we can still assign all four units to this train and have them delivered to the destination. However, the OTP of this plan is 75% because the topmost unit departs within its delay departure interval.

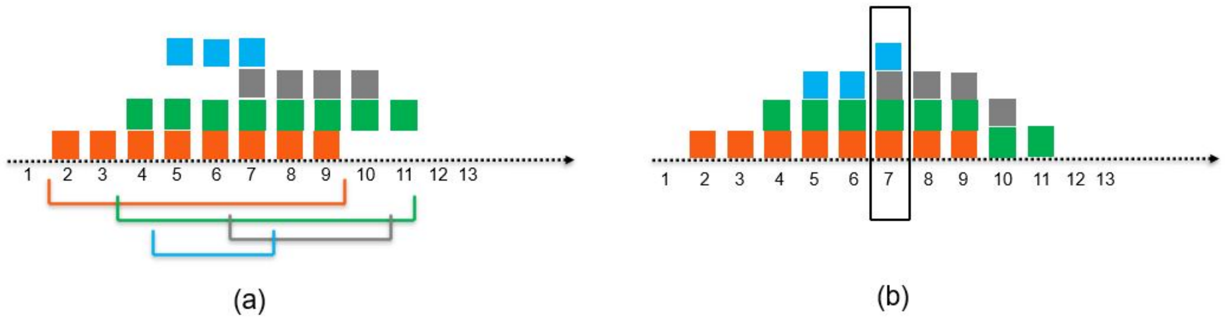


Figure 3.1: Units and their allowed departure interval

Thus, given a set of units, each with a desired departure interval, the TAP creates an outbound train schedule that assembles units with a common destination into trains, where each train has an associated departure time. We consider three different objectives for a TAP problem:

- minimizing the number of trains
- maximizing on-time performance (OTP)
- maximizing smoothness

When considering the minimization of the number of trains, we wish to find an outbound train schedule that meets a certain level of OTP with the least amount of trains. For example, if we require 100% OTP, the result of this objective function provides us the minimum number of trains required for a fully on-time shipment. Any schedule with a smaller number of trains, would necessarily have a lower OTP ratio.

Although a hundred percent OTP is desirable, in reality, a facility's operational constraints limit the number of possible outbound trains on the schedule. These operational constraints arise because of a facility's inadequate resources, such as the number of locomotives available to pull the outbound trains, total workforce, number of cranes, etc. These limitations can clearly impact OTP. Therefore, in the second objective we seek an outbound train schedule that maximizes OTP given a predetermined maximum number of trains (e.g., six outbound trains per week). Note that if the given maximum number of trains is greater than what is achieved using the first objective, the same solution is optimal for the second objective as well.

In the third objective, smoothness (balance) refers to evenly distributing unit loads throughout a week. For this purpose, we define time windows over the course of a week, e.g., daily windows, and we compare the number of units, either on-time or delayed, processed within these time windows. The goal is to balance, as closely as possible, the number of units departing during each time window. In the best case, a schedule would transport all units and have the same workload for each day of the week. Later, we will define a measure of smoothness in detail.

In the following, we will define each objective in more detail. We provide a mixed integer programming formulation associated with each of these optimization problems. Because, as we later show, the resulting optimization problems are $\mathcal{NP}$-Hard, we will also propose heuristic approaches to ensure an ability to find high-quality solutions in acceptable computing time.

## 3.4 TAP Mathematical Model

As explained before, in Train Assignment Planning we seek an optimal schedule of trains and their unit assignments for a given objective. Each unit has a destination, on-time departure interval, and permitted delay interval. For convenience and practical application, we use a weekly planning horizon broken down into hourly intervals, assuming at most one train departure per hour. In the following, we define the variables and parameters required to model this problem with three different objectives, and then we provide a mathematical formulation, followed by the model description.

### 3.4.1 TAP: Minimizing Number of Trains

We first consider an objective that aims to minimize the number of trains required to meet a prescribed level of on-time performance (in terms of percentage of on-time departures) while ensuring that all units are assigned to departing trains.

To define a problem instance, we first define the set of units $U$, with the total number of units denoted as $|U|$. For each unit $u \in U$ let $OI_u$ be its on-time interval; i.e., the set of hours for which unit $u$ can be assigned to a departing train and arrive at its destination on-time. Similarly, let $DI_u$ be its delay interval, i.e., the set of hours for which unit $u$ may be assigned to a departing train and arrive at its destination with a delay. Let $D$ denote the set of destinations. Therefore, we can define $U_d$ as the set of units with destination $d$ where $d \in D$. To avoid constructing excessively under-weighted or over-weighted trains, let $m$ and $n$ be the minimum and maximum allowed number of units on a train. We define three sets of binary variables to track unit assignments to trains and train assignments to departure times:

- $X_u^t = 1$ if unit $u$ is loaded on a train at $t$, and 0 otherwise, for $t \in OI_u$;
- $Z_u^t = 1$ if unit $u$ is loaded on a train at $t$, and 0 otherwise, for $t \in DI_u$;
- $Y_d^t = 1$ if a train departs to destination $d$ at time $t$, and 0 otherwise, for $t = 1, \ldots, T$;

Note that variables $X$ and $Z$ could be aggregated into one set of variables with domain $t \in \{OI_u \cup DI_u\}$. We use separate notation ($X$ and $Z$) to more easily distinguish between variables associated with on-time and delayed units. Using these definitions, we next formulate our integer programming model. Recall that the objective is to minimize the number of trains required to transport all units while meeting a minimum value of $OTP$ ratio.

$$\text{Minimize} \qquad \sum_{t=1}^{T} \sum_{d \in D} Y_d^t \tag{3.1}$$

$$\text{s.t.} \quad \sum_{t \in OI_u} X_u^t + \sum_{t \in DI_u} Z_u^t = 1 \qquad u \in U, \tag{3.2}$$

$$\sum_{d \in D} Y_d^t \leq 1 \qquad t = 1, \ldots, T, \tag{3.3}$$

$$\sum_{u \in U_d} (X_u^t + Z_u^t) \leq nY_d^t \qquad d \in D, t = 1, \ldots, T, \tag{3.4}$$

$$\sum_{u \in U_d} (X_u^t + Z_u^t) \geq mY_d^t \qquad d \in D, t = 1, \ldots, T, \tag{3.5}$$

$$\sum_{u \in U} \sum_{t \in OI_u} X_u^t \geq OTP|U| \quad ,$$

$$X_u^t \in \{0, 1\} \qquad u \in U, t \in OI_u, \tag{3.6}$$

$$Z_u^t \in \{0, 1\} \qquad u \in U, t \in DI_u, \tag{3.7}$$

$$Y_d^t \in \{0, 1\} \qquad d \in D, t = 1, \ldots, T. \tag{3.8}$$

The objective function (3.1) minimizes the number of trains required to transport all units. Constraint (3.2) makes sure all units are transported while Constraint (3.3) allows at most one train departure per hour. Constraints (3.4-3.5) require that the number of loaded units on a train stays within the allowed range. Constraint (3.6) enforces the given minimum OTP ratio. Finally constraints (3.6-3.8) define the binary variable restrictions.

This model determines an assignment of destinations and departure times to trains over the given time horizon, along with the associated units. While this integer programming formulation is useful for determining the number of trains required to fulfill the outbound demands, as we mentioned previously many practical cases have restrictions on total number of train departures. In the following, we will address another objective function that is beneficial in constructing a train schedule when the number of train departures are limited.

### 3.4.2 TAP: Maximizing On-time Performance

The previous objective minimized the number of trains subject to a minimum level of on-time performance. In this section, we consider an objective that instead maximizes on-time performance, while ensuring that all units are assigned to departing trains. In this formulation, the maximum number of trains that can depart the facility during the time horizon is limited.

By letting $R$ denote an upper bound on the number of trains permitted to depart during the time horizon $T$, and using the same definition of parameters and variables as in the previous section, we formulate the problem as follows:

$$\text{Maximize} \qquad \sum_{u \in U} \sum_{t \in OI_u} X_u^t \tag{3.9}$$

$$\text{s.t.} \quad \sum_{t \in OI_u} X_u^t + \sum_{t \in DI_u} Z_u^t = 1 \qquad u \in U, \tag{3.10}$$

$$\sum_{d \in D} Y_d^t \leq 1 \qquad t = 1, \ldots, T, \tag{3.11}$$

$$\sum_{u \in U_d} (X_u^t + Z_u^t) \leq nY_d^t \qquad d \in D, t = 1, \ldots, T, \tag{3.12}$$

$$\sum_{u \in U_d} (X_u^t + Z_u^t) \geq mY_d^t \qquad d \in D, t = 1, \ldots, T, \tag{3.13}$$

$$\sum_{d \in D} \sum_{t=1}^{T} Y_d^t \leq R \qquad , \tag{3.14}$$

$$X_u^t \in \{0, 1\} \qquad u \in U, t \in OI_u, \tag{3.15}$$

$$Z_u^t \in \{0, 1\} \qquad u \in U, t \in DI_u, \tag{3.16}$$

$$Y_d^t \in \{0, 1\} \qquad d \in D, t = 1, \ldots, T. \tag{3.17}$$

The objective function (3.9) maximizes the number of units that can be delivered on time. Constraint (3.10) makes sure all units are transported while Constraint (3.11) allows at most one train departure per hour. Constraints (3.12-3.13) require that number of loaded units on a train stays within the allowed range. Constraint (3.14) ensures that the number of trains does not exceed the maximum number of available trains. Finally, Constraints (3.15-3.17) impose binary restrictions on the variables.

The objective function in this model captures the total number of on-time units. Dividing this number by the total number of units gives the solution's OTP ratio. Note that if OTP is not satisfactory, one could increase the capacity of the railyard in terms of the number of available outbound trains ($R$) for an improved level of OTP.

Although on-time performance is a critical factor in freight transportation, many factors, such as workforce availability, may limit the workload throughout the planning horizon. Thus, the output of the models we have discussed thus far may provide a train schedule that is highly congested at some hours due to a high number of scheduled departures, with low levels of activity during other hours. In the following, we propose another objective function that seeks a schedule with a balanced workload over the time horizon.

### 3.4.3 TAP: Maximizing Schedule Smoothness

Maximizing efficiency by minimizing the number of outbound trains and maximizing on-time delivery are two important criteria in constructing a train schedule. However, having a balanced workload throughout the planning horizon is also essential for effectively managing operations. We next propose another model with an objective that to maximize schedule smoothness throughout the week.

To define a smoothness measure, first we define a set of time windows. Because a week is 168 hours, we consider 168 time windows, each containing 24 hours, meaning that each time window differs from the next time window by one hour. Let $W$ be the set of time windows (for each of the 168 time windows). We denote $W_\tau$ as the set of 24 consecutive hours starting at $\tau$, where $\tau \in \{1 \dots T\}$. See figure 3.2 for an illustration of these time windows. Note that we take a cyclic approach. For example, a time window that starts at hour 166 rolls over into the next week, and therefore includes the initial hours of the following week.

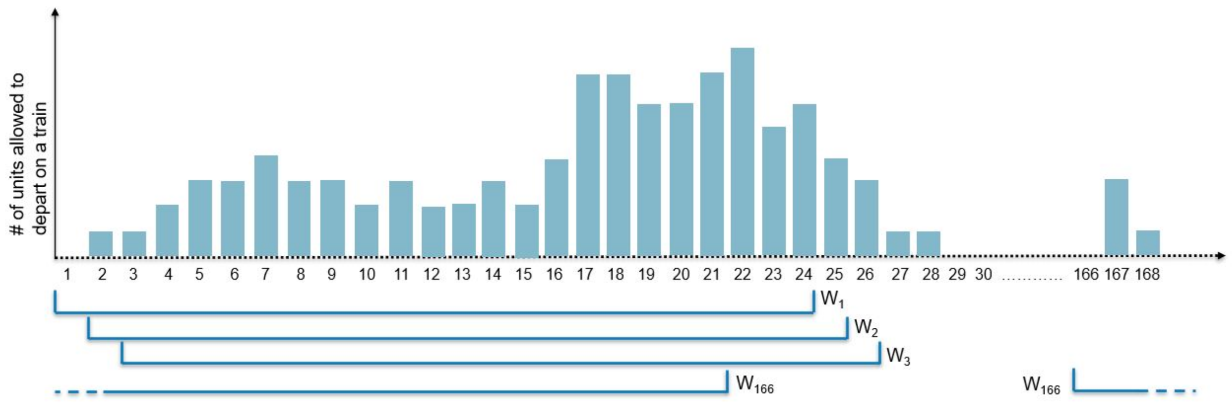


Figure 3.2: Time window demonstration

One way to evaluate a schedule's smoothness is to compare the number of unit assignments within each time window. We want to minimize the difference between the maximum and minimum load across time windows to ensure a smooth schedule. To do this, we first define the workload of a time window as the number of units that depart within the time window. For a given schedule, let $L_w$ denote the minimum workload across all time windows. Also let $L_{ub} =$ (Total # of Units)(Length of Time Windows)/(Total # of Time Windows). $L_{ub}$ is an upper bound on $L_w$. Next, for a given schedule, we define the smoothness factor as $SF = L_w/L_{ub}$, where $SF \leq 1$ and is increasing in the smoothness of the schedule. We wish to maximize the smoothness factor, and since $L_{ub}$ is a constant value for each problem instance, this is equivalent to maximizing $L_w$.

Thus, in addition to $X_u^t$, $Z_u^t$, and $Y_d^t$, we now define the variable $L_w$ as the minimum workload among all windows. The rest of the notation remains the same as in the previous section, and we formulate the problem as follows:

$$\text{Maximize} \qquad L_w \tag{3.18}$$

$$\text{s.t.} \quad \sum_{t \in OI_u} X_u^t + \sum_{t \in DI_u} Z_u^t = 1 \qquad u \in U, \tag{3.19}$$

$$\sum_{d \in D} Y_d^t \leq 1 \qquad t = 1, \ldots, T, \tag{3.20}$$

$$\sum_{u \in U_d} (X_u^t + Z_u^t) \leq nY_d^t \qquad d \in D, t = 1, \ldots, T, \tag{3.21}$$

$$\sum_{u \in U_d} (X_u^t + Z_u^t) \geq mY_d^t \qquad d \in D, t = 1, \ldots, T, \tag{3.22}$$

$$\sum_{d \in D} \sum_{t=1}^{T} Y_d^t \leq R \qquad , \tag{3.23}$$

$$\sum_{u \in U} \sum_{t \in OI_u} X_u^t \geq OTP|U| \qquad , \tag{3.24}$$

$$\sum_{u \in U} \sum_{t \in W_\tau} (X_u^t + Z_u^t) \geq L_w \qquad \tau = 1, \ldots, T, \tag{3.25}$$

$$X_u^t \in \{0, 1\} \qquad u \in U, t \in OI_u, \tag{3.26}$$

$$Z_u^t \in \{0, 1\} \qquad u \in U, t \in DI_u, \tag{3.27}$$

$$Y_d^t \in \{0, 1\} \qquad d \in D, t = 1, \ldots, T, \tag{3.28}$$

$$L_w \geq 0. \tag{3.29}$$

The objective function (3.18) maximizes the minimum workload across all windows. Constraint (3.19) makes sure all units are delivered, while constraint (3.20) allows at most one train departure per hour. Constraints (3.21-3.22) require that number of loaded units on a train stays within the allowed range. Constraint (3.23) ensures that the number of trains does not exceed the maximum number of available trains. Constraint (3.24) enforces the given minimum OTP ratio. Constraint (3.25), along with the objective function, determines the minimum workload across all time windows. Finally, constraints (3.26)–(3.29) define the binary and continuous variable restrictions.

This model's solution leads to a smoother schedule compared to the previous two models; however, there is a trade-off between smoothness and on-time performance. In other words, if one reduces the OTP threshold requirement, a smoother schedule can likely be achieved because units have a broader range of hours to which they can be assigned to train departures, which results in trains having a wider range of departure options over the planning horizon.

## 3.5 Complexity

In this section, we investigate the complexity of previously discussed problems. We show that a special case of our problem can be cast as a version of the problem addressed by Chang, Erlebach, Gailis, and Khuller [33], who explored different versions of the broadcast scheduling problem. One of the problems they study is broadcast scheduling with deadlines. In this problem, information is disseminated in response to clients' requests in the form of pages that are broadcast throughout a network. Clients may request any subset of pages at different time points. Every request has a deadline by which it should be satisfied by a broadcast, and the goal is to determine a broadcast schedule that maximizes the number of page requests that are met before their deadlines. This is equivalent to our objective of maximizing on-time performance. They provide a reduction from the vertex cover problem of size $k$ to prove that broadcast scheduling with deadlines is $\mathcal{NP}$-Hard. A more detailed discussion of this reduction is provided in Appendix A.

Chang, et al. [33] discretize the time horizon, where at each time point at most one page can be broadcast. A request for page $p$ at time $t$ has a due date $D_t^p$. In our problem, destinations correspond to pages, which means that at each time point at most one destination can be served (with a departure, which is equivalent to a page broadcast). There are $n$ pages, $P = \{1, 2, \ldots, n\}$ and there may be multiple requests for a page in overlapping time windows. Sending out a broadcast at any time within the overlapping time windows satisfies all requests for that page. For our problem, a destination exists corresponding to each page, and each page request would correspond to a unit shipment request for the corresponding destination within the required time window. Therefore any broadcast of page $p$ at time $t$ corresponds to a train departing at time $t$ for destination $p$, and this can happen at most once per time point. In their proof, any given broadcast covers at most three requests for the same page, which translates into each train carrying at most three units to the same destination.

See Figure 3.3 for a more transparent demonstration of request time window length and number of requests for a broadcast in two cases. Different colors are associated with different pages/destinations in this figure, whereas each presence of a colored cube demonstrates a request/unit. In offline scheduling of such requests, the broadcast/train schedule is proposed with colored arrows for different pages/destinations. In case $(a)$, scheduling a broadcast/train at time 4 is enough to handle the three requests/trains, whereas in case $(b)$ we would need two broadcasts/trains.

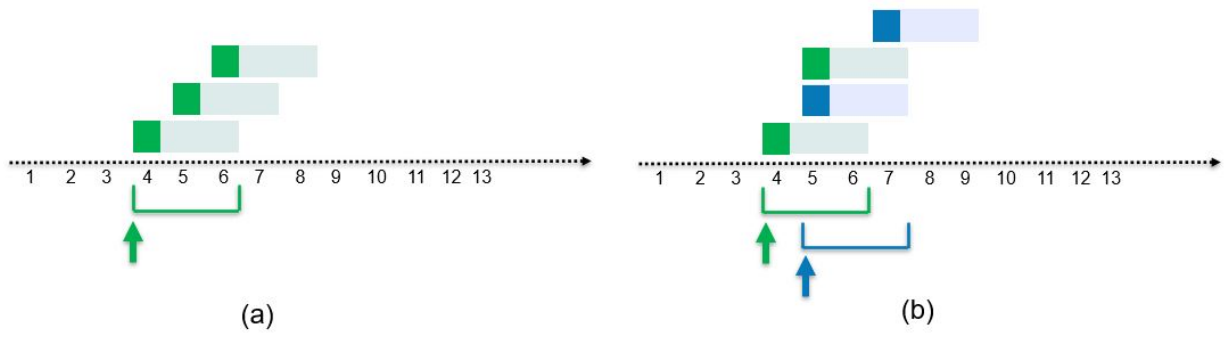


Figure 3.3: Broadcast scheduling with deadlines

Thus, if we consider the special case of our problem with no delay intervals while setting $m = 0$ and $n = 3$, then this special case is equivalent to the broadcast scheduling problem instance that was shown to be $\mathcal{NP}$-Hard by Chang, et al. [33], which implies that the Train Assignment Planning problem with an objective of maximizing on-time performance is $\mathcal{NP}$-Hard. Observe that a special case of the TAP arises in which the set of possible delay intervals is empty (equivalently, in which the minimum OTP is 100%). The proof implies that the feasibility problem is $\mathcal{NP}$-Complete, i.e., the recognition problem that asks whether a solution exists such that all units can be scheduled within their time windows. Thus, a special case exists under all three of the objective functions we consider such that determining whether or not a feasible solution exists is $\mathcal{NP}$-Complete. As a result, the resulting optimization problem will be $\mathcal{NP}$-Hard for any objective (because determining whether a feasible solution exists is $\mathcal{NP}$-Complete in general).

## 3.6 TAP Solution Approach

Section 3.4 provided mathematical models for obtaining exact solutions for various versions of the Train Assignment Planning problem. However, the problem's complexity (discussed in the previous section) and the scale of the problem for practical size instances will often make its use in practice difficult due to the time required to find an optimal solution. The problem size increases dramatically with the number of units, the length of the departure intervals, and the length of the planning horizon. In this section, we propose methods that enable obtaining solutions for real-world problem sizes in faster computing time that guarantee solution time efficiency. Section 3.6.1 first provides a constructive heuristic for maximizing on-time performance, while Section 3.6.2 proposes a Lagrangian relaxation based heuristic for this same objective. Section 3.6.3 then provides a heuristic that modifies a given proposed schedule with a goal of maximizing schedule smoothness.

### 3.6.1 Maximizing On-time Performance: OTP Heuristic

The input to the TAP Heuristic with the objective of maximizing on-time performance consists of a list of destinations, a set of units, the destination of each unit, the on-time departure interval for each unit, and the allowable delay departure interval for each unit. We use this information along with the maximum number of available trains $R$, and the train capacity (i.e., the maximum number of units that can be assigned to a train) to construct a schedule for transporting all units. We will refer to this approach as the constructive OTP heuristic.

This constructive OTP heuristic starts by creating a scoring mechanism for every potential train departure. To do this, we first consider a candidate train for each *Hour-Destination* combination. For each of these candidate trains, we establish an on-time units list, i.e., a list of units that can leave on the train and arrive on-time. We consider the number of elements on this list as the score for the given *Hour-Destination* candidate train. This means that for a planning horizon of a week, and with five destinations on our list, we will have 840 (= 168*5) candidate trains, each of which has a score based on the number of units that can be transported on-time. Next, we sort these candidate trains in non-ascending order and select the one with the highest score.

The next step is to assign units to the selected *Hour-Destination* train. We assign units with the associated destination such that the given hour falls within the unit's on-time departure interval, in non-decreasing order of on-time interval width until we reach the train's capacity. The idea is that units with a wider range of on-time intervals have a better chance of being assigned to an alternate train with the same destination. After filling up the train's capacity or assigning all units in the associated on-time unit list, we update the on-time units list for the remainder of the candidate trains for the hours that have not yet had a departing train, and repeat the process. After selecting and allocating on-time units to $R$ candidate trains, we fill the remainder of any available capacity with delayed units. Suppose the total number of selected trains is less than $R$, and after fully allocating delayed items, unassigned units remain. In this case, the constructive OTP heuristic may add extra trains to accommodate the unassigned and delayed units. Algorithm 4 represents the pseudocode for our proposed heuristic.

Note that a feasible solution does not exist if the number of units exceeds all available trains' total capacity. Also, observe that the number of units per destination divided by the train capacity generates a logical lower bound on the number of trains needed for that specific destination. For example, suppose 100 and 300 units are required for destinations $a$ and $b$, respectively, and a train's capacity is set to a maximum of 250 units. A total number of 400 units with the same destination would require two trains, but we would need at least three trains to transport the 400 units in this case because of the two different destinations.

The constructive OTP heuristic we have described provides a feasible solution for any problem in which a feasible solution exists and where there is no minimum unit load for trains. If trains have minimum unit load however, the heuristic might not provide a feasible solution since the minimum load requirement is not checked at any point. To address this drawback, a post-processing procedure is proposed that takes in the algorithm's proposed solution and attempts to adjust the solution

**Algorithm 4** Pseudocode for maximizing schedule's OTP (Constructive OTP Heuristic)

```
Total number of available trains for TSP $\rightarrow TotalAvailableTrains$
List of all units $\rightarrow UnitList$
for each $Hour - Destination$ combination do
    Define a candidate train and add it to $CandidateList$
    Find on-time units that can go on this candidate train $\rightarrow train\_OntimeUnitList\_P$
    Sort units in $train\_OntimeUnitList\_P$ based on non-descending order of unit on-time interval length
    Find delayed units that can go on this candidate train $\rightarrow train\_DelayUnitList\_P$
    Sort units in $train\_DelayUnitList\_P$ based on non-descending order of unit delay interval length
    $train\_OntimeUnitListlength \rightarrow train\_Score$
end for
Create a list for final trains $\rightarrow FinalScheduleList$
$numberOfSelectedTrains = 0$
while TRUE do
    Sort $CandidateList$ based on $train\_Score$ in non-ascending order
    if $numberOfSelectedTrains < TotalAvailableTrains$ then
        Select candidate train with highest $train\_Score$ among unselected hours of horizon
        if $train\_Score > 0$ then
            Add the candidate train to $FinalScheduleList$
            $numberOfSelectedTrains \leftarrow numberOfSelectedTrains + 1$
            Assign units from $train\_OnTimeUnitList\_P$, as long as train's capacity is not violated $\rightarrow train\_OnTimeUnits$
            Update $train\_OntimeUnitList\_P$ for all candidate trains by removing assigned units
            Update $train\_DelayUnitList\_P$ for all candidate trains by removing assigned units
            Update $train\_Score$ for all candidate trains by removing assigned units
            Update $UnitList$ by removing assigned units
        else
            break
        end if
    else
        break
    end if
end while
```

**Algorithm 4** (continued) Pseudocode for maximizing schedule's OTP (Constructive OTP Heuristic)

```
Sort $FinalScheduleList$ based on $train\_OnTimeUnits$ size in non-descending order
for each train in $FinalScheduleList$ do
    if train has excess capacity then
        Assign units from $train\_DelayUnitList\_P$ to the train, as long as train's capacity is not violated
                                                                        $\rightarrow train\_DelayUnits$
        Update $train\_DelayUnitList\_P$ for all candidate trains by removing assigned units;
        Update $UnitList$ by removing assigned units;
    end if
end for
while ($UnitList$ is not empty) & ($FinalScheduleList$ length $<$ $TotalAvailableTrains$) do
    Select candidate train with largest $train\_DelayUnitList\_P$ among unselected hours of horizon
    Add the candidate train to $FinalScheduleList$
    Assign units from $train\_DelayUnitList\_P$ to the candidate train as long as train's capacity is not violated
                                                                        $\rightarrow train\_DelayUnits$
    Update $UnitList$, and $train\_DelayUnitList$ by removing assigned units
end while
```

(if needed) to account for the minimum load constraint. Algorithm 5 explains this procedure. This approach finds any trains that violate the minimum load constraint, and assigns units to them until the number of assigned units is within the allowed range. The units assigned are selected from the trains with maximum loads and where the destination matches and the unit interval covers both train departure times, giving priority to delayed units.

**Algorithm 5** Post-processing procedure pseudocode

```
Sort $FinalScheduleList$ based on $train\_size$ in non-descending order
for each train $selectedTrain$ in $FinalScheduleList$ do
    if $selectedTrain\_size$ is greater than minimum allowed load then
        break
    else
        while $selectedTrain\_size$ is less than minimum allowed load do
            Select the train from $FinalScheduleList$ with highest $train\_size$ and matching destination
                $\rightarrow fillingTrain$
            Assign a unit from $fillingTrain$ that can depart on $selectedTrain$ with respect to unit's
            departure interval
        end while
    end if
end for
```

One challenge in applying the proposed constructive OTP heuristic is that delayed units are not accounted for in the scoring mechanism. One could consider accounting for delayed units in the scoring mechanism by adding a term with a weighted coefficient. However, this would potentially degrade the solution in terms of the objective of maximizing OTP.

### 3.6.2 Maximizing On-time Performance: OTP Lagrangian Relaxation

We next provide a Lagrangian relaxation approach with an on-time performance maximization objective. Our goal, in addition to providing additional candidate heuristic solutions, is to obtain quality upper bounds in order to gauge the performance of the heuristic approach introduced in the previous section. In doing so, we relax the assignment constraint, which ensures that all units are transported on some train (Constraint 3.10).

To simplify the Lagrangian model exposition and the approach we take, we first provide a new representation of the model. If we have defined $OI_u$ as the on-time interval of periods to which we can assign unit $u$, and $DI_u$ as set of feasible delay periods, then we only need to define $X_t^u$ variables for the assignment of unit $u$ to a departure time. Let $FI_u$ denote the set of feasible departure periods for unit $u$, where $FI_u = OI_u \cup DIu$. The mathematical formulation for maximizing OTP can then be reformulated as

$$\text{Maximize} \quad \sum_{u \in U} \sum_{t \in OI_u} X_u^t \tag{3.30}$$

$$\text{s.t.} \quad \sum_{t \in FI_u} X_u^t = 1 \qquad u \in U, \tag{3.31}$$

$$\sum_{d \in D} Y_d^t \leq 1 \qquad t = 1, \ldots, T, \tag{3.32}$$

$$mY_d^t \leq \sum_{u \in U_d:\, t \in FI_u} X_u^t \leq nY_d^t \quad d \in D, t = 1, \ldots, T, \tag{3.33}$$

$$\sum_{d \in D} \sum_{t=1}^{T} Y_d^t \leq R \qquad ,$$

$$X_u^t \in \{0, 1\} \qquad u \in U, t \in OI_u, \tag{3.34}$$

$$Y_d^t \in \{0, 1\} \qquad d \in D, t = 1, \ldots, T. \tag{3.35}$$

Consider $\lambda_u, u \in U$ as the set of Lagrangian multipliers associated with constraint 3.31, which are unrestricted in sign, and let $\lambda$ denote the $|U|$-vector of $\lambda_u$ values. The associated Lagrangian relaxation model can be written as:

$$\text{Maximize} \quad \sum_{u \in U} \left[ \sum_{t \in OI_u} (\lambda_u + 1) X_u^t + \sum_{t \in DI_u} \lambda_u X_u^t \right] - \sum_{u \in U} \lambda_u \tag{3.36}$$

$$\text{s.t.} \qquad \sum_{d \in D} Y_d^t \leq 1 \qquad t = 1, \ldots, T, \tag{3.37}$$

$$mY_d^t \leq \sum_{u \in U_d :\, t \in FI_u} X_u^t \leq nY_d^t \qquad d \in D, t = 1, \ldots, T, \tag{3.38}$$

$$\sum_{d \in D} \sum_{t=1}^{T} Y_d^t \leq R \qquad , \tag{3.39}$$

$$X_u^t \in \{0, 1\} \qquad u \in U, t \in OI_u, \tag{3.40}$$

$$Y_d^t \in \{0, 1\} \qquad d \in D, t = 1, \ldots, T. \tag{3.41}$$

Suppose we temporarily ignore Constraint (3.39). For a fixed set of values of the Lagrangian multipliers ($\lambda$), the remaining subproblem decomposes by time period. In other words, for each value of the time step $t$, we have a separate optimization problem for which we will provide an efficient solution method. Let us define $\tilde{\lambda_u}(t)$ as the coefficient of the variable in the objective function for time step $t$. We will have:

$$\tilde{\lambda_u}(t) = \begin{cases} \lambda_u + 1, & t \in OI_u \\ \lambda_u, & t \in DI_u \\ 0, & t \notin OI_u \cup DI_u \end{cases} \tag{3.42}$$

The subproblem associated with time period $t$ can be written as follows:

$$\text{Maximize} \qquad \sum_{u \in U} \tilde{\lambda_u}(t) X_u^t \tag{3.43}$$

$$\text{s.t.} \qquad \sum_{d \in D} Y_d^t \leq 1 \qquad , \tag{3.44}$$

$$mY_d^t \leq \sum_{u \in U_d:\, t \in FI_u} X_u^t \leq nY_d^t \quad d \in D, \tag{3.45}$$

$$X_u^t \in \{0,1\} \qquad u \in U, \tag{3.46}$$

$$Y_d^t \in \{0,1\} \qquad d \in D. \tag{3.47}$$

This subproblem can be solved by considering the maximum value of the objective when $Y_d^t = 1$ for each $d \in D$. Suppose, therefore, that we set $Y_{d'}^t = 1$ (and $Y_d^t = 0$ for $d \neq d'$), and that we sort the set of $X_u^t$ variables for each $u \in U_{d'}: t \in FI_u$ in nonincreasing order of $\tilde{\lambda_u}(t)$ values. Let $\bar{u}(d')$ denote the smallest index value among $u \in U_{d'}: t \in FI_u$ such that $\tilde{\lambda_u}(t) \leq 0$. If $\bar{u}(d') \leq m$, then the maximum objective function value when $Y_{d'}^t = 1$ is achieved by setting $X_u^t = 1$ for $u = 1, \ldots, m$. If $\bar{u}(d') \geq n$, then the maximum objective function value when $Y_{d'}^t = 1$ is achieved by setting $X_u^t = 1$ for $u = 1, \ldots, n$. Otherwise, the objective function is maximized by setting $X_u^t = 1$ for $u = 1, \ldots, \bar{u}(d') - 1$. We can therefore easily compute the maximum objective function value when $Y_{d'}^t = 1$ for each value of $d' \in D$, which we denote by $Z_\lambda(d', t)$. Let $d_\lambda^*(t) = \arg\max_{d' \in D} Z_\lambda(d', t)$ and let $Z_\lambda^*(t) = Z_\lambda(d_\lambda^*(t), t)$. If $Z_\lambda^*(t) > 0$, then we set $Y_{d^*(t)}^t = 1$, and set the corresponding $X_u^t$ values equal to 1 that led to the maximum objective value of $Z_\lambda^*(t)$. Otherwise, we set all $Y_d^t$ and $X_u^t$ values and $Z_\lambda^*(t)$ to zero.

Suppose we have solved the period-subproblem for each period, and recall that we ignored Constraint (3.39). Solving the problem with this constraint imposed simply requires choosing the $R$ period subproblems with the highest objective function values. Suppose we re-sort the period indices in nonincreasing order of $Z_\lambda^*(t)$, and let $\bar{t}$ index this re-sorted set of period indices.

Then the optimal Lagrangian solution retains the solutions with the $R$ highest period subproblem objective function values, and sets the $Y_d^t$ and $X_u^t$ values for the remaining subproblems equal to zero (if they are not already zero). The optimal Lagrangian objective in this case then equals $Z_{LR}(\lambda) = \sum_{\bar{t}=1}^{R} Z_\lambda^*(\bar{t}) - \sum_{u \in U} \lambda_u$. The Lagrangian dual problem can then be written as:

$$Z_{LD} = \min \quad Z_{LR}(\lambda), \tag{3.48}$$

$$\text{s.t.} \quad \lambda \in \mathbb{R}_{|U|}. \tag{3.49}$$

At each iteration of the Lagrangian relaxation (for each fixed $\lambda$), we obtain an upper bound on the optimal solution value of the original problem, and $Z_{LD}$ minimizes this upper bound among all $\lambda$. At a given iteration, the Lagrangian relaxation solution may violate the relaxed constraints. In particular, for any given unit $u$, we may assign the unit to multiple trains, or we may not have assigned the unit to any train. We will, therefore, need a heuristic solution approach to resolve these infeasibilities in order to create a feasible solution (and corresponding lower bound) at each iteration if possible. If a unit has been assigned to multiple trains, and we can remove its assignment from all but one of the trains without destroying the lower bound of $m$ units for each train, then this is easy to resolve. If a unit $u$ has not been assigned to any train, then we attempt to assign it to some train with a departure period $t \in FI_u$ without violating the upper bound of $n$ units per train (note that we may not be able to guarantee feasibility of this heuristic at each iteration, depending on how constraining the values of $m$ and $n$ are).

One of the reasons to perform a Lagrangian relaxation approach for maximizing OTP is to evaluate the performance of the heuristic approach explained in the previous section. Therefore, the Lagrangian solution serves as an upper bound for our heuristic approach, and the optimal solution as well. In other words, the optimal solution is bounded from below by the heuristic solution and from above by the Lagrangian dual solution. We will later discuss computational results from the application of the Lagrangian relaxation procedure in Section 3.7.

### 3.6.3 Maximizing Schedule Smoothness: Smoothness Heuristic

In this section, we propose a constructive heuristic for the TAP problem with the objective of maximizing the schedule's smoothness. This heuristic takes a Train Assignment Planning solution as input and create an unit to train assignment with respect to balancing workload. The input TAP solution can be achieved, for example, by applying a heuristic method for maximizing on-time performance. In other words, we can solve the problem by applying one of the heuristic solution methods proposed in the previous subsections, and using the resulting schedule as the input for the heuristic approach for maximizing smoothness that we next discuss.

Our heuristic first takes in an initial TAP schedule and considers the train departure times (and their destinations) as fixed. We then de-assign all of the units and consider their re-assignment to one of the (at most $R$) train departures. In order to spread the unit volume throughout the train schedule, we assign one unit at a time to the trains in the given TAP solution. To prioritize on-time performance, for each train $r$ we create a list that contains all of the units that can leave on train $r$ and arrive on-time. We sort the elements of this list in non-decreasing order of on-time interval width, and assign the first unit to depart on train $r$. In case no such unit is found, we apply the same procedure for units that can leave on train $r$ and arrive with a delay, and assign the unit with the shortest delay interval width. The process continues assigning one unit at a time to the trains in the given TAP solution until all units are assigned.

The given pseudocode in Algorithm 6 represents the general steps of this heuristic. Note that in this heuristic, the goal is to construct a unit to train assignment that maximize the smoothness for a given train schedule. As we discussed before, there is a trade-off between smoothness and on-time performance; nevertheless, our heuristic prioritizes on-time units over delayed ones when assigning units to trains.

The quality of this heuristic will depend in part on the quality of the TAP solution, which is fed to it as input. We propose using the TAP solution obtained by maximizing OTP to indirectly incorporate the importance of on-time performance in the heuristic for maximizing smoothness. However, one can use any feasible TAP solution as input to this heuristic. In fact, some railyards

**Algorithm 6** Pseudocode for maximizing schedule's smoothness

```
Take in a TAP solution → $givenTAP$
List of all units → $UnitList$
for each train in $givenTSP$ do
    Define a sorted list of on-time unit list, in non-descending on-time interval length, that can go on that train → $train\_OnTimeUnitList$
    Define a sorted list of delay unit list, in non-descending delay interval length, that can go on that train → $train\_DelayUnitList$
end for
while $UnitList$ is not empty do
    Pick next train in the $givenTSP$
    if $train\_OnTimeUnitList$ is not empty then
        Assign the first unit in the list
    else if $train\_DelayUnitList$ is not empty then
        Assign the first unit in the list
    end if
    Update $train\_OnTimeUnitList$ and $train\_DelayUnitList$ for all trains by removing the assigned unit;
    Update $UnitList$ by removing the assigned unit;
end while
```

may be bound to a fixed train departure schedule, or we can generate different TAP solutions as input to the heuristic and compare the smoothing heuristic results. Meta-heuristics such as genetic or neighborhood search algorithms can also be used to develop different feasible train schedules. At each iteration in these algorithms, our smoothness heuristic can be used to calculate the fitness function for each solution.

## 3.7 Computational Results

We performed a set of computational experiments based on problems of practical size. Our results show that the constructive OTP heuristic approach outperforms the exact optimization models in terms of solution time. In the following, we will provide a discussion on computational results and models' performance.

We consider a week as the planning horizon, using hourly time buckets, for a total horizon length of 168 periods (a full week is 7 days with 24 hours each). We created instances with 4000 and 5000 shipping units as small-size instances, 6000 and 7000 units as medium-size instances, 8000 and 9000 units as large-size instances, and randomly assigned a destination and departure interval to each unit. The number of destinations was set to either 4 or 8 in order to consider how an increase in the number of destinations affects solution quality. We considered relatively low and high values of the maximum number of trains in order to determine whether solution quality is affected by the number of available trains. In particular, we considered 30 or 40 trains for small instances, 40 or 50 trains for medium-size instances, and 50 or 60 trains for large instances. Note that a larger number of trains is required as the number of shipment units increases in order to ensure that a feasible schedule exists. Overall, we created 24 combinations of the number of shipment units, the maximum number of trains, and the number of destinations. For each of these combinations, we created ten feasible problem instances, resulting a total of 240 individual problem instances. Table 3.2 summarizes the optimality gap results for these instances. Note that the column labeled "Exact" provides optimality gap information based on the best solution that was found by CPLEX within 15 minutes of computing time. The subcolumn labeled "Avg Gap (CPLEX)" shows the optimality gap for this solution provided by CPLEX, while the subcolumn labeled "Avg Gap (LR)" reports the optimality gap for this solution when compared with the Lagrangian relaxation upper bound. Thus, as the table shows, the Lagrangian relaxation approach discussed in Section 3.6.2 was able to achieve substantially stronger upper bounds than were provided by CPLEX.

We observe from the results that a change in the number of shipment units does not appear to have a meaningful effect on solution quality. When the maximum number of trains increases for instances with the same number of shipment units, the gap reported by CPLEX is slightly lower. However, when we consider the Lagrangian upper bound, it does not appear that the solution quality is in fact affected by the maximum number of trains. On the other hand, when the number of destinations increases, a significant increase in optimality gap is observed for the exact solution.

The performance of the constructive OTP heuristic approach does not seem to be affected by any changes in problem size (number of units, maximum number of trains, or the number of destinations). The overall average optimality gap for this heuristic approach is 1.4%, compared

| | | | Exact | | Heuristic |
|---|---|---|---|---|---|
| **# Destinations** | **#Trains** | **# Units** | **Avg Gap (CPLEX)** | **Avg Gap (LR)** | **Avg Gap (LR)** |
| 4 destinations | 30 | 4000-5000 | 8.81 | 4.01 | 1.36 |
| | 40 | 6000-7000 | 10.75 | 5.67 | 1.64 |
| | 50 | 8000-9000 | 11.17 | 5.52 | 1.57 |
| | Avg for cases with lower number of trains | | 10.24 | 5.07 | 1.52 |
| | 40 | 4000-5000 | 7.33 | 4.41 | 1.08 |
| | 50 | 6000-7000 | 5.79 | 4.03 | 1.19 |
| | 60 | 8000-9000 | 12.41 | 6.72 | 1.54 |
| | Avg for cases with higher number of trains | | 8.51 | 5.05 | 1.27 |
| **Avg for cases with 4 destinations** | | | **9.37** | **5.06** | **1.40** |
| 8 destinations | 30 | 4000-5000 | 17.00 | 8.96 | 1.43 |
| | 40 | 6000-7000 | 15.53 | 7.82 | 1.44 |
| | 50 | 8000-9000 | 22.19 | 9.94 | 1.17 |
| | Avg for cases with lower number of trains | | 18.24 | 8.91 | 1.34 |
| | 40 | 4000-5000 | 16.93 | 9.42 | 1.64 |
| | 50 | 6000-7000 | 11.41 | 8.43 | 1.58 |
| | 60 | 8000-9000 | 22.91 | 10.93 | 1.23 |
| | Avg for cases with higher number of trains | | 17.08 | 9.59 | 1.48 |
| **Avg for cases with 8 destinations** | | | **17.66** | **9.25** | **1.41** |
| **Total Avg** | | | **13.52** | **7.16** | **1.40** |

Table 3.2: Computational Results and Optimality Gap

to 7.16% for the exact approach. As discussed before, the exact model is relatively less time efficient in providing high-quality solutions. That is, while the optimality gap results for the exact approach consider the best solution found by CPLEX within 15 minutes, the solution time for the constructive OTP heuristic algorithm was about 4 seconds, on average. Given the solution time and quality of the heuristic solutions, it seems reasonable to use the heuristic in practice when a high-quality solution is required in a relatively short amount of time.

Figure 3.4 illustrates the difference in solution quality for the different solution methods. In the left figure, the average solution value for each of the 24 problem instance types (# units-max # trains-# destinations) is illustrated for the exact, constructive OTP heuristic, and the upper bound gained by Lagrangian heuristic approach. In the right figure, the average solution value is given based on problem instance size, where the problem instance size is simply calculated as the product of the number of units, the number of trains, and the number of destinations, represented in thousands.

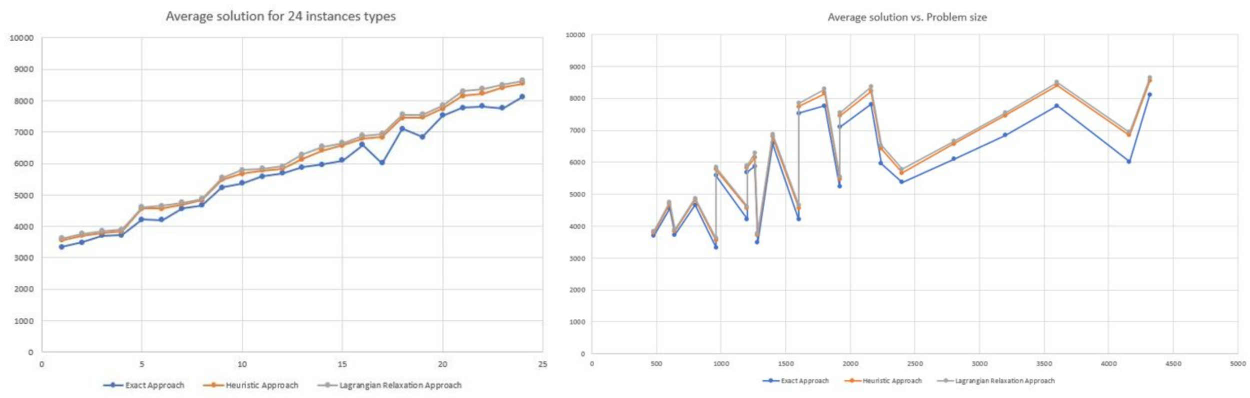


Figure 3.4: Comparison of approaches

To illustrate the potential benefits of our heuristic for smoothness improvement, we next provide an example that shows how the loads within time windows are adjusted for better balance. The smoothness is improved by trying to reduce the gap between minimum and maximum load across all time windows. Recall that we considered a time window as a set of 24 consecutive hours. In Figure 3.5, the height of each bar in the graph corresponds to the total unit load within the window corresponding to the starting hour on the horizontal axis. For example, column 5 shows the total unit load of about 1080 units within the time window that starts at hour 5 and ends at hour 28 (see Figure 3.2). Figure 3.5 shows the result of apply the constructive OTP heuristic on an instance with 6000 units, 40 trains, and 5 destinations. In this case, the maximum and minimum load across all time windows equal 1305 and 456, respectively. The Smoothness Factor in this case equals 0.53.

We take the train schedule provided by the OTP heuristic in Figure 3.5 and fix the train departure times and destinations. We then feed this information into the TAP heuristic for maximizing schedule smoothness. The result is a more balanced schedule as depicted in Figure 3.6. The maximum and minimum load across all time windows, in this case, equal 1221 and 570, respectively. The Smoothness Factor in this case increases to a value of 0.67. Thus, the window load range,

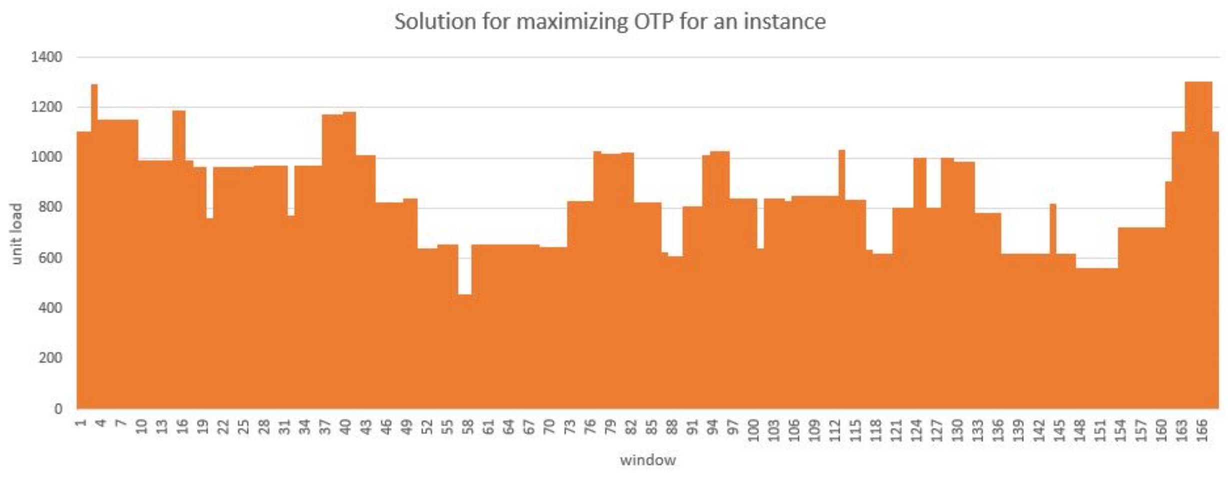


Figure 3.5: Windows' unit load for an instance when OTP is maximized

defined as the difference in total load between the maximum and minimum load within any 24 hour period was reduced from 849 to 651 for this problem instance, a 23.3% reduction.

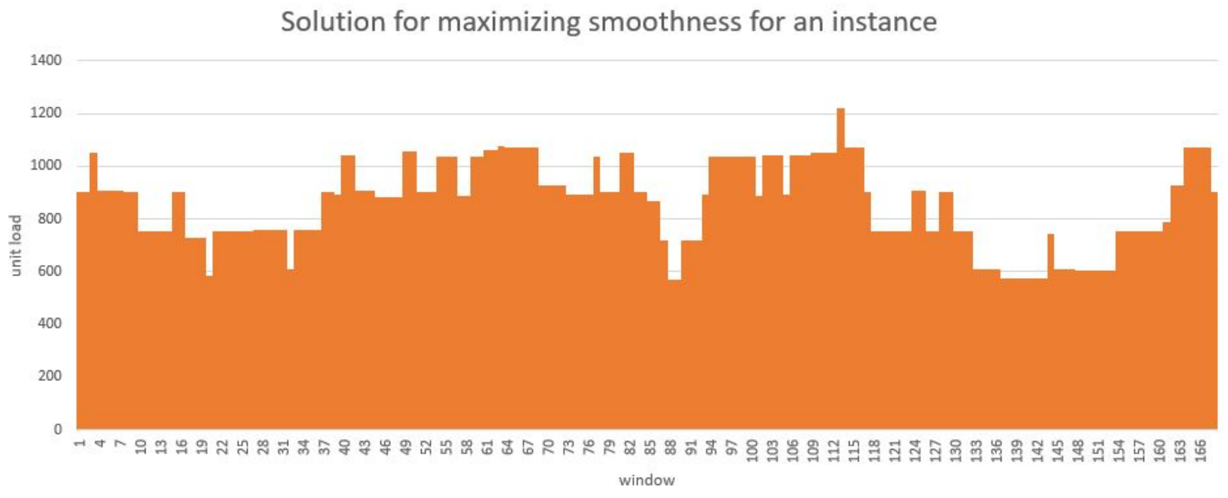


Figure 3.6: Windows' unit load for an instance when schedule's smoothness is maximized

Note that using the train schedule obtained by applying the constructive OTP heuristic is one possibility, and that any arbitrary schedule can be fed in to the heuristic to achieve a more balanced schedule. Obviously, there is a trade-off between schedule smoothness and the degree of on-time performance. In other words, the greater the degree of difference between the schedule obtained by maximizing OTP and the solution after applying the smoothing heuristic, the greater the likelihood of decreasing on-time performance.

The clear advantage of the heuristic approaches that we proposed lies in the solution time. Both the constructive OTP heuristic and the schedule smoothing heuristic solve within seconds. The quality of the solution, as discussed in our computational results, is also reasonably close to optimality. In most practical cases, on-time performance is a priority. Therefore the heuristic for maximizing schedule smoothness can be used as quick post-processing routine after applying the constructive OTP heuristic, in order to improve the load balance across time windows.

## 3.8 Discussion and Conclusion

In this work, we considered the problem of determining departure hours and destinations for outbound trains in a railyard. In this problem, we also provide the assignment of units (containers) to the scheduled trains. The timetable generated for outbound trains is dependent on the specifications of trains and units. Train capacity, each unit's destination, and each unit's on-time and delay interval are some of the restrictive characteristics of our problem. To tackle this problem we consider different objective functions, such as maximizing on-time performance (OTP) and maximizing schedule smoothness (balance). In the former, we desire to have as many on-time shipments as possible, and in the latter, we sacrifice the level of OTP to gain a more balanced schedule. We define smoothness as the difference between the maximum and minimum load across all time windows, where in our examples a time window is a consecutive 24 hour time interval.

To maximize on-time performance, we proposed an integer programming model. We also proposed constructive heuristics to approach this problem. The scale of the problem increases dramatically with parameters such as number of units, number of trains, and number of destinations. The mathematical model for maximizing OTP results in optimal solution for small-scale instances, but for practical size instances, our computational results show that the heuristic approach performs better in terms of solution time without sacrificing solution quality.

To maximize schedule smoothness, we presented a mixed integer programming model. However, due to the scale of the problem for practical size instances, we also propose a constructive heuristic approach. This approach takes in a timetable for the trains and attempts to balance the workload in terms of the number of loaded units across all time windows. In our examples, we defined a time window as a consecutive 24-hour time interval. Computational results show that this heuristic results in a more balanced schedule compared to a solution that maximizes OTP. As an example, We illustrated how the heuristic approach for maximizing schedule smoothness reduces the difference between the maximum and minimum load across time windows at the expense of on-time performance.

The Train Assignment Planning (TAP) problem provides the timetable along with the unit assignments. The output of this problem can also be used for detailed scheduling of the outbound trains, or as a high-level means to test and analyze different scenarios or changes caused by any operational constraints. One example is to evaluate a railyard's capacity expansion requirements in terms of the number of outbound trains or units given a certain level of required on-time performance.

# 4. SUMMARY AND CONCLUSIONS

In this research presented as my dissertation, we have considered problems that arise in railyard facilities. We have used optimization techniques to tackle these problems. A railyard is defined as a set of connected track segments that are used as stations in freight transportation. Railyard activities include loading and unloading railcars with departing and arriving shipments (units). The capacity of a railyard facility is defined by how many units are shipped per unit time. Inbound trains bring in units, while outbound trains transport the units. There is a frequent need to disconnect railcars of inbound trains, reconnect them with other railcars that might have been sitting on some part of the railyard, and construct outbound trains. This frequent daily reshuffling of the railcars and the need to schedule outbound trains to ship the units loaded on railcars led to the definition of multiple operations planning problems for which we proposed solution methods.

The first problem is a single-train shortest route problem. In this problem, given an origin and destination for a train on a railyard facility, we would like to find the shortest route. The special geometrical structure of railyard networks, especially at switch points, makes the use of the shortest path algorithm unpractical. Thus, we propose two approaches that transform the railyard network into an expanded network over which any shortest path algorithm can be used to find the shortest route between the given origin-destination pair. The polynomial-time algorithm for this transformation is easy to implement, accounts for the orientation of the locomotive (pulling or pushing railcars), and allows a train to span multiple nodes in the network.

The second problem also deals with the routing of trains; however, it considers multiple train routing with the possibility of having simultaneous moves. Given a set of origin-destination pairs for trains that are subject to repositioning over the railyard network, we would like to find the set of routes such that the total time required for all trains to reach their destinations is minimized. We propose an integer programming model as well as a heuristic approach to solve this problem. Our mathematical model has the capability of keeping a set of railcars attached during the whole repositioning operation, if necessary. Our heuristic approach utilizes the outcome of the single-train shortest route problem that we proposed as our first problem. It considers multiple possible routes for each train, then iteratively adjusts train routes considering the possible conflicts they may have. The conflicts are resolved by adding a delay right before entering the track segment where the collision with another train may occur. The result of the approaches proposed for this problem is a set of conflict-free routes to reposition multiple trains on a railyard network. Our results show that as the problem size grows, the mathematical approach becomes computationally expensive; however, the heuristic approach provides solutions in fast time regardless of the problem size.

The third problem that we defined deals with scheduling trains. As discussed before, the purpose of repositioning single or multiple trains/railcars on a railyard is to construct outbound trains. The departure times and destinations of these trains depend on the units that are assigned to them. There are two key characteristics for units: destination and allowed departure interval. Units with the same destination that have overlapping departure intervals can depart on the same train. Thus, our scheduling problem aims at determining a timetable for the departing trains, along with the unit assignments. We call this Train Assignment Planning problem and we consider multiple objective functions. By maximizing on-time performance we desire to ship units on-time (within their on-time departure intervals). By maximizing schedule smoothness we aim to have a balanced workload across time horizon (usually a week). We propose integer and mixed integer programming models as well as heuristics to solve these problems. Both approaches provide quality solutions; however, the proposed heuristic approaches provide results in far less computing time.

The three problems proposed in this dissertation are useful tools to simulate railyard operations or test different scenarios. Routing of the railcars and scheduling outbound trains are among the main operations of railyard facilities and providing optimal or near-optimal solutions can help improve operations and capacity.

# REFERENCES

[1] N. Enayaty Ahangar, K. Sullivan, S. Spanton, and Y. Wang, "Algorithms and complexity results for rail-yard routing," *Working Paper, University of Arkansas, Fayetteville, AR*, 2019.

[2] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] J. Cordeau, P. Toth, and D. Vigo, "A survey of optimization models for train routing and scheduling," *Transportation Science*, vol. 32, no. 4, pp. 380–408, 1998.

[4] L. Bodin, B. Golden, A. Schuster, and W. Romig, "A model for the blocking of trains," *Transportation Research Part B*, vol. 14, no. 1-2, pp. 115–120, 1980.

[5] K. Jha, R. Ahuja, and G. Şahin, "New approaches for solving the block-to-train assignment problem," *Networks*, vol. 51, no. 1, pp. 48–62, 2008.

[6] E. Petersen, "Railyard modeling: Part i. prediction of put-through time," *Transportation Science*, vol. 11, no. 1, pp. 37–49, 1977.

[7] M. Turnquist and M. Daskin, "Queuing models of classification and connection delay in railyards," *Transportation Science*, vol. 16, no. 2, pp. 207–230, 1982.

[8] C. Daganzo, R. Dowling, and R. Hall, "Railroad classification yard throughput: The case of multistage triangular sorting," *Transportation Research Part A*, vol. 17, no. 2, pp. 95–106, 1983.

[9] P. Zwaneveld, L. Kroon, H. Romeijn, and M. Salomon, "Routing trains through railway stations: model formulation and algorithms," *Transportation Science*, vol. 30, no. 3, pp. 181–194, 1996.

[10] M. Sama, P. Pellegrini, A. D'Ariano, J. Rodriguez, and D. Pacciarelli, "Ant colony optimization for the real-time train routing selection problem," *Transportation Research Part B*, vol. 85, pp. 89–108, 2016.

[11] J. Lange and F. Werner, "Approaches to modelling train scheduling problems as job-shop problems with blocking constraints," *Journal of Scheduling*, vol. 21, no. 2, pp. 191–207, 2018.

[12] J. Riezebos and W. Wezel, "k-shortest routing of trains on shunting yards," *OR Spectrum*, vol. 31, no. 4, p. 745–758, 2009.

[13] R. M. Lusby, J. Larsen, M. Ehrgott, and D. Ryan, "Railway track allocation: models and methods," *OR spectrum*, vol. 33, no. 4, pp. 843–883, 2011.

[14] A. Higgins, E. Kozan, and L. Ferreira, "Optimal scheduling of trains on a single line track," *Transportation research part B: Methodological*, vol. 30, no. 2, pp. 147–161, 1996.

[15] M. Carey and D. Lockwood, "A model, algorithms and strategy for train pathing," *Journal of the Operational Research Society*, vol. 46, no. 8, pp. 988–1005, 1995.

[16] S. Q. Liu and E. Kozan, "Scheduling trains with priorities: a no-wait blocking parallel-machine job-shop scheduling model," *Transportation Science*, vol. 45, no. 2, pp. 175–198, 2011.

[17] A. Caprara, M. Fischetti, and P. Toth, "Modeling and solving the train timetabling problem," *Operations research*, vol. 50, no. 5, pp. 851–861, 2002.

[18] V. Cacchiani, A. Caprara, and P. Toth, "A column generation approach to train timetabling on a corridor," *4OR*, vol. 6, no. 2, pp. 125–142, 2008.

[19] M. Fuchsberger and P. D. H. J. Lüthi, "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*, 2007.

[20] A. Mascis and D. Pacciarelli, "Job-shop scheduling with blocking and no-wait constraints," *European Journal of Operational Research*, vol. 143, no. 3, pp. 498–517, 2002.

[21] A. D'ariano, D. Pacciarelli, and M. Pranzo, "A branch-and-bound algorithm for scheduling trains in a railway network," *European journal of operational research*, vol. 183, no. 2, pp. 643–657, 2007.

[22] J. Rodriguez, "A constraint programming model for real-time train scheduling at junctions," *Transportation Research Part B: Methodological*, vol. 41, no. 2, pp. 231–245, 2007.

[23] S. Cornelsen and G. Di Stefano, "Track assignment," *Journal of Discrete Algorithms*, vol. 5, no. 2, pp. 250–261, 2007.

[24] B. Peis, M. Skutella, and A. Wiese, "Packet routing: Complexity and algorithms," *International Workshop on Approximation and Online Algorithms*, pp. 217–228 Springer, Berlin, Heidelberg, 2009.

[25] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[26] J. J. Kanet, "Tactically delayed versus non-delay scheduling: An experimental investigation," *European Journal of Operational Research*, vol. 24, no. 1, pp. 99–105, 1986.

[27] M. Resende and C. Ribeiro, "Grasp: Greedy randomized adaptive search procedures," *Search Methodologies - Introductory tutorials in optimization and decision support systems*, no. 2nd edition, pp. 287–312, 2014.

[28] R. K. Ahuja, C. B. Cunha, and G. Şahin, "Network models in railroad planning and scheduling," *Emerging Theory, Methods, and Applications, INFORMS*, pp. 54–101, 2005.

[29] R. C. Wong, T. W. Yuen, K. W. Fung, and J. M. Leung, "Optimizing timetable synchronization for rail mass transit," *Transportation Science*, vol. 42, no. 1, pp. 57–69, 2008.

[30] M. Fischetti, D. Salvagnin, and A. Zanette, "Fast approaches to improve the robustness of a railway timetable," *Transportation Science*, vol. 43, no. 3, pp. 321–335, 2009.

[31] V. Cacchiani, A. Caprara, and P. Toth, "Solving a real-world train-unit assignment problem," *Mathematical programming*, vol. 124, no. 1-2, pp. 207–231, 2010.

[32] V. Cacchiani, A. Caprara, and P. Toth, “An effective peak period heuristic for railway rolling stock planning,” *Transportation Science*, vol. 53, no. 3, pp. 746–762, 2019.

[33] J. Chang, T. Erlebach, R. Gailis, and S. Khuller, “Broadcast scheduling: algorithms and complexity,” *ACM Transactions on Algorithms (TALG)*, vol. 7, no. 4, pp. 1–14, 2011.

# APPENDIX A

## Explanation of $k$ Vertex Cover Reduction

The vertex cover problem begins with a graph $G = (V, E)$ and asks whether a subset of vertices of size $k$ exists such that every edge is incident to at least one of the $k$ vertices in the cover. Let $n = |V|$ and $m = |E|$. Given an instance of vertex cover, the proof of $\mathcal{NP}$-completeness of the broadcast scheduling problem creates an instance of the problem as follows. We first create a page corresponding to each vertex in $G$. Every $n$ periods, starting at time zero, a request is made for every page with a corresponding window size of $n$. Thus, there is a request for every page at times $0, n, 2n, \ldots, (m-1)n$. Notice that we have $m$ blocks of $n$ periods in total (call the one beginning at time zero block 1, and the one beginning at time $(m-1)n$ block $n$).

Letting $\ell = n - k$, then starting at period $\ell$, we once again request each page every $n$ periods, at times $\ell, n+\ell, 2n+\ell, \ldots, (m-1)n+\ell$. Again, each of these requests has a time window of size $n$. Finally, suppose we number all of the edges from $i = 1, \ldots, m$, and let $u_i, v_i$ denote the vertices incident to edge $i$. A request is made for vertices (pages) $u_i$ and $v_i$ at time $n(i-1)+\ell-1$, with a time window of length 3.

If a vertex cover of size $k$ exists, there are $n$ vertices (pages), the $k$ "cover" vertices and the remaining $\ell = n - k$ "non-cover" vertices. In each block, we schedule the broadcast of the $\ell$ non-cover vertices (pages) during the first $\ell$ periods, and we schedule the remaining $k$ cover vertices (pages) during the last $k$ periods. Note that within each of these sub-blocks, we can schedule them in any order, and this can change from block to block if we want. Notice that starting at block 2, each broadcast made in the first $\ell$ periods during the block will satisfy the two requests made at period $\ell = n - k$ and at period $n$. Similarly, each broadcast made in the last $k$ broadcasts of the first block will satisfy the two requests made at period 0 and at period $\ell$ (therefore, each "train" will contain at least two "units"). Now, within block $i$, we also have a request made for vertices (pages) $u_i$ and $v_i$ at time $n(i-1)+\ell-1$ (say, at the end of the $(\ell-1)^{\text{st}}$ period of the block). For every edge $(u_i, v_i)$, if we have a vertex cover of size $k$, then at least one of $u_i$ or $v_i$ is in the cover (possibly both). If both are part of the cover, they can be scheduled in periods $n(i-1)+\ell+1$ and $n(i-1)+\ell+2$. If $v_i$ is in the cover and $u_i$ is not, then $u_i$ can be scheduled in period $n(i-1)+\ell$, with $v_i$ in period $n(i-1)+\ell+1$ or $n(i-1)+\ell+2$. If $u_i$ is in the cover and $v_i$ is not, then $v_i$ can be scheduled in period $n(i-1)+\ell$, with $u_i$ in period $n(i-1)+\ell+1$ or $n(i-1)+\ell+2$ (this is why these requests have windows of 3 periods). Notice then that two broadcasts during each block will cover three requests, i.e., those broadcasts containing the "extra" request for pages $u_i$ and $v_i$. So this is a case of a train going out every period, where each train has two or three units (except for the first $\ell$ periods, which have one unit, and a final set of $\ell - k$ periods extended past period $mn$, which will have one). It can then be shown that if a vertex cover of size $k$ does not exist in the graph, there is no way to schedule all page broadcasts on time.

We can use the same reduction for our problem in polynomial time by exchanging pages for destinations, and exchanging requests for pages with requests for a unit delivery to the corresponding location. We then have that a schedule exists for the corresponding instance of the TAP with all deliveries on time if and only if a vertex cover of size $k$ exists. Thus, if the TAP can be solved in polynomial time, then the vertex cover problem can be solved in polynomial time, which is only possible if $\mathcal{P} = \mathcal{NP}$.

As an aside, recall that demonstrating $\mathcal{NP}$-Completeness requires first showing that our problem is in $\mathcal{P}$. But this is easy because given any solution we can verify its feasibility and objective function value in polynomial time. Second, we need to show that the reduction from the vertex $k$ cover can be done in polynomial time. But given a graph with $m$ edges and $n$ vertices, the reduction to the broadcast problem requires creating a problem with $n$ pages, $\mathcal{O}(mn)$ time periods, and $\mathcal{O}(mn)$ user requests. Our corresponding TAP problem has $n$ destinations, $\mathcal{O}(mn)$ units, and $\mathcal{O}(mn)$ time periods. So this reduction is polynomial.