GREEDY APPROACHES FOR LARGE-SCALE FLOW AND LOAD PLANNING

A Dissertation
Presented to
The Academic Faculty

By

Daniel Ulch

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Industrial and Systems Engineering

Georgia Institute of Technology

August 2022

GREEDY APPROACHES FOR LARGE-SCALE FLOW AND LOAD PLANNING

Date approved: July 7, 2022

ACKNOWLEDGMENTS

I would like to begin by thanking my advisor Alan Erera. Thank you for the hours of mentorship and giving me the ability to work on great projects. I have learned so much technically and professionally from you. I would also like to thank Martin Savelsbergh for his mentorship in the early years of the program. Thank you for giving me the initial idea for my favorite chapter of this thesis. My fondest memories at Georgia Tech are meetings with Alan and Martin. I would like to thank the remainder of my thesis committee: Alejandro Toriello, Pascal Van Hentenryck, and Sushil Poudel. Your helpful feedback and support helped me focus and shape the most important aspects of this work.

In my time at Georgia Tech, I was able to work on industry projects with SF Express and UPS. I am extremely grateful for this opportunity. I would like to thank Alvin Scott, Sushil Poudel, Vincent Wan, and Gaohao for collaborating and giving me real world problems to work on. I would like to thank Alvin in particular for sharing of his expertise during countless meetings and helping me understand every minute detail of logistics operations.

I would like to thank my cohort in ISyE for making my time at Georgia Tech so memorable. The discussions and group studying in the common offices and lounges helped me immensely. The *select select select crew*, Arden, Adam, Chris, Tony, Hannah, Lacy, Alex, Connor, and Jaden, in particular organized and held so many amazing events and vacations. I will miss you all. I would like to thank the *bike crew* for so many healthy and enjoyable rides and fond memories. Finally, I would like to thank everyone who attended the lunch basketball games. These social groups truly enhanced my experience beyond any reasonable expectation.

Finally, I would like to thank my family for all their support. I could not have completed this work without the support of my mom Gale who was always there for me during our weekly calls. To my aunt Debbie, my uncles Michael and John, and my brother Johnathan, thank you for always being supportive and encouraging.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The flow and load planning problem is to determine freight flows through a network and to schedule containerized dispatches to accommodate these flows. The problem is perhaps the core decision problem faced by parcel express companies, LTL carriers, and large online retailers that operate private consolidation networks. Cost minimization is achieved by consolidating goods into common trailers along their journey. Goods are released to the network at specified times and must be delivered before their due date. Goods are typically grouped into sets with common origin, release time, destination, and due date; such a group is referred to as a *commodity*. The scale of the problems faced in industry can involve thousands of locations and millions of commodities; however, most approaches in the literature are exact and are only able to solve instances with hundreds of commodities. The work in this thesis focuses on developing heuristic approaches that can solve industry-scale instances.

In Chapter 2, we provide a sequential marginal cost path based approach to solve the flow and load planning problem. Commodities are sequentially pathed on a partially time-expanded network with optimistically mapped arcs. Heuristic dynamic discretization discovery (DDD) is used to detect when consolidations are not possible due to mapping error and to specify which time points must be added to prevent the infeasibility. Several improvements are made to the shortest path algorithm to exploit the repeated structure of the time-expanded network. Computational experiments show that the approach has an optimality gap of 4-18% on instances which can be solved with exact methods ($\sim$300 commodities). Further experiments show that the approach outperforms a commonly used slope scaling heuristic by 16-34% on both small and realistic sized ($\sim$100k commodities) instances.

In Chapter 3, we develop a different sequential marginal cost path approach that removes the requirement for maintaining and altering a time-expanded network. The approach maintains a set of dispatches currently in the solution, and a time window associated with each dispatch representing when it can depart. At each iteration, a shortest path algorithm on an expanded state space is solved to determine for a commodity a collection of new or existing dispatches which deliver it from its origin to its destination within its delivery window. Then the set of existing dispatches is updated, and the optimal set of dispatch windows is determined. We prove that finding the optimal set of dispatch windows is equivalent to solving a minimum mean cost path on a dispatch dependency graph. We use the Bellman-Ford algorithm to solve this problem and use a creative way to maintain labels between iterations to improve the efficiency of the approach. Computational experiments were performed and demonstrate that this approach outperforms the approach from the first chapter in plan cost by up to 10% with approximately one tenth of the solution time.

Finally, in Chapter 4, we incorporate many practical constraints into the model. In the literature, most works either ignore the structure of flow plans or force them to adhere to in-tree structures where each flow class has exactly one next stop. In practice, more general structures are used where flow classes can have multiple possible next stops. We present a model for generating flow and load plans where commodities are aggregated into flow classes that may be independently directed at every building. Our model produces plans that guarantee each package will be delivered on time despite the aggregated level of control. We give a general heuristic template that we prove can preserve the flow-rule feasibility of solutions. We give four possible implementations adhering to this template. One of these heuristics was shown to produce cost savings of up to 3.4% and savings in trailer-miles of up to 8.5% on large-scale industry instances in under one hour.

CHAPTER 1
INTRODUCTION

1.1 Problem and Motivation

This work is focused on the development and evaluation of approaches for large-scale flow and load planning. This operational problem is faced by every large parcel express company, E-retailer, and LTL trucking company. It can comprise a large proportion of operating costs with huge potential for savings. Furthermore, the decisions made during flow and load planning have a significant impact on downstream decisions such as trip planning and driver scheduling.

The problem is concerned with determining flows through time and space to deliver goods from their origins to their destinations, respecting delivery windows, and simultaneously determining timed trailer loads having discrete capacity to accommodate all flows. A cost is incurred for every trailer, and a handling cost is incurred every time goods flow through hubs in the logistics network. Inherent in this problem is the transportation vs handling cost trade off: fewer loads over a smaller distance may be achieved if goods meet at some intermediate point to be consolidated. Common metrics of interest of solutions to this problem include the total cost, the average trailer utilization, the total number of trailer-miles allocated, the total volume handled at every hub, metrics of early delivery, and the distribution of the number of stops a good makes along its journey. Decisions that are not included in the problem include how the loads are strung together into routes, how drivers are assigned to routes, and empty-trailer re-balancing.

The focus here is particularly on the large instances faced in industry. These instances are orders of magnitude larger than those that can be addressed by most existing approaches in the literature. For such practically sized instances, the deployment of automated planning or recommendation software in industry is also limited. Most plans were largely developed over years of operational decisions and tweaked by human planners. This work demonstrates that simple-but-fast heuristics can perform well. There are many practical insights on algorithmic efficiency in order to keep solution times reasonable for industry deployment. Table 1.1 contains some metrics for the industry instances we attempt to solve.

The methods introduced in this work can all be incorporated into some form of local search. This is driven by the desired industry application of making changes to some base operational plan given up-to-date demand forecasts and capacity information. Large companies should not operate completely different plans every calendar day; there is some benefit to operational regularity. The methodology herein, particularly that in Chapter 4, was used to develop prototypes of decision support tools for a large package express company.

Table 1.1: Industry instances

Instance	Number of Buildings	Number of Commodities
South China	69	100,281
6/12-South	527	366,525
6/12-East	1,373	2,490,498
6/19-South	527	366,303
6/19-East	1,373	2,495,337
6/26-South	527	365,707
6/26-East	1,373	2,499,377

The value of the ability to adjust plans using automated or recommendation software has become apparent in light of the recent pandemic. Between December 2020 and August 2021, the number of job openings in the transportation, warehousing, and utilities section had doubled [1]. The number of employed truckers dropped significantly during the pandemic, and the demand for goods to be transported changed dramatically between April 2020 and February 2021. [1]. Changes in demand as well as human resources render plans based on average historical operations incredibly inefficient.

1.2 Background

This section aims to provide a very high level introduction of the core background topics that are used in all subsequent chapters of this work.

Although flow and load planning involves some aspect of time, one can view the service network design problem as the underlying problem [2]. This problem is concerned with flowing commodities on a network and allocating discrete capacity on the arcs of the network to accommodate the flows. The objective is to minimize the cost of the discrete capacity and a linear flow cost on the arcs. It has a notoriously bad LP bound.

Flow and load planning can be solved using discrete units of time or with continuous time. When solving using discrete units of time, time expanded networks are typically used to model the problem. In these networks, each node in the geographical logistic network is duplicated for each time point in the horizon. Arcs are then added, each of which represents a movement in time and space. The difference in time between the ends of the arc should approximate the travel time between the locations at the ends of the arc. If one wishes to solve flow and load planning on a fixed time discretization (as opposed to continuous time) one can model the flow and load planning problem as a service network design problem on a time expanded network.

If the discretization is fine enough, the time difference along an arc will equal the travel time; however, this comes at the cost of increased solution time. Some approaches use a coarse discretization. A decision is made whether to map arcs such that they arrive earlier than the travel time (called optimistic mapping) or later than the travel time (called pessimistic mapping). An optimal flow and load planning solution on an optimistically mapped network yields a lower bound on the optimal solution to the problem. There exist approaches in the literature that solve the continuous time service network design problem by finding lower bounds on the coarse discretization and either converting the solution into a feasible continuous time solutions or detecting a mapping error and refining the coarse network by adding new nodes locally [3].

In every chapter, we present methods that can be incorporated into a local search procedure. These procedures begin with an initial feasible solution. There is a set of neighbouring solutions that can be reached by taking an action on the current solution. An example of this from Chapter 2 is that we can remove all commodities on one particular arc and re-optimize only those flows. The set of solutions that can be reached by doing this to any arc is the neighbourhood of the current solution. Local search moves between neighbouring solutions until a time limit or a local optimum is reached.

Many methods presented contain a marginal cost path finding subroutine. This subroutine is a shortest path algorithm to find a path for some commodity. The costs on the arcs are the marginal cost of adding the commodity's weight to the arc. In many cases this is zero if there is sufficient allocated capacity, or the cost of adding an additional discrete unit of capacity. Since coarsely discretized time expanded networks can not be guaranteed to be acyclic, we use a variant of Dijkstra's algorithm [4]. The regular structure of the time expanded network allows us to provide additional pruning strategies to improve the solution.

1.3 Outline and Contributions

In Chapter 2, we show that a simple and easily implementable heuristic can perform well on extremely large instances. Most strategies to tackle such instances were designed to decompose them into smaller problems capable of being handled by a MIP-based local search method; however, having methods to work on the full instance is beneficial. The marginal cost path heuristic presented in this chapter has reasonable performance on small instances that can be solved exactly and outperforms slope scaling, another known heuristic, on large instances by 16-34%. In the development of the marginal cost path heuristic, we demonstrate that specialized algorithmic enhancements made for the special structure of time expanded networks can dramatically affect overall performance.

In Chapter 3, we provide a sequential greedy approach that does not rely on a fixed discretization. In such an approach, one does not need to store the entire time expanded network; only dispatches used by at least one commodity are stored. Furthermore, the approach allows dispatch departure times to shift or move within a known time interval, allowing consolidations that are not possible in sequential pathing on a fixed discretization. We provide an in-depth analysis on the properties and methods of updating two types of departure intervals, each having its own benefits and trade offs. This type of interval modelling and updating can be applied to other domains where intervals are fixed during and updated after some planning iteration. For example, this work could be applied to some kind of sequential time-based route cover problem. We demonstrate that the modelling and algorithms are capable of outperforming the best known heuristics for large-scale flow and load planning. The approach outperforms MCPH from Chapter 2 on industry scale instances by up to 10.48% in one tenth of the solution time.

In Chapter 4, we incorporate the practical constraints necessary for solutions to be directly operable. In the work of Chapter 2 and Chapter 3, we assume each commodity can be independently directed at every building; however, in practice, there is not sufficient automation present at each building to distinguish among commodities. In the package express setting, packages are typically directed to an outbound trailer based only on their final destination and service class. Existing approaches in the literature consider the case where flow belonging to any particular destination and service class has a unique next stop from each building. They model this using "in-tree" constraints which derive their name from the fact that flow into any destination forms a tree for each service class. In actual operations, packages having the same final destination and service class might have two or more possible next stops. For example, if there is 1.1 trailer loads of flow sharing the same final destination and service class, decision makers typically send 1 trailer direct to the final destination and send the remaining 0.1 trailer load to some intermediate building to consolidate with other flow. In this case, flow will not adhere to in-tree constraints, but packages are still directed based only on their final destination and service class. We aggregate commodities into flow classes that can be directed through the network by flow rules. We provide a model that determines flow rules and timed loads such that all commodities can be delivered on time even though they are not individually corrected. We give a general scheme for developing improvement heuristics that produce flow-rule-feasible plans, and we provide four possible implementations. One of these heuristics, Alt, was shown to produce cost savings of up to 3.4% and savings in trailer-miles of up to 8.5% on large-scale industry instances in under one hour.

CHAPTER 2
SEQUENTIAL MARGINAL COST PATH APPROACH

2.1 Introduction

Flow and load planning is an operational problem faced by every large parcel express company, E-retailer, and LTL trucking company. It is concerned with determining paths in time and space through which every good in the system is delivered from their origin to their destination. Along each leg of these paths, discrete units of capacity, for example trailers, are allocated and an associated cost is incurred; goods can share this capacity. The goal is to efficiently consolidate goods into common trailers to achieve minimum cost. Goods sharing common characteristics, such as origin, destination, origination time, and due time, are typically aggregated into groups called commodities.

The problem is a combination of the flow planning problem and the load planning problem which have traditionally been solved sequentially. The flow planning problem is to determine for each commodity a spatial path, a sequence of locations independent of time, from its origin to its destination. Additional tree-like structure is often imposed on the flow plan to simplify manual sort operations. Once the flow planning problem is solved, each commodity is restricted to travel along its path in the flow planning solution, and the load planning problem is solved to determine which commodities are consolidated into common vehicles so that they can be delivered along their predefined routes while their release and due times are respected.

This work focuses on a particular application of flow and load planning for a large Chinese parcel delivery company. Recently, the demand for one-day and even same-day delivery has increased dramatically, leading to extremely tight delivery windows. This makes the sequential solution approach of solving the flow problem first and then the load planning problem too restrictive.

Existing approaches to solve the flow and load planning problem tend to model the problem as a service network design problem on a time-expanded network. They typically incorporate inter-programming (IP) based local search heuristics. However, the time sensitivity of the parcel delivery setting requires a very detailed time expanded network. The problem is complicated even further by the multitude of parcels of different service classes. This can lead to hundreds of thousands of commodities. These factors make the problem too large for the application of IP methods.

The marginal cost path approach proposed in this work produces a solution on a partially expanded time-space network. Down-to-the-minute modelling is required for the application of interest; therefore, operating on a fully time-expanded network is not possible. Arcs in the network are mapped optimistically: if the exact arrival time does not exist in the partial network, the head of the arc is selected to be the latest node occurring before the actual arrival time. This means that solutions on the partial network might not convert to a time feasible schedule, and solutions on the partial network are a lower bound on the actual problem.

The approach sequentially adds one commodity at a time to a partial solution by finding a single path on a partially expanded network. This can be performed by solving a shortest path problem having non-negative costs equal to the marginal cost of adding the commodity to an arc. Even on extremely large networks with hundreds of thousand of commodities, this can be completed quickly. This sequential procedure can be used as a construction heuristic to generate an initial solution or as a component of a local search heuristic.

If the error induced by the mapping and coarse discretization has the potential to make a path time-infeasible (as in the case where a commodity departs a location in the network before it arrives in real time), a time-refinement procedure adds additional nodes to the time-expanded network to locally remove the mapping error. This is a simplification of the MIP based refinement used in the Dynamic Discretization Discovery (DDD) literature, which is used due to the scale of the problem.

The approach continues alternating between solving shortest paths and performing time-refinement until all commodities have a time-feasible path in the network. The efficiency of the approach comes from the fact that the most difficult subproblem solved is a shortest path problem.

We demonstrate through a computational study that this simple and easily implementable heuristic can generate high quality solutions on extremely large instances quickly. Approaches currently in the literature decompose large instances into smaller problems and use MIP-based local search to solve them. Having methods that can globally alter large plans in a useful way can be a valuable component of large scale flow and load planning software. In the development of the marginal cost path heuristic, we demonstrate that specialized algorithmic enhancements made for the special structure of time expanded networks can dramatically affect overall performance. The computational experiments show that our heuristic has a gap of 5-18% on instances that can be solved exactly (those with $\sim$300 commodities); however, it generates these solutions in seconds while a MIP solver takes up to an hour to prove the bounds on these solutions. We benchmark against slope scaling on realistically sized instances with up to 100,000 commodities. Our heuristic outperforms slope scaling on every instance by 16-33%.

2.2 Literature Review

For an overview of motor carrier service network design, and methods for flow and load planning, see [5]

As mentioned in the introduction, some existing approaches solve the service network design problem on a fully time expanded network. In the freight transportation context, typical modelling of service network design yields a class of mixed-integer network optimization problems which cannot be solved efficiently with known optimization methods [2]. [6] explore several classes of valid inequalities for fixed-charge network design, focusing on the effects on models with particular commodity representations. [7] review fundamental techniques for deriving valid inequalities for multi-commodity service network design.

Preceding the more recent work on time-expanded network models, work focusing on the timing of aircraft and vehicles routes in the express shipment setting can be found in [8], [9], [10], and

[11]. These works focus on integer programming formulations and traditional solution approaches such as column generation.

Slope scaling is a heuristic approach for solving fixed-charge flow problems by solving a sequence of linear programming problems and updating costs [12].The application of slope scaling to multi-commodity service network design is also presented in [13].

[14] uses a time-expanded network approach for LTL flow planning incorporating many of the constraints addressed by this work. They introduce a time-space node for each weekday, model a series of integer programming problems, and explore the application of slope scaling heuristics in their solution methodology. Their problem is more difficult, because they enforce in-tree structure on their flow plan.

[15] and [16] approach the flow and load planning problem using IP-based local search heuristics to solve a multi-commodity fixed charge flow problem on a time expanded network.

Given a solution to the flow and load planning problem, [17] develop an approach to generate operation driver schedules and routes.

[3] propose an exact algorithm for solving the continuous time service network design problem using partially time expanded networks. This approach uses the idea of optimistically mapping arcs in a partially expanded time-space network, and iterating between solving a network design problem on this network and refining the network until the solution to the network design problem can be converted into a time-feasible solution to the overall problem. However, the network design problem, and the method of identifying how to refine the network involve solving optimization problems that are not tractable for the scale in which we are interested.

[18] extend the work in [3] by associating time windows with the departure times of dispatches. They use a structure they call the *solution graph* which provides an efficient way for determining whether a solution on the partially discretized network can be converted to a feasible schedule. Furthermore, if the solution cannot be converted, the structure provides an efficient way of determining how the network must be refined to remove the mapping error inducing the time-infeasibility.

However, the approach still relies on solving IPs that are too large in our setting.

2.3 Problem Description

The spatial geography of the logistics network can be represented as a graph $F = (L, A)$, which will be referred to as the *flat network*; each vertex is a hub location, and each directed arc represents the ability to travel and deliver freight from the hub represented by the tail of the arc to the hub represented by the head of the arc. Typically, for each arc $a \in A$, there is an associated set of vehicles V_a which can deliver freight along the arc. The vehicle sets are arc dependent, because the docks required for every vehicle type are typically not present at every hub in a logistics network. Let $V = \bigcup_{a \in A} V_a$ be all vehicle types. Each vehicle $v \in V$ has a capacity $Q_v > 0$ corresponding to the mass or volume limits of the vehicle. Each arc $a \in A$ has an associated travel time τ_a. For each arc $a \in A$, and each vehicle type available on that arc $v \in V_a$, there is a cost $C_{av} > 0$ of scheduling one vehicle of that type on that arc.

Let T denote the time horizon: the set of time points; it contains all times at which vehicles can depart from and arrive at any location, as well as all time points at which supplies and demands arrive to the system. Since we assume that there is no cost of holding, we can enumerate T in a discrete manner by propagating arrival and departure times from every commodity origin and destination. All other time points would be redundant. This set is not continuous; however it grows rapidly in the size of F and the time-span of the commodity demand and supplies.

We denote by K the set of commodities. Each element $k \in K$ represents a group of goods that can be individually directed through time and space in the system. We assume the commodities are aggregations of goods sharing common origin, destination, release time, and due time. Note that if release and due times are modelled on a minute-level granularity, directing each individual package would require significant automation: a package due at 4:00am might be sent on a different vehicle than an identical package that is due instead at 4:05am.

Let q_k denote a commodity's weight or volume in the same units as Q_v for vehicles. Let

$o_k \in L$ denote the location in which commodity k originates and $e_k \in T$ denote the time at which it becomes available for shipping. Let $d_k \in L$ denote the the destination of commodity k, and $l_k \in T$ denote the time at which it is due at the destination. We have assumed each $k \in K$ has unique o_k, e_k, d_k, and l_k.

Let

$$\delta_{ik}^t = \begin{cases} -1 & \text{if } i = o_k, t = e_k \\ 1 & \text{if } i = d_k, t = l_k \\ 0 & \text{otherwise} \end{cases}$$

for $i \in L, k \in K, t \in T$.

Let x_{ak}^t denote the proportion of commodity $k \in K$ dispatched on the arc $a \in A$ at time $t \in T$. Then $0 \leq x_{ak}^t \leq 1$. For each sequential $t_1 < t_2 \in T$ and each $i \in L$, let $x_{ik}^{t_1 t_2}$ denote the proportion of commodity k that remains in location i for the period of time $[t_1, t_2)$. For $t \in T$, we refer to the time point immediately preceding t as t^-, and the time point immediately after t as t^+. Let y_{av}^t denote the number of vehicles of type $v \in V_a$ dispatched on arc $a \in A$ at time $t \in T$. Then the flow and load planning problem can be represented as

$$\min \sum_{a \in A} \sum_{v \in V_a} C_{av} \sum_{t \in T} y_{av}^t$$

$$\text{s.t.} \sum_{a \in \delta^-(i)} x_{ak}^{t-\tau_a} - \sum_{a \in \delta^+(i)} x_{ak}^t + x_{ik}^{t^- t} - x_{ik}^{tt^+} = \delta_{ik}^t \qquad \forall i \in L, k \in K, t \in T \qquad (2.1)$$

$$\sum_{k \in K} q_k x_{ak}^t \leq \sum_{v \in V_a} Q_v y_{av}^t \qquad \forall a \in A, t \in T \qquad (2.2)$$

$$y_{av}^t \in \mathbb{Z}^+ \qquad \forall t \in T, v \in V_a, a \in A$$

$$x_{ak}^t \in \{0, 1\} \qquad \forall a \in A, k \in K, t \in T$$

$$x_{ik}^{t_1 t_2} \in \{0, 1\} \qquad \forall i \in L, k \in K, t_1 < t_2 \in T$$

In different applications, one may desire that commodities be *splitable*, in that the entire weight of the commodity need not flow together through the system. In this model this corresponds to whether or not the x variables should be restricted to be binary. We proceed with non-splitable commodities based on our application of interest in parcel delivery.

Constraints (2.1) ensure flow balance for every commodity, and constraints (2.2) ensure that commodities can only be dispatched along arc a at time t if sufficient vehicle capacity has been scheduled along that link at that time.

One can also view the above problem on a time expanded network $N_T = (L_T, A_T)$, where, given the same set of time points T, a vertex is created for each vertex in the flat network $F = (L, A)$ at each time point. That is $L_T = L \times T$. For each $(u, t) \in L_T$, we add an arc $a_t = ((u, t), (v, t + \tau_a)$ for every $a = (u, v) \in A$ to A_T, along with arcs $((u, t), (u, t'))$ for $t < t' \in T$. Then the problem described above can be viewed as as a minimum cost multicommodity flow problem with discrete variable capacities on the network N_T. This problem description is notation-wise simpler and shown below.

$$
\begin{aligned}
\min \sum_{a \in A_T} \sum_{v \in V_a} & C_{av} y_{av} && \\
\text{s.t.} \sum_{a \in \delta^-(i)} x_{ak} - \sum_{a \in \delta^+(i)} x_{ak} &= \delta_{ik} && \forall i \in L_T, k \in K \\
\sum_{k \in K} q_k x_{ak} &\le \sum_{v \in V_a} Q_v y_{av} && \forall a \in A_T \\
y_{av} &\in \mathbb{Z}^+ && \forall v \in V_a, a \in A_T \\
x_{ak} &\in \{0, 1\} && \forall a \in A_t, k \in K
\end{aligned}
$$

2.4 Marginal Cost Path Approach

Since T becomes prohibitively large for problems of practical size, one cannot hope to solve the flow and load planning problem exactly. We adopt an approach which optimizes over a time space network which contains a subset of the nodes of the complete time space network. This approach constructs a solution from scratch and then attempts to improve the solution using local search heuristics. Both the construction and improvement procedures rely on sequentially adding commodities to the solution by assigning them to their minimum marginal cost path. The heuristic operates on a partially time expanded network; solutions on this network may not be time feasible. Therefore, the approach contains a procedure to refine the network until the current solution is guaranteed to be time feasible.

There are four key stages to our approach for the flow planning problem:

1. Initialization of a coarse partially time expanded network
2. Sequential pathing of commodities on this network
3. Refinement of this network by adding additional time points
4. Local improvement of the paths on the network

2.4.1 Initialization of a Coarse Time Expanded Network

Notation-wise, we refer to the full discretization as T, and the fully time expanded network as N_T. For some partial discretization T' constructed by us, we refer to the partial expanded network on this discretization as $N_{T'} = (L_{T'}, A_{T'})$ where $L_{T'} = L \times T'$. For all $(l_1, t) \in L_{T'}$ and $(l_1, l_2) \in \delta_A^{out}(l_1)$ there exists an arc $((l_1, t), (l_2, t')) \in A_{T'}$ having $t' \leq t + \tau_{l_1 l_2}$. Furthermore, for sequential nodes at the same location, e.g. (l_1, t) and (l_1, t'), we assume there is a hold arc $((l_1, t), (l_1, t')) \in A_{T'}$.

We initialize a network as follows: Given an initial discretization Δ (e.g. 30 minutes), and a finite time horizon $\tau = \max\{t - s : t, s \in T\}$, we create a node every Δ time units at each location l. That is, let the initial set of time points be $T' = \{0 \leq t \leq \tau : t = i\Delta, i \in \mathbb{N}\}$, then the initial set of nodes is $L_{T'} = L \times T'$, where L is the set of locations from the flat network. We will add additional time points to T' during the refinement procedure.

Between vertices at a given location which are consecutive in time, we add a *wait arc*, which models waiting at the location during that time interval. That is, we add $((l, t), (l, t + \Delta))$ to the set of arcs $A_{T'}$ for each location l and time $t \in T'$ where $t + \Delta \in T'$. We denote the set of wait arcs a $W_{T'} \subseteq A_{T'}$.

Next, for each arc $a = (l_1, l_2) \in A$ of the flat network, we add for every node $(l_1, t) \in L_{T'}$ a *dispatch arc*, an arc representing this travel in the time space network. However, using this construction, there is no guarantee that $L_{T'}$ contains every endpoint of a travel arc. In order to overcome this obstacle, we *map* the head of this arc to a vertex in $L_{T'}$. If the vertex $(l_2, \max\{t' \in T' : t' \leq t + \tau_a\})$ is chosen, we call the mapping *optimistic*, since entities using this arc in the network will arrive earlier in time than physically possible. Similarly, if the vertex $(l_2, \min\{t' \in T : t' \geq t + \tau_a\})$, we refer to the mapping as *pessimistic*. This concept is presented in Figure 2.1.

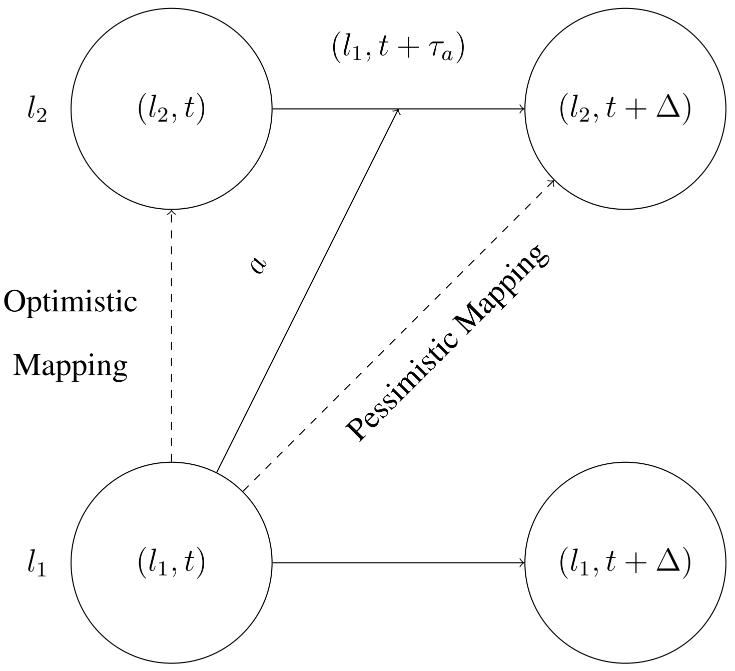


Figure 2.1: Possible mappings of a travel arc from l_1 to l_2 originating at time t

We use an optimistically mapped network, because we can always find a solution. If that solution happens to be time-infeasible, we can insert additional nodes to prevent the infeasibility from reoccurring. If we used a pessimistically mapped network instead, there is no guarantee that a solution could be found on the network. Determining which nodes to add to the network in this scenario is more difficult. Furthermore, even with a feasible solution, profitable consolidations may be prevented by a coarse pessimistic mapping. Identifying these would become an additional component to the improvement routines, rather than simply requiring method to find good new flows.

An optimal solution to the problem on the coarse network with arcs mapped optimistically yields a lower bound on the optimal solution to the original problem. The approach in [3] is to find exact solutions on the coarse network and then attempt to convert this to a solution to the original problem. If this is not possible, the network is refined, and another exact solution on the refined network is found. The approach in this work is similar except that we find heuristic solutions on the partially expanded network using a sequential marginal cost pathing algorithm.

2.4.2 Marginal Cost Path-Finding

Given a set of commodities K, we find a path for each commodity from its origin to its destination satisfying the release time and due time constraints. We do this by processing the commodities sequentially in some order, mapping their origins and destinations to nodes in the network, and solving a sequence of shortest path problems. The cost for a commodity k to use an arc a in the network is the marginal cost of accommodating the weight q_k on that arc. We assume that all wait arcs (moving between two time points at the same location) have a marginal cost of 0.

If there are no trucks scheduled on a, then the cost would be given by the cheapest truck which can accommodate q_k. However, if there exist commodities using a already, and there is room for q_k more units of weight on the trucks scheduled on a, then the cost would be 0. Finally if there exists a truck scheduled on a, but it does not have sufficient capacity to accommodate q_k more unit of weight, but this truck can be replaced by a larger truck, the cost is given by the cost of the upgrade.

Suppose we are finding a path for commodity c. Let K_a denote the set of commodities who have already been pathed and whose path contains a, for each arc a of the partially time expanded network. Let

$$C_a(q) = \begin{cases} 0 & \text{if } a \in W \\ \min_{y \in \mathbb{Z}_+^{|V_a|}} \{\sum_{v \in V_a} C_{av} y_v : \sum_{v \in V_a} Q_v y_v \geq q\} & \text{otherwise} \end{cases}$$

Then the marginal cost on arc a for commodity c is

$$\bar{C}_a(q_c) = C_a\left(q_c + \sum_{k \in K_a} q_k\right) - C_a\left(\sum_{k \in K_a} q_k\right)$$

.

For every arc a, we can pre-compute and tabulate the cost $C_a(\cdot)$ above as a function of the total weight assigned to the arc using a dynamic program. Pseudocode for such an algorithm is given in Appendix A.1. Thus, we can access the marginal costs on the arcs in constant time.

When pathing a commodity k which originates at location o_k at the time e_k and is due at location d_k at l_k, we do not assume that (o_k, e_k) or (d_k, l_k) are vertices in the partially time expanded network. We map the release and due points of the commodity to vertices. We say the commodity is mapped *optimistically* if the source node is chosen to be $(o_k, \max\{t \in T' : t <= e_k\})$ and the destination node is chosen to be $(d_k, \min\{t \in T' : t >= l_k\})$. Similarly, if the source node is chosen to be $(o_k, \min\{t \in T' : t >= e_k\})$ and the destination node is chosen to be $(d_k, \max\{t \in T' : t <= l_k\})$, the mapping is said to be *pessimistic*.

As a result of the optimistic mapping of the arcs, there exist paths in the coarse network that are not *time-feasible* in the sense that a path may use the sequence $((l_1, t), (l_2, t')), ((l_2, t'), (l_3, t''))$ where $t' < t + \tau_{l_1, l_2}$. That is, a truck may depart from an intermediate location in the network before it can actually arrive to that intermediate location in real time. Figure 2.2 displays such a scenario, where the path $\{a, a_1\}$ is time-infeasible. However, the path $\{a, \text{wait}, a_2\}$ is time-feasible, because the entity using the path is available at location l_2 before it departs in real time.

Note that this notion of time feasibility/infeasibility is more strict than declaring a path time-feasible if it can be converted into a solution to the original problem. If only one commodity is using the path $\{a, a_1\}$, it may be the case that this path can be converted to a solution by shifting the departure time of a_1 to $t + \tau_a$. However, identifying the ability to convert the solution would require maintaining a structure like the *solution graph* of [18], which we opted not to in the sake of efficiency.

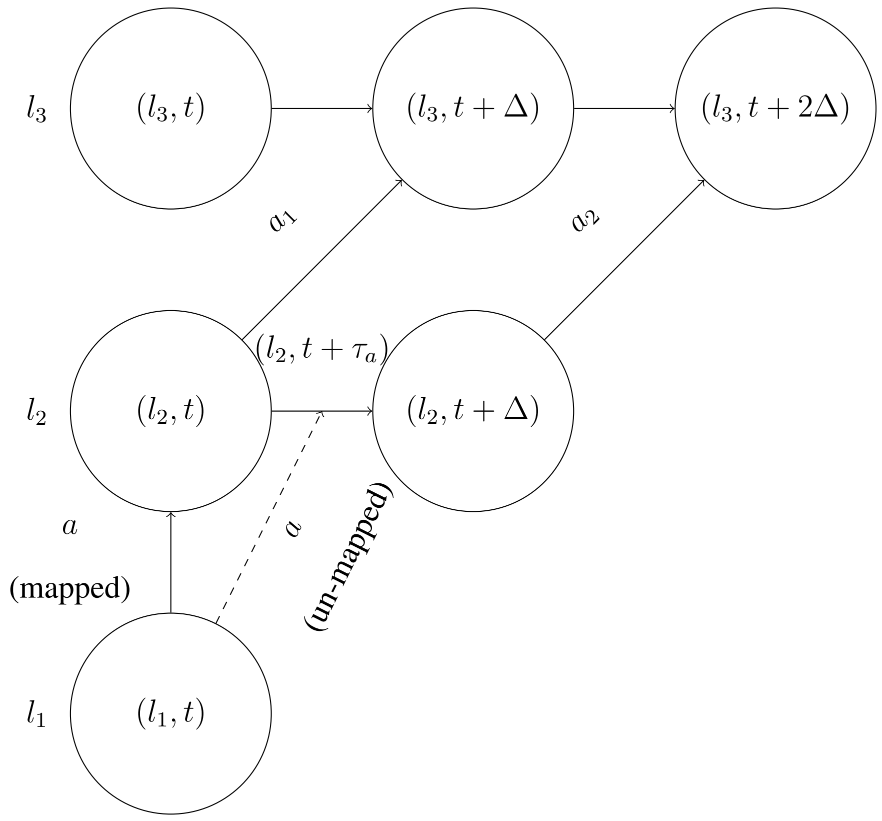


Figure 2.2: A subnetwork containing a single time-feasible $l_1 - l_3$ path and a single time-infeasible $l_1 - l_3$ path.

We assume that all commodities can be delivered on time. This is true if and only if there is a shortest time path in the flat network F between the origin and destination of each commodity of duration at most the length of the delivery window for that commodity. Therefore, any commodities not meeting this criteria can be identified and removed from the instance before beginning this procedure.

Once we have mapped the commodity to a source and destination node, we solve a shortest path problem on the network using the marginal costs of the weight of the commodity. If there exists a path between the source and destination, then it may or may not be time-feasible. When all the arcs of the network are mapped optimistically, the commodity release and due points are mapped optimistically, and the commodity is deliverable, then there must exist a path in the network between the mapped source node and the mapped sink node. Therefore, we need not consider the possibility that the shortest path problem is infeasible.

Algorithm 1 Marginal Cost Pathing

```
1: procedure PathCommodities(List of commodities $K$, Partially time expanded network $N = (L_{T'}, A_{T'})$)
2:     for $k \in K$ do
3:         $o \leftarrow (o_k, \max\{t' : (o_k, t') \in L_{T'}, t' \le e_k\})$
4:         $d \leftarrow (d_k, \min\{t' : (d_k, t') \in L_{T'}, t' \ge l\})$
5:         $P \leftarrow$ShortestPath$(o, d, q_k)$    ▷ Solve shortest $o, d$ path problem on $N$ with costs $\bar{C}_a(q_k)$
6:         if IsTimeFeasible$(k, P)$ then
7:             $FeasiblePaths[k] \leftarrow P$
8:         else
9:             $InfeasiblePaths[k] \leftarrow P$
10:        for $a \in P$ do
11:            $K_a \leftarrow K_a \cup k$    ▷ Update the marginal cost function $\bar{C}_a(\cdot)$ of each arc in $P$
12: function IsTimeFeasible(Commodity $k$, Path $P$)
13:     $a^- = ((l_1^-, t_1^-), (l_2^-, t_2^-)) \leftarrow P[0]$
14:     if $t_1^- < e_k$ then    ▷ If commodity leaves origin before its release time
15:         return $false$
16:     for $i \in 1, ...|P| - 2$ do    ▷ Iterate over consecutive arcs $a^-, a$ in $P$
17:         $a = ((l_1, t_1), (l_2, t_2)) \leftarrow P[i]$
18:         if $t_1 < t_1^- + \tau_{(l_1^-, l_2^-)}$ then    ▷ If commodity departs on $a$ before $a^-$ arrives in real time
19:             return $false$
            $a^- \leftarrow a$
20:     if $t_1^- + \tau_{(l_1^-, l_2^-)} > l_k$ then    ▷ If commodity arrives to destination after its due time
21:         return $false$
22:     return $true$
```

Algorithm 1 gives the pseudo-code for the marginal cost path procedure. The procedure iterates over the set of all commodities, maps the origin and destination of the commodity to vertices in the network optimistically, solves a shortest path problem, records the resulting path as time-feasible or not time-feasible, and updates the marginal cost functions of arcs along the path. Note during this procedure, the structure of the network remains the same. Once this procedure is completed, a portion of the commodities may have paths that are time-infeasible. The next section details how the network is refined by adding time points in order to remove time-infeasible paths from the network.

2.4.3 Time Refinement

The marginal cost pathing approach involves sequentially finding a path for each commodity. If a path is found to be time-infeasible, then nodes are added to the network to remove the mapping error along arcs used in the path. In the case of Figure 2.2, if a commodity used the path a, a_1, then inserting the node $(l_2, t + \tau_a)$ and remapping a to the new node would remove this time-infeasible path from the set of feasible paths in the network.

High level psedueocode for the time refinement procedure is given in Algorithm 2, and detailed pseudocode is provided in Appendix A.2. Given a commodity path which is time-infeasible, the algorithm first determines which vertices should be inserted into the network to prevent any commodity from using this path in the future. Once a new node (l, t) is added to the network, every arc in the forward star of l in the flat network is added originating from (l, t) and optimistically mapped. Finally, existing arcs in the locality of the new node are examined and remapped to the new node if this reduces the mapping error. The reconfiguration exhibited in Figure 2.3, where $v_{\text{new}} = (l_2, t + \tau_a)$ is inserted, and every arc incoming to (l_2, t) is checked to see if it should be remapped to v_{new}. During this reconfiguration process, paths using arcs which are remapped may become dis-contiguous. Thus we repath affected commodities on the refined network. Which commodities are chosen to be repathed is a design decision discussed later.

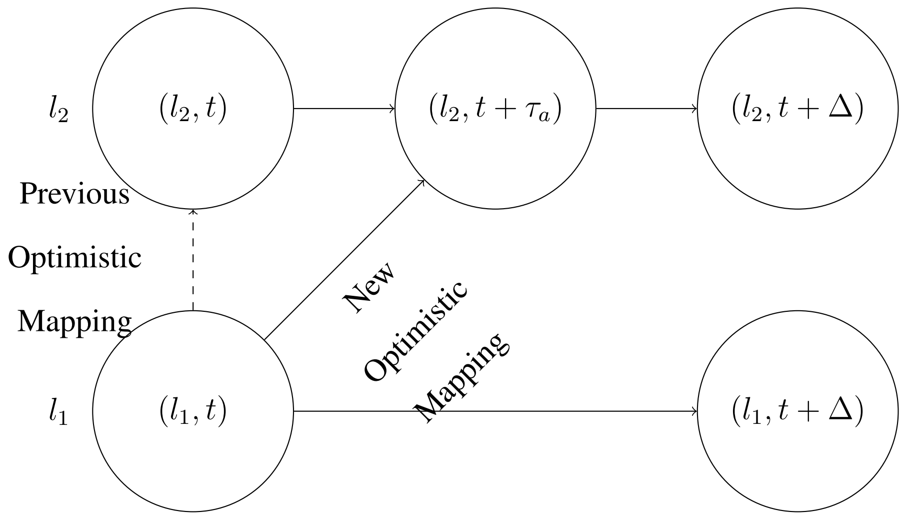


Figure 2.3: The insertion of a new node to the network.

Algorithm 2 Time Refinement

```
1: procedure TimeRefinement
2:     while There exists an infeasible path do
3:         Select an infeasible path $p = ((l_1, t_1), (l_2, t_2), ..., (l_m, t_m))$ and corresponding commodity $k$
4:         Remove $(k, p)$ from the solution
5:         $N^{new} \leftarrow \begin{cases} \{(o_k, e_k)\} & \text{if } t_1 < e_k \\ \{(l_i, t_i)\} & \text{if } \exists\, i : t_{i-1} + \tau_{l_{i-1} l_i} > t_i \; i = 2, ..., m-1 \\ \{(l_m, t_m), (d_k, l_k)\} & \text{otherwise} \end{cases}$
6:         Insert nodes in $N^{new}$ along with their forward stars
7:         Reconfigure arcs which can be remapped to a node in $N^{new}$ with reduced mapping error
8:         Repath commodities affected by the arc remapping
```

2.4.4 Local Improvement

After all commodities have been pathed and time refinement is complete, the algorithm has yielded a feasible solution. This section concerns two local improvement strategies. The first involves iteratively removing paths from the network and repathing them. The second involves selecting an arc, removing all commodities using the arc from the network, and repathing those commodities. One hope is that these methods remove the bias from the order in which the commodities were pathed. These local improvement neighbourhoods are simple when contrasted with those used in IP-based local search work; the simplicity of the neighbourhood structure was motivated by the need for computational efficiency.

Single Path Neighbourhood

The first improvement strategy simply iterates over the set of commodities, removes each commodity's path from the network and repaths it. The pseudocode for this procedure is given in Algorithm 3. Despite the fact that it is very simple, this method works well in practice, which indicates that the sequence in which the commodities are pathed can significantly impact the solution which is generated.

Algorithm 3 Path Improvement

```
1: procedure PathImprovementPass
2:     for $k \in K$ do
3:         Remove $k$ and its path from $FeasiblePaths$ or $InfeasiblePaths$
4:         PathCommodities($\{k\}$)
5:     TimeRefinement()
```

Arc Neighbourhood

The path improvement procedure was found to work well; however, due to the simplistic structure of the neighbourhood, the procedures becomes stuck in a local optimum after a few passes over the set of commodities. Therefore, a second improvement procedure with a richer neighbourhood structure was developed. Pseudocode for this improvement procedure is given in Algorithm 4.

The inspiration for this approach is to find low utilization dispatches, remove all commodities using the dispatch or neighbouring dispatches, and find new paths for these commodities. The hope is that they wont reintroduce the low utilization dispatch. Such an improvement can not be found using the path neighbourhood, because it requires multiple commodities to be removed simultaneously.

Algorithm 4 Arc Improvement

```
1: procedure ArcImprovementIteration
2:     $R \leftarrow \emptyset$
3:     $a \leftarrow$FindArcToImprove()
4:     for $(k, p) \in a$.PathsUsingArc() do
5:         $R \leftarrow R \cup \{k\}$
6:         for $a' \in p$ do
7:             $a'$.DeAssign($k$)
8:     PathCommodities($R$)
```

This procedure is relatively straightforward once one has decided on an implementation of the "FindArcToImprove" function. We experiment with various implementations. Each implementation requires a measure of the cost change induced by improving a particular arc. To obtain the exact cost change, we perform the improvement, compute the cost change along each affected arc, sum the cost differences between solutions, and then undo the improvement. The details of this approach are given in Appendix A.3, but note that it is very computationally expensive.

The first routine we consider is to iterate over arcs which are used by at least one commodity in an arbitrary order, evaluate the cost change induced by improving each arc, and commit to improving the first arc which results in a reduction in the cost of the solution. This is given in Algorithm 5. In the implementation of Algorithm 5, we order the arcs to be considered in ascending order of utilization.

Algorithm 5 First Improving Arc

```
1: function FindFirstImprovingArc
2:     for $a \in A_T$ such that a commodity path uses $a$ do
3:         $\Delta C \leftarrow$ EvaluateArcImprovement($a$)
4:         if $\Delta C < 0$ then
5:             return $a$
6:     return $\emptyset$                                    ▷ No improving arc
```

The second method for finding an arc to improve is to exhaustively evaluate the benefit of improving every dispatch in the network, and to commit to improving the arc yielding the most improvement. This approach is very expensive and is given in Algorithm 6.

Algorithm 6 Best Improving Arc

```
1:  function FindBestImprovingArc
2:      $\Delta C^* \leftarrow 0$
3:      for $a \in A_T$ such that a commodity path uses $a$ do
4:          $\Delta C \leftarrow$ EvaluateArcImprovement($a$)
5:          if $\Delta C < \Delta C^*$ then
6:              $\Delta C^* \leftarrow \Delta C$
7:              $a^* \leftarrow a$
8:      if $\Delta C^*$ then
9:          return $a^*$
10:     else
11:         return $\emptyset$                                ▷ No improving arc
```

2.4.5 Design Decisions

Withing this framework, we address specific algorithmic design decisions made to improve efficiency and solution quality, Namely, improved pruning in the shortest path algorithm, altering which commodities get repathed when arcs are reconfigured during refinement, and how frequently refinement should be performed.

Improved Shortest Path Algorithm

The framework requires repeatedly solving shortest path problems. Therefore, we would like an extremely efficient shortest path algorithm. The time space network maintains a special structure. However, we cannot guarantee that it is acyclic. When using a uniform initial discretization, we can guarantee that all strongly connected components are composed of nodes having the same time. This nearly acyclic property of the network can no longer be guaranteed after the insertion of new nodes.

Therefore, we require efficient shortest path algorithms suitable for graphs with non-negative costs containing directed cycles. We therefore use a variant of Dijkstra's algorithm ([4]). However, we can make several adjustments to a standard implementation of Dijkstra's algorithm to improve performance. Pseudo-code for the standard implementation of Dijkstra's algorithm is given in

Algorithm 7.

Algorithm 7 Dijkstra's Algorithm

```
1:  function DIJKSTRA(Node $s$, Node $t$)
2:      $U \leftarrow \{s\}$
3:      $C[n] \leftarrow \infty \; \forall \, n \in N$
4:      while $U \neq \emptyset$ do
5:          $u \leftarrow \arg\min_{n \in U} C[n]$
6:          $U \leftarrow U \setminus \{u\}$
7:          if $u == t$ then
8:              return $(P, C)$
9:          for $v \in \delta^{\text{out}}(u)$ do
10:             if $C[u] + c_{uv} < C[v]$ then
11:                 $C[v] \leftarrow C[u] + c_{uv}$
12:                 $P[v] \leftarrow u$
```

The first adjustment is to maintain the length of the shortest time path between each pair of vertices in the flat network (each pair of locations) $\tilde{\tau}_{l_1,l_2}$. With this information, one can prune any time space node (l, t) from consideration if $t + \tilde{\tau}_{l,d(k)} > l_k$. That is, if a commodity cannot visit that node and subsequently visit its destination location before the due time of the commodity.

The second adjustment harnesses the fact that, typically, the flat network is nearly complete. Then for most commodities, there exists a direct arc from their origin location to their destination location during their delivery window. The marginal cost of the cheapest such arc is an upper bound on the optimal marginal cost path. Therefore, any node whose cost to visit is larger than this upper-bound need not be added to the ordered frontier. This reduces the amount of time spent managing the frontier, because U is implemented as an ordered set. During the course of execution, we may further reduce this upper-bound threshold: it can be assigned the minimum marginal cost to visit any node at the destination location.

Algorithm 8 Improved Dijkstra's Algorithm

```
1: **function** IMPROVEDDIJKSTRA(Node $(l_o, t_o)$, Node $(l_d, t_d)$)
2:     $U \leftarrow \{(l_o, t_o)\}$
3:     $C[n] \leftarrow \infty \; \forall \, n \in N$
4:     **if** $l_d \in \delta_F^{\text{out}}(l_o)$ **then**                ▷ If there exist directs from $l(s)$ to $l(t)$
5:         $d = (u, v) \leftarrow \arg\min\{c_a : a = ((l_1, t_1), (l_2, t_2) \in A_T, t_1 \geq t_o, l_1 = l_o, t_2 \leq t_d, l_2 = l_d\}$    ▷ Find cheapest direct during time frame
6:         $UB \leftarrow c_d$
7:         $C[(l_d, t_d)] \leftarrow UB$
8:         $P[(l_d, t_d)] \leftarrow v$
9:         $C[v] \leftarrow UB$
10:        $P[v] \leftarrow u$
11:        $C[u] \leftarrow 0$
12:        $P[u] \leftarrow (l_o, t_o)$
13:    **while** $U \neq \emptyset$ **do**
14:        $u = (l_1, t_1) \leftarrow \arg\min_{u \in U} C[u]$
15:        $U \leftarrow U \setminus \{u\}$
16:        **if** $l_1 == l_d$ and $t_1 \leq t_d$ **then**
17:            **return** $(P, C)$
18:        **for** $v = (l_2, t_2) \in \delta^{\text{out}}(u)$ **do**
19:            **if** $t_2 + \tilde{\tau}_{l_2 l_d} \leq t_d$ **then** ▷ Only explore nodes for which there exists a path to $(l_d, t_d)$
20:                **if** $C[u] + c_{uv} < C[v]$ **and** $C[u] + c_{uv} \leq UB$ **then** ▷ Only include nodes of cost at most the upper bound
21:                    $C[v] \leftarrow C[u] + c_{uv}$
22:                    $P[v] \leftarrow u$
23:                    $U \leftarrow U \cup \{v\}$
24:                    **if** $l_2 == l_d$**and**$C[v] < UB$ **then**        ▷ If we can improve upper bound
25:                        $UB \leftarrow C[v]$
26:                        $C[t] \leftarrow C[v]$
27:                        $P[t] \leftarrow v$
```

These improvements are particularly effective due to the structure of the network. There are many duplicated nodes minutes away in time which, if added to the frontier, result in a lot of time wasted sorting.

Repathing Strategy

This section describes different ways to determine which are those "affected commodities" named in Algorithm 2. During the insertion of a new node $v_{\text{new}} = (l_2, t)$ into the network, we may *remap* existing arcs to it. That is, for an arc $a = ((l_1, t_1), (l_2, t_2))$, we will replace this arc with $a_{new} = ((l_1, t_1)(l_2, t))$ if $t_2 < t \leq t_1 + \tau_{l_1 l_2}$. For any commodity-path using a, we also replace a with a_{new}. However, the resulting collection of arcs may no longer be a path. Specifically if a path departs l_2 strictly earlier in time than t. One approach to solving this problem would be to re-path every commodity whose path is destroyed as such. This approach is given the name Economic Repathing and is given in Algorithm 9.

A downside of the economic repathing approach arises from the fact that other paths (those that use a, but remain paths when $a \rightarrow a_{\text{new}}$) may have had the incentive to use a with the assumption that there would exist the benefits of consolidation with the existing commodities. However, we have no guarantee that the new paths found during economic repathing will use a_{new}. Hence, another sensible approach is to repath every commodity using a, regardless of whether or not it remains a path during the remapping. This approach is given in Standard Repathing in Algorithm 9.

Finally, we explore a more aggressive approach which extends the argument above to a one arc look-ahead in each of the paths using a: the approach repaths every commodity a, and if a is not the last arc on any of these paths, it repaths all commodities using the arc subsequent to a on the individual paths. This approach is titled Aggressive Repathing and is given in Algorithm 9.

Algorithm 9 Methods of Repathing

```
function EconomicRepathing(Arc $a$, Arc $a_{\text{new}}$)
    $R \leftarrow \emptyset$                    ▷ The set of commodities which will be repathed
    for $k \in K_a$ do
        $P_k \leftarrow CurrentPathOf(k)$
        $P_k$.Replace$(a, a_{\text{new}})$
        $K_a \leftarrow K_a \setminus \{k\}$
        $K_{a_{new}} \leftarrow K_{a_{new}} \cup \{k\}$
        if not IsPath($P_k$) then
            $R \leftarrow R \cup \{k\}$
    return $R$
function StandardRepathing(Arc $a$, Arc $a_{\text{new}}$)
    $R \leftarrow \emptyset$                    ▷ The set of commodities which will be repathed
    for $k \in K_a$ do
        $P_k \leftarrow CurrentPathOf(k)$
        $P_k$.Replace$(a, a_{\text{new}})$
        $K_a \leftarrow K_a \setminus \{k\}$
        $K_{a_{new}} \leftarrow K_{a_{new}} \cup \{k\}$
        $R \leftarrow R \cup \{k\}$
    return $R$
function AggressiveRepathing(Arc $a$, Arc $a_{\text{new}}$)
    $R \leftarrow \emptyset$                    ▷ The set of commodities which will be repathed
    for $k \in K_a$ do
        $P_k \leftarrow CurrentPathOf(k)$
        $P_k$.Replace$(a, a_{\text{new}})$
        $K_a \leftarrow K_a \setminus \{k\}$
        $K_{a_{new}} \leftarrow K_{a_{new}} \cup \{k\}$
        $R \leftarrow R \cup \{k\}$
        if $a_{\text{new}} \neq p$.LastArc() then
            $a' \leftarrow P_k$.NextArc$(a_{\text{new}})$
            $R \leftarrow R \cup K_{a'}$
    return $R$
```

Refinement Frequency

Should the network be refined any time a commodity is assigned to a time-infeasible path? On one hand, leaving the commodity on a path that has error, one that it may not be able to use after refinement, could bias new paths to consolidate when the benefits wont be available. On the other hand, by delaying refinement, other commodities, later in the sequence, may provide better consolidation opportunities for this particular commodity if it requires repathing. We explore different refinement frequencies ranging from refining every path to refining after all commodities are assigned a path.

2.5 Computational Experiments

Computation experiments were performed to determine the efficacy of the approach on a real world instance, as well as randomly generated instances. We first provide some experiments demonstrating the impact of the proposed algorithmic enhancements. Then we test the best implementation of the approach on a variety of instances. We have broken the set of test instances we consider into two categories: small and large. The small instances can be solved exactly using a MIP solver and our approach can be benchmarked against the bounds generated by the solver. The large instances are on the scale typically seen in industry, and finding an optimal solution using a solver is hopeless. We benchmark only against slope scaling for the large instances.

2.5.1 Slope Scaling as a Benchmark

In order to evaluate the approach, we need a suitable benchmark. The problem can only be solved exactly for very small instances. On the large instances we address in this work, we have chosen to compare against a slope scaling procedure. Slope scaling has been shown to work well on large scale service network design problems. Since slope scaling has no built-in method for DDD, we can not adapt it to a continuous time scenario. We therefore perform slope scaling on instances with horizons of multiple days, and use a fully-time-expanded network with a one minute discretization. Given a time expanded network, the slope scaling algorithm is given in Algorithm 10. In the implementation of this algorithm, at each iteration, we find the shortest paths for all $k \in K$ in parallel.

Algorithm 10 Slope Scaling

```
1: function SLOPESCALING(Fully time expanded network $N = (L_T, A_T)$, commodities $K$)
2:     $\rho_a \leftarrow \frac{C_a(\epsilon)}{Q_a(\epsilon)} \; \forall \, a \in A_T$                         ▷ Initial costs
3:     $C^* \leftarrow \infty$
4:     while Time remains and solution does not repeat do
5:         for $k \in K$ do                                          ▷ Implemented in parallel
6:             $x^k \leftarrow ShortestPath(k, \rho)$    ▷ Find shortest path from $(o_k, e_k)$ to $(d_k, l_k)$ with arc costs $\rho$
7:         $y^a \leftarrow Q_a(\sum_k x_a^k)$
8:         $\rho_a \leftarrow \frac{C_a(\sum_k x_a^k}{\sum_k x_a^k}$
9:         $C \leftarrow \sum_a \sum_v C_{av} y_v^a$
10:        if $C < C^*$ then
11:            $x^* \leftarrow x, y^* \leftarrow y, C^* \leftarrow C$
```

2.5.2 Industry Instance Characteristics

Random Instance Generator

An instance generator was developed to test the performance of the algorithm on instances of different sizes. The main parameters of the instances are

1. The earliest date and time at which commodities can arrive to the system $\underline{t}$
2. The latest date and time at which commodities can arrive to the system $\overline{t}$
3. The number of vehicle types n_V
4. The dimensions of the rectangular map on which terminals are placed
5. The number of regions in the map
6. The number of terminals in the map
7. The number of hubs in each region
8. The number of commodities

The topology of the flat network was developed as follows. Each terminal was assigned random coordinates following a uniform distribution across the entire rectangular map. The terminals were then clustered into regions using k-means, with k being the given number of regions. Terminals were randomly chosen to be hubs, with weights proportional to the inverse of the distance between the terminal and the mean of the coordinates of all terminals. This was done to bias hubs towards connecting regions. The terminals are the nodes of the flat network. The set of arcs consists of the union of all pairs of terminals in the same region and all pairs of hubs. That is, the hubs induce a complete graph and all terminals in any region induce a complete graph. The travel time between terminals u and v was set to be $\tau_{uv} = \lceil 10 d_{uv} \rceil$ minutes, where d_{uv} is the euclidean distance between the coordinates of u and v.

Given the number of vehicles n_V, we set $V = [n_V]$, $Q_v = 50 + 50v$, and $C_{av} = \sqrt{Q_v}\tau_a \ \forall a \in A, v \in V$.

The origin and destination of every commodity were chosen randomly with equal probability for any terminal. The weight of every commodity q_k was chosen random according to a $Unif[1, 20]$ distribution. The earliest release time of each commodity, e_k, was chosen uniformly in $[\underline{t}, \overline{t}]$. Let $\tilde{\tau}_{o_k d_k}$ represent the minimum travel time between the origin and destination of commodity k. The due time of every commodity was set to be $l_k \leftarrow e_k + \tilde{\tau}_{o_k d_k}(1 + Unif[0.05, 0.25])$.

2.5.3 Impact of Improved Dijkstra's Pruning

To demonstrate the significant impact of the improvements made to the shortest path algorithm, we compared a default implementation of Dijkstra's algorithm (pseudo-code in Algorithm 7) to the enhanced version (pseudo-code in Algorithm 8). The two algorithms were used to construct a solution for various instances (without performing any refinement). Table 2.1 gives the construction time in seconds on synthetic instances for the improved Dijkstra's (column Enh.) and the default implementation (column Reg.). Table 2.2 gives similar results on the industry instance.

Table 2.1: Default vs improved Dijkstra's construction times: synthetic instances

n_R	$\|L\|$	n_H	$\|K\|$	Enh.	Reg.
3	25	3	10,000	4.054	18.91
3	25	3	25,000	10.686	49.45
3	25	3	50,000	19.249	95.566
3	25	3	100,000	41.235	190.54
5	50	3	10,000	15.443	78.564
5	50	3	25,000	32.791	180.695
5	50	3	50,000	58.35	352.374
5	50	3	100,000	124.1	658.43

Table 2.2: Default vs improved Dijkstra's construction times: South China instance

$\|K\|$	Enh.	Reg.
100,281	257.057	1140.549

The results demonstrate that some simple exploitation of the repeated structure of time expanded networks can provide huge benefits in terms of reduced solution time. We proceed in the experiments with the shortest path algorithm fixed as the enhanced version.

2.5.4 Impact of Design Decisions

In order to determine the best refinement frequency and repathing strategy, the following experiment was run using the South China instance. For each repathing strategy and each refinement frequency, a feasible solution was generated to the problem. No improvement was performed. The refinement frequencies chosen were 1,5%,10%,25%, and 50%, where 1 is to refine after every commodity, and x% is to refine after every x% of the commodities have been pathed.

Table 2.3: Refinement frequency and repathing strategy results

Aggressive Repathing	Freq	Time(s)	Plan Cost	$\lvert L_{T'}\rvert$	$\lvert A_{T'}\rvert$	Avg. Utilization
	1	1,608.11	4,764,980	18,314	831,669	0.706
	5%	2,399.30	4,693,760	19,529	892,716	0.710
	10%	2,617.50	4,682,360	19,336	880,319	0.707
	25%	4,715.85	4,696,900	20,564	938,866	0.707
	50%	9,587.15	4,723,940	22,406	1,025,378	0.696
Standard Repathing	1	520.84	4,864,990	15,747	704,618	0.716
	5%	660.8	4,830,050	16,766	755,317	0.719
	10%	793.32	4,754,930	17,097	772,360	0.718
	25%	927.54	4,820,470	18,091	818,396	0.704
	50%	1,443.42	5,058,440	19,165	866,501	0.673
Economic Repathing	1	414.4	4,980,140	14,848	661,390	0.711
	5%	516.68	5,015,670	15,422	689,167	0.703
	10%	533.03	5,120,310	15,390	686,079	0.693
	25%	537.69	5,360,820	15,527	690,575	0.655
	50%	631.23	6,234,190	15,531	687,160	0.565

The results are somewhat counter-intuitive, since the amount of time spent refining, and the number of nodes added to the graph increases as the refinement frequency decreases. Aggressive pathing yields the best plans, but takes substantially longer. However, the solution times are not unreasonable for problems of this size. We proceed in the computational experiments using an implementation with aggressive repathing and refinement after every path.

2.5.5 Results on Small Instances

The full results for all small instances are given in Table 2.7. In the results, we use MCPH to refer to the marginal cost path heuristic approach presented in this work. Slp Sc refers to slope scaling. The MIP solver was given an hour to optimize. For some instances, it was not able to reach or prove optimality during this time, so we provide both bounds and benchmark against the lower bound.

A summary of the performance across all instance classes is given in Table 2.4 for MCPH and Table 2.5 for slope scaling. Average solution times for both methods across classes are given in Table 2.6.

Table 2.4: MCPH gap on small instances

				MCPH Gap		
n_R	$\lvert L\rvert$	n_H	$\lvert K\rvert$	min	avg	max
3	25	3	100	6.56	10.91	13.24
3	25	3	200	10.07	12.71	15.18
3	25	3	300	9.59	13.21	17.97
3	25	3	400	4.95	8.40	11.37
5	50	3	100	9.69	10.99	12.52
5	50	3	200	9.70	11.30	12.86
5	50	3	300	8.46	11.73	14.34

Table 2.5: Slope scaling gap on small instances

				Slp Sc Gap		
n_R	$\|L\|$	n_H	$\|K\|$	min	avg	max
3	25	3	100	19.12	23.66	26.40
3	25	3	200	24.00	25.94	28.27
3	25	3	300	22.64	27.74	30.08
3	25	3	400	21.49	27.01	29.18
5	50	3	100	15.98	19.11	22.79
5	50	3	200	21.27	24.34	25.73
5	50	3	300	23.08	24.44	28.46

Table 2.6: Small instance solution times

n_R	$\|L\|$	n_H	$\|K\|$	MCPH Avg Time (s)	Slp Sc Avg Time (s)
3	25	3	100	0.0746	42.53
3	25	3	200	0.1858	78.34
3	25	3	300	0.316	123.62
3	25	3	400	0.9336	120.60
5	50	3	100	0.1742	283.18
5	50	3	200	0.473	382.72
5	50	3	300	0.9104	435.39

There is no instance for which slope scaling outperforms MCPH in either solution time or solution cost. The solution time of slope scaling is significantly larger than MCPH. At each iteration of slope scaling, a shortest path must be found for each commodity. This computation time, although performed in parallel, is equivalent to the entire construction phase of MCPH.

Table 2.7: Small instance results

n_R	$\lvert L\rvert$	n_H	$\lvert K\rvert$	Replication	MCPH Obj	MCPH Time(s)	Slp Sc Obj	Slp Sc Time(s)	FullMIP LB	FullMIP UB	FullMIP Time(s)
3	25	3	100	1	504,250	0.066	567,850	28.58	447,920	447,920	266.425
3	25	3	100	2	412,980	0.079	491,380	19.946	367,250	367,250	130.948
3	25	3	100	3	519,120	0.065	611,900	88.272	450,410	450,410	322.887
3	25	3	100	4	456,890	0.083	543,250	34.198	399,850	399,850	344.503
3	25	3	100	5	461,290	0.08	532,890	41.653	431,020	431,020	367.826
3	25	3	200	1	826,830	0.229	968,610	61.818	728,530	728,530	1367.04
3	25	3	200	2	773,230	0.183	926,420	91.463	661,130	664,490	3609.52
3	25	3	200	3	738,550	0.163	873,940	66.876	664,210	664,210	888.219
3	25	3	200	4	807,720	0.119	949,240	80.712	685,100	685,100	1512.43
3	25	3	200	5	760,040	0.235	885,640	90.832	666,000	666,000	792.449
3	25	3	300	1	1,049,381	0.379	1,288,520	113.804	812,752	925,300	3607.53
3	25	3	300	2	979,840	0.151	1,188,780	56.459	839,183	841,290	3608.25
3	25	3	300	3	1,061,441	0.269	1,200,337	191.876	853,564	928,547	3608.31
3	25	3	300	4	977,734	0.164	1,147,112	80.426	739,477	802,034	3624.92
3	25	3	300	5	987,951	0.617	1,250,576	175.524	755,632	893,190	3608.1
3	25	3	400	1	1,292,249	0.726	1,609,331	124.03	870,417	1,186,730	3609.54
3	25	3	400	2	1,164,010	1.091	1,465,300	159.369	837,592	1,037,690	3610.39
3	25	3	400	3	1,227,800	0.237	1,459,839	176.473	852,590	1,146,060	3609.88
3	25	3	400	4	1,286,780	1.124	1,608,514	69.512	919,492	1,140,520	3609.19
3	25	3	400	5	1,279,200	1.49	1,712,932	73.623	963,470	1,215,860	3610.12
5	50	3	100	1	514,160	0.183	557,760	133.963	464,350	464,350	399.548
5	50	3	100	2	546,250	0.22	573,070	209.336	481,480	481,480	1161
5	50	3	100	3	524,990	0.193	590,620	755.104	473,080	473,080	1197.06
5	50	3	100	4	480,120	0.14	553,550	88.851	427,410	427,410	1305.58
5	50	3	100	5	504,360	0.135	552,530	228.634	441,220	441,220	970.974
5	50	3	200	1	902,540	0.605	1,024,080	239.194	806,250	806,250	2628.76
5	50	3	200	2	884,990	0.438	1,071,860	391.395	799,130	799,130	2301.04
5	50	3	200	3	887,077	0.205	1,024,280	525.802	714,806	772,990	3616.34
5	50	3	200	4	825,950	0.394	974,370	325.272	732,455	733,700	3613.94
5	50	3	200	5	821,430	0.723	972,029	431.935	721,950	721,950	2551.97
5	50	3	300	1	1,280,410	0.371	1,427,460	269.435	1,095,780	1,096,780	3643.83
5	50	3	300	2	1,208,800	1.393	1,414,570	390.518	937,077	1,085,890	3622.22
5	50	3	300	3	1,133,830	1.412	1,349,410	344.08	892,467	1,037,900	3625.51
5	50	3	300	4	1,189,090	0.408	1,436,040	753.099	920,944	1,027,330	3634.13
5	50	3	300	5	1,158,480	0.968	1,344,760	419.796	906,712	1,018,820	3620.48
5	50	3	400	1	1,464,810	2.183	1,871,571	388.173	990,059	1,404,270	3639.06
5	50	3	400	2	1,369,150	2.951	1,700,030	920.753	953,110	1,297,680	3635.89

2.5.6 Results on Large Instances

For the large instances, the size of the set of commodities ranges from 10,000 to 100,000. An optimal solution can not be found currently in a reasonable amount of time. Therefore, we compare only to slope scaling. Table 2.11 shows the results on all large instances. Similar to the small instances, MCPH outperforms slope scaling in terms of cost on every instance. However, for the largest instances, MCPH begins reaching the time limit of one hour before quitting. A summary of the cost improvement of MCPH over slope scaling across instance classes is given in Table 2.8. The average solution time of both methods is given in Table 2.10 for each instance class.

Table 2.8: Large instance summary

				Imp.		
n_R	$\lvert L\rvert$	n_H	$\lvert K\rvert$	min	avg	max
3	25	3	10,000	27.5	29.9	33.9
5	50	3	10,000	22.6	25.3	29.5
3	25	3	25,000	25.9	29.4	30.6
5	50	3	25,000	25.4	27.3	28.4
3	25	3	50,000	20.9	22.5	24.3
5	50	3	50,000	24.7	26.1	26.6
3	25	3	100,000	16.1	18.1	19.5
5	50	3	100,000	22.5	23.7	25.3

Table 2.9: Large instance times

n_R	$\|L\|$	n_H	$\|K\|$	MCPH Avg Time	Slp Sc Avg Time
3	25	3	10,000	1090.6	3600.8
5	50	3	10,000	918.9	3601.3
3	25	3	25,000	2364.0	3602.1
5	50	3	25,000	2433.8	3603.4
3	25	3	50,000	3429.0	3604.8
5	50	3	50,000	3383.0	3606.0
3	25	3	100,000	3351.5	3605.0
5	50	3	100,000	3602.9	3625.2

Table 2.10: Large instance times

Table 2.11: Large instance results

n_R	$\|L\|$	n_H	$\|K\|$	Replication	MCPH Obj	MCPH Time(s)	Slp Sc Obj	Slp Sc Time(s)	Imp.
3	25	3	10000	1	7,807,562	772.231	11,313,854	3600.74	30.99
3	25	3	10000	2	8,498,513	359.989	11,858,786	3601.14	28.34
3	25	3	10000	3	7,979,189	505.49	11,199,462	3601.32	28.75
3	25	3	10000	4	8,121,972	213.818	11,207,124	3600.39	27.53
3	25	3	10000	5	7,862,328	3601.53	11,901,673	3600.18	33.94
5	50	3	10000	1	11,832,865	36.747	15,304,486	3600.56	22.68
5	50	3	10000	2	10,287,658	3602.76	14,582,272	3602.62	29.45
5	50	3	10000	3	11,010,285	685.307	15,045,394	3602.12	26.82
5	50	3	10000	4	12,175,055	150.953	16,195,680	3600.52	24.83
5	50	3	10000	5	10,860,157	118.753	14,036,197	3600.53	22.63
3	25	3	25000	1	14,452,109	2233.3	20,462,194	3603.38	29.37
3	25	3	25000	2	14,566,572	349.419	19,653,122	3600.63	25.88
3	25	3	25000	3	13,634,433	2031.49	19,606,250	3602.06	30.46
3	25	3	25000	4	14,870,115	3603.8	21,377,883	3603.47	30.44
3	25	3	25000	5	14,957,126	3602.15	21,551,921	3601.04	30.60
5	50	3	25000	1	17,795,391	3600.11	24,814,660	3604.02	28.29
5	50	3	25000	2	18,518,142	1863.56	25,607,409	3603.36	27.68
5	50	3	25000	3	17,976,947	180.045	24,088,497	3603.55	25.37
5	50	3	25000	4	17,732,359	2923.83	24,771,584	3602.08	28.42
5	50	3	25000	5	19,020,463	3601.52	26,014,358	3604.01	26.88
3	25	3	50000	1	23,577,490	2735.78	31,139,236	3602.77	24.28
3	25	3	50000	2	24,443,618	3601.42	31,332,086	3601.1	21.99
3	25	3	50000	3	24,397,782	3600.14	31,203,711	3609.17	21.81
3	25	3	50000	4	29,615,640	3604.13	37,422,178	3602.17	20.86
3	25	3	50000	5	24,662,085	3603.73	32,305,600	3608.72	23.66
5	50	3	50000	1	28,298,059	3603.13	38,552,407	3607.19	26.60
5	50	3	50000	2	26,316,825	2508.85	35,830,681	3604.31	26.55
5	50	3	50000	3	28,797,424	3600.16	39,248,722	3603.44	26.63
5	50	3	50000	4	30,771,044	3602.26	40,872,615	3609.03	24.71
5	50	3	50000	5	29,764,983	3600.39	40,214,342	3606.15	25.98
3	25	3	100000	1	43,487,232	2350.77	51,843,528	3600.3	16.12
3	25	3	100000	2	42,248,895	3602.83	51,583,270	3604.56	18.10
3	25	3	100000	3	37,729,193	3600.73	46,652,340	3603.69	19.13
3	25	3	100000	4	44,286,305	3602.3	55,046,102	3609.24	19.55
3	25	3	100000	5	40,549,522	3600.72	49,347,055	3607.02	17.83
5	50	3	100000	1	53,723,102	3602.34	71,946,463	3622.77	25.33
5	50	3	100000	2	53,364,022	3603.44	69,371,534	3623.65	23.08
5	50	3	100000	3	50,479,254	3604.82	66,385,200	3611.9	23.96
5	50	3	100000	4	50,309,632	3600.41	65,856,738	3626.87	23.61
5	50	3	100000	5	55,764,440	3603.25	71,931,742	3640.73	22.48

Table 2.12 shows the results of the two heuristics on an industry instance. Note that it took both methods over one hour to produce a solution. Therefore, the solution produced by MCPH

had no improvement time, and the solution produced by slope scaling is equivalent to rounding up the optimal solution to the LP-relaxation of the problem. This may not be a fair comparison of the heuristics, but demonstrates that MCPH can produce good solutions in a reasonable amount of time.

Table 2.12: Industry instance results

Instance	MCPH Obj	MCPH Time(s)	Slp Sc Obj	Slp Sc Time(s)	Imp.
South China	4,632,395.45	4016.47	71,802,412.77	4426.55	93.55

Gauging the quality of the solution produced by MCPH is difficult, because we can not benchmark against an optimal solution. However, we can investigate some properties of the plan such as trailer utilization. Figure 2.4 shows the distribution of trailer utilization for the plans produced by each heuristic, and Table 2.13 shows some statistics for these distributions. With an average utilization of 75%, we can conclude that the plan produced by MCPH is a high quality plan.

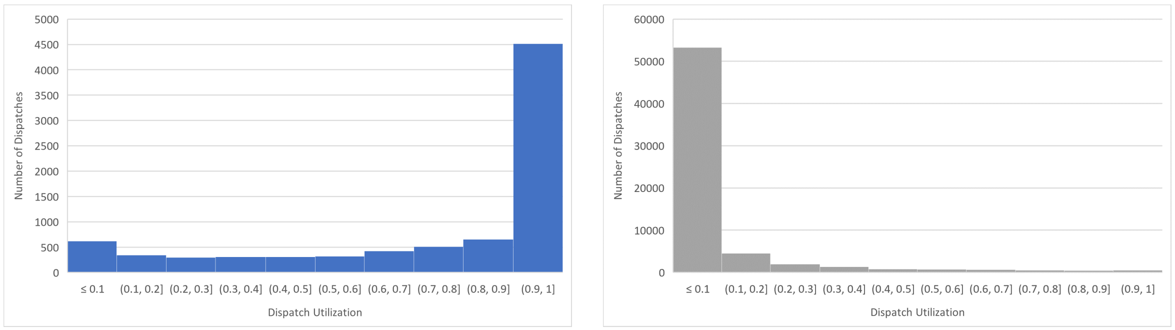


Figure 2.4: Utilization distributions MCPH(blue) vs Slope Scaling(grey): South China instance

Table 2.13: Utilization summary MCPH vs Slope Scaling: South China instance

Method	Min. Util.	Avg. Util.	Max Util.
MCPH	1.4E-04	0.751	1.000
SlpSc	1.4E-05	0.074	1.000

2.6 Conclusion

The main contribution of this work is demonstrating that sequential marginal cost pathing and heuristic dynamic discretization discovery can produce high quality solutions to the flow and load planning problem. We have demonstrated that the heuristic can produce solutions to industry-scale instances in under two hours. Furthermore, we have shown that algorithmic enhancements that exploit the specific structure of time expanded networks can have dramatic effects on the efficiency of the heuristics.

On small instances which can be solved optimally (those having fewer than 400 commodities), the heuristic produced solutions within 6-18% of the optimal solution. On large instances, containing up to 100,000 commodities, the heuristic dramatically outperformed another well known heuristic, slope scaling, by 16-34%. Finally, we showed that the heuristic can produce a plan with 75% trailer utilization for an industry instance in under two hours.

Most approaches currently in the literature focus on solving instances we categorize as small. They involve solving MIPs that when proportionally scaled with the instance, become intractable for instances faced by industry. Furthermore, as more planning adjustments are being made in real time due to weather events and demand shocks, the need for quick solutions in industry is increasing. This approach can be incorporated to quickly adjust plans or to entire produce plans from scratch, providing immense value to parcel express companies and E-retailers facing this difficult problem.

CHAPTER 3
SEQUENTIAL DISCRETIZATIONLESS APPROACH

3.1 Introduction

Flow and load planning is a service network design problem faced in the planning of freight transportation consolidation networks. This problem is to determine freight flows through a network and to schedule containerized dispatches to accommodate these flows. The problem is perhaps the core consolidation decision problem faced by parcel express companies, LTL carriers, and large online retailers that operate private consolidation networks; its solution determines where and when shipments will be consolidated into trailers and containers to best take advantage of transportation cost scale economies. Solutions to the problem are day-differentiated schedules of container dispatches between all terminals and a load plan specifying which shipments are loaded onto which trailers. Freight flow routing through a consolidation network creates handling and sorting for shipments that move through intermediate terminals, so the problem often includes models for the costs and capacities for freight sorting and transfer. Lastly, it is typical that the problem is solved as a prerequisite to detailed transportation scheduling decisions for moving the loads.

Flow and load planning is a *scheduled service network design problem*, and exact approaches for optimization tend to be limited to solving small instances. There is a lack of specialized heuristics for solving the problem on realistic-sized instances, with only a few types proposed in the research literature. One of the difficulties is that typical modelling of these problems involves solving a variant of a service network design mixed-integer linear program on a time-expanded network, see [5].

Earlier work on flow and load planning focused on the LTL industry. Here, freight sorting operations were modeled relatively simply. Predefined sort schedules allowed modeling terminals with at most five time-expanded network layers per day. Furthermore, smaller LTL carriers operate a smaller network of terminals and the resulting smaller number of origin-destination demand pairs means that problem instances could be modeled with relatively few flow commodities. For such problems, exact solution approaches can be developed on a full time-expanded network.

This work addresses the need to develop more detailed plans. The LTL industry has evolved to focus on services that move freight quickly and offer time guarantees. Package transportation companies offer such services (such as two-day and next-day guarantees) and also operate much larger terminal networks. Finally, the largest online retailers are building private middle-mile consolidation networks and also promise fast transit times to purchasers. In such systems, it is important to know precisely when loads arrive for sorting; those arriving 30 minutes later than modelled can lead to many parcels not making service.

This demand for more detailed plans has led to work on partially time-expanded networks and the continuous time service network design problem; modelling the problem on a fully time-expanded network with a fine time discretization leads to intractable models. [3] introduce Dynamic discretization discovery (DDD), an exact solution approach that has been applied to flow and load planning mixed-integer programs with relatively small networks and numbers of commodities. However, package transportation companies face truly massive instances that cannot be solved exactly. There exist few specialized heuristics addressing large-scale flow and load planning. Work on sequential marginal cost pathing with heuristic DDD has shown promise. However, there is a significant drawback in dynamic discretization approaches especially when used within such sequential flow pathing approaches: load dispatches are fixed to occur at a specific time in each candidate solution. In later iterations, there may exist shipments that could consolidate with those already planned if load dispatches could be shifted in time. Exact DDD approaches have been proposed which associate a time interval with each time-expanded node, rather than a particular time (DDDI), see [18]. Again, however, these approaches are not capable yet of addressing large instances.

Our work proposes a heuristic which constructs a solution by sequential marginal cost pathing. However, instead of using time-expanded network, the path finding algorithm is a dynamic program which decides whether to use existing dispatches in the solution or introduce new ones where each dispatch can occur within a time window. A dispatch precedence graph similar to the solution graph of DDDI is used to encode dependencies between commodities and dispatches. At each iteration, we find a path for some commodity not yet in the solution and update the dispatch precedence graph. Next, we solve optimization problems, equivalent to shortest path problems, on the solution graph to determine time intervals for each dispatch in the current solution. These time intervals are used in the path-finding algorithm in the next iteration.

We show that the choice of how the dispatch time intervals are modelled is important. We provide two types of intervals: flexible and rigid. Flexible intervals are easy to update, but do not necessarily yield a feasible path using our proposed dynamic program. We thus also propose rigid intervals, which always yield a feasible path using our proposed dynamic program. We show that finding an optimal set of rigid intervals under the max-min objective is equivalent to finding a minimum mean cost path on the precedence graph, which can be performed in polynomial time. Finally, we conduct a computational study that shows that our approach dramatically outperforms sequential marginal cost pathing approaches on a time-expanded network in both solution quality and required computation time.

The remainder of this work is structured as follows. In Section 3.2 we provide a brief literature review. We formally define the problem in Section 3.3. Section 3.4 outlines the discretizationless solution approach, providing theoretical results on the correctness of the algorithms for maintaining dispatch time intervals. Section 3.5 contains extensive computational experiments benchmarking the approach against other known heuristics. It also contains insights into algorithmic enhancements that dramatically improve the efficiency of the approach. Finally, concluding remarks are given in Section 3.6.

3.2 Literature Review

[5] provide a recent comprehensive overview of service network design and flow and load planning for the trucking industry. [19] provide a review of formulations and a comparison of solution frameworks for service network design. [20] provides an overview of freight transportation modelling.

Flow and load planning has historically been decomposed into sequential subproblems similar to the field of airline crew scheduling. Methods in this field could lend well. [8], [9], [10], and [11] provide work in this area. These works focus on integer programming formulations and traditional solution approaches such as column generation. [21] explain the hierarchy of decision making in air crew scheduling and also model the problem where the departure time of flights can vary within time windows.

For traditional modelling and solution approaches for service network design on LTL networks see [22], [23], [2], and [13]. Note [13] outlines a heuristic we benchmark against in the computational results section. There is extensive work on network design for graphs that are not time expanded, which in itself is difficult. Problems take the form of multicommodity fixed-charge flow problems. [24], [25], [26], [27] provide heuristics to solve these problems. [28] gives an IP local search approach for these problems.

Some works on flow and load planning enforce *in-tree* constraints. These constraints are helpful for designing a practically operable plan, because they direct commodities through the system based on final destination, rather than all characteristics of the commodity such as release time. [14] provides a model and solution approach for this problem that can work on instances with hundreds of commodities using tree-based variables. They model in-tree constraints and solve using a slope scaling and tree generation heuristic. They generate a set of geographic freight flow paths for each commodity in advance and use these to construct solutions. They maintain the same tree flow plan for each day of the week. They note that, even in the LTL setting, moving to a weekly regenerated plan achieves significant cost savings.

The time sensitivity of flow and load planning in the parcel express setting is discussed in earlier works such as [11]. More recently, work on general scheduled service network design optimization models have recognized the need to develop more detailed plans that better account for time constraints. Along these lines, [15] and [16] use heuristic solution approaches to solve problems on more detailed time-space networks.

Some research has also focused on exact solution methods for these more detailed problems. Since modelling these problems on a fully time-expanded network, with layers occurring every minute, leads to intractable models, researchers have investigated partially time-expanded networks and the continuous time service network design problem. [3] propose Dynamic Discretization Discovery, an exact solution approach, was proposed which solves sequences of small mixed integer programs (MIPs) to dynamically expand a time-expanded network. Such a network only contains nodes used by a candidate solution in the solution sequence, and leaves unused regions of the network unexpanded. These exact approaches work well on problems faced by LTL carriers who can collapse their demand-volume onto a small set of commodities ($\sim$500). However, parcel express companies face instances with tens of thousands to millions of commodities. Such instances can not be solved using DDD, and are unlikely to be successfully addressed by any methodology relying on solving multi-commodity service network design MIPs.

[3] propose the Continuous Time Service Network Design problem, where loads can be scheduled on a continuous time axis. In this approach, a time interval is associated with each time-expanded node, rather than a particular time and a solution graph is used to encode the dependencies between commodities and dispatches to recover feasible departure time intervals for every trailer dispatch in the current solution. The paper proposes an exact solution methodology, DDD, which maintains a partially time expanded network having optimistically mapped arcs, and at each iteration solves a service network design MIP on this network. If the solution can be converted into a time feasible solution, it is the optimal solution; otherwise, nodes are selected to insert into

the network, and the process continues. This approach allows precise timing, but can only handle instances with $\sim$500 commodities. [29] provide several enhancements for DDD applied to LTL load planning. The work bases some enhancements on the structure of time expanded networks. They note that a path formulation with a scheme for generating timed-paths in a DDD scheme would be an interesting extension of their work.

[18] extend the work of [3] by associating intervals with dispatch departures rather than a fixed time. However, their solution approach is exact and is only tested on instances having fewer than 1000 commodities.

Sequential marginal cost pathing has seen application in service network design and flow and load planning. It has been explored or used in [30] and [17] where it was performed on a time expanded network with a fixed discretization. [30] propose a heuristic DDD scheme that produces feasible solutions, but induces lots of solution time spent refining the network and repathing commodities.

3.3 Problem Statement

The flow and load planning problem to be addressed in this work is to simultaneously determine the flow of parcels through a logistics network in space and time and schedule trailers to accommodate them along each leg of their journey. A solution to the problem specifies a timed path for each parcel that moves it from its origin to its destination respecting its due time constraint. We assume that parcels originate at consolidation terminals at particular times, referred to as the release times. Each parcel is due at a particular terminal at a particular time, referred to as the due time. The origin and destination terminals are not a choice in the model, and pickup and delivery routing at the origins and destinations is outside of this problem scope. A solution to the problem also specifies a schedule of trailers moving between terminals with sufficient capacity to carry all parcels along their timed-paths. We group all parcels with a common origin terminal, destination terminal, release time, and due time into a single aggregate flow class that is referred to as a commodity.

We restrict the problem further by requiring that all parcels belonging to a commodity must take the same timed-path.

Let $F = (B, A)$ be a directed graph representing the geography of the logistics network. Each element in B represents a consolidation terminal building, and each tuple $(u, v) \in A$ represents an option to send trailers from u to v. Let V_a represent the set of trailer types or vehicles able to operate on link $a \in A$. For any arc a, every vehicle $v \in V_a$ has a capacity Q_v and a cost per vehicle C_{av}.

Let T be the set of time points in the planning horizon. We assume the horizon contains every time at which a vehicle can be dispatched or arrive to any terminal, as well as any commodity release or due times there. Let K be the set of commodities. Each commodity has an origin terminal $o_k \in B$, a destination terminal $d_k \in B$, a release time $e_k \in T$, a due time $l_k \in T$ by which it must be delivered to its destination, and a size q_k which occupies vehicle capacity.

Let

$$\delta_{ik}^{t} = \begin{cases} -1 & \text{if } i = o_k, t = e_k \\ 1 & \text{if } i = d_k, t = l_k \\ 0 & \text{otherwise} \end{cases}$$

indicate when k is released or due at time-space tuple (i, t) for $i \in B, k \in K, t \in T$.

To model the flows of commodities through space and time, let the variable $0 \leq x_{ak}^{t} \leq 1$ represent the proportion of commodity $k \in K$ that is scheduled to depart on arc $a \in A$ at time $t \in T$.

Commodities may also remain at the same terminal for some period of time; let $x_{ik}^{t_1 t_2}$ denote the proportion of commodity $k \in K$ which remains at terminal $i \in B$ for the interval $[t_1, t_2)$ for $t_1, t_2 \in T$.

Let the variable y_{av}^{t} denote the number of vehicles of type $v \in V_a$ scheduled to staff arc $a \in A$ departing at time $t \in T$.

The problem we wish to address is

$$
\begin{aligned}
\min & \sum_{a\in A}\sum_{v\in V_a} C_{av}\sum_{t\in T} y^t_{av} && && \\
\text{s.t.} & \sum_{a\in\delta^-(i)} x^{t-\tau_a}_{ak} - \sum_{a\in\delta^+(i)} x^t_{ak} + x^{t^-t}_{ik} - x^{tt^+}_{ik} = \delta^t_{ik} && \forall i\in B, k\in K, t\in T && (3.1)\\
& \sum_{k\in K} q_k x^t_{ak} \le \sum_{v\in V_a} Q_v y^t_{av} && \forall a\in A, t\in T && (3.2)\\
& y^t_{av}\in\mathbb{Z}^+ && \forall t\in T, v\in V_a, a\in A && \\
& x^t_{ak}\in\{0,1\} && \forall a\in A, k\in K, t\in T && \\
& x^{t_1t_2}_{ik}\in\{0,1\} && \forall i\in B, k\in K, t_1<t_2\in T &&
\end{aligned}
$$

. The objective function is to minimize the total cost of scheduling any vehicle at any time. Constraints 3.1 ensure flow balance on any time space tuple (i, t) for every commodity k. Constraints 3.2 ensure that enough vehicle capacity has been scheduled to accommodate commodities traveling on arc a at time t.

We note here that there is no cost of holding flow at any terminal in this problem. We exploit this in the algorithms we develop where there is a natural trade-off between arriving earlier and the cost to reach a building. We prune states that arrive later at an equal or more expensive cost.

3.4 Methodology

We propose a heuristic methodology which constructs solutions to the problem above by sequentially adding commodities individually to the current solution. The current solution consists of a set of commodities $\tilde{K}$ each of which has a path represented by a tuple of dispatches $(d_1, d_2, ..., d_{n_k})_k$ which deliver it from its origin to its destination. These dispatches do not have fixed departure times, we simply maintain the property that at the end of each iteration, there exists a feasible set of departure times for the dispatches that deliver every commodity in $\tilde{K}$ within their delivery window.

At each iteration, a commodity not present in the current solution is selected, and a dynamic program finds a path for the commodity which minimizes the marginal cost of adding the commodity. Then, an optimization problem is solved to find a dispatch time interval for every dispatch in the current solution. These intervals are provided to the path finding algorithm in the next iteration.

During the execution of the algorithm, a list D, of existing dispatches (those used by at least one commodity) is maintained. Recall the flat network is represented as a directed graph $F = (B, A)$. The relevant data that is maintained for each dispatch $d \in D$ is an origin terminal $u_d \in B$, a destination terminal $v_d \in B$, an earliest departure time α_d, a latest departure time β_d, and the total allocated weight Q_d to be shipped on the dispatch in the current solution.

3.4.1 Path Finding Algorithm

Given a partial solution, and an unpathed commodity k, we find a path for k that minimizes the cost of the additional vehicles required to accommodate k. We are provided with a list of existing trailer dispatches in the current solution, their available capacity, and for each existing dispatch a time interval specifying when the trailer can depart and still make service for each commodity assigned to it. The dynamic program presented here finds a sequence of new and existing dispatches which delivery the unpathed commodity k from its origin to its destination respecting the time constraints of the commodity, such that each existing dispatch in the path can depart within its provided time interval. During the execution of the algorithm, the dispatch intervals for existing dispatches remain fixed. We provide a discussion on modelling dispatch intervals for this purpose, and the possible suboptimality of this algorithm in a later section.

The dynamic program is a labelling algorithm on states (u, e, c) where $u \in B$ is a terminal, e is a time, and c is a cost. e represents the earliest time at which u can be visited at the cost c. We are provided with a list of existing dispatches with data $(u_d, v_d, a_d, b_d, Q_d)_{d \in D}$ where $(u_d, v_d) \in A$ is the arc along which the dispatch travels, $[a_d, b_d]$ is the time interval in which the dispatch must depart, and Q_d is the total volume assigned to the dispatch in the current solution. Let the frontier U be those states which have yet to be reached from. We begin with $U = \{(o_k, e_k, 0)\}$. At each iteration, a state is selected removed from U, removed from U, and is reached from. The algorithm can reach by either using an existing dispatch $d \in D$ or by introducing a new dispatch to D.

When reaching, we use a marginal cost function to generate costs. Let

$$C_a(q) = \begin{cases} 0 & \text{if } a \in W \\ \min_{y \in \mathbb{Z}_+^{|V_a|}} \{\sum_{v \in V_a} C_{av} y_v : \sum_{v \in V_a} Q_v y_v \geq q\} & \text{otherwise} \end{cases}$$

The marginal cost function is defined at

$$C_{u,v_d,Q_d}(q_k) = C_{uv_d}(q_k + Q_d) - C_{uv_d}(Q_d)$$

.

For a state (u, e, c), we first consider reaching via existing dispatches $(u_d, v_d, a_d, b_d, Q_d)_{d \in D}$ where $u = u_d$ and $b_d \geq e$. Each of these dispatches generates a new candidate state $(v, \max\{e, a_d\} + \tau_{uv_d}, c + C_{u,v_d,Q_d}(q_k))$, where $C_{u,v_d,Q_d}(q_k)$ is the marginal cost of adding q_k weight to the existing dispatch. We check to see if this candidate state is dominated by an existing state, and if not, update the set of non-dominated states and add the state to the frontier. A candidate state (u, e, c) is dominated if there exists a state (u, e', c') already visited where $e' \leq e$ and $c' \leq c$. Pseudocode for dominance checking is presented as well in Algorithm 11.

Once we have considered all such existing dispatches, we consider introducing new dispatches. For every outbound location v from u, we consider the new state $(v, e + \tau_{uv}, c + C_{u,v,0}(q_k))$, where $C_{u,v,0}(q_k)$ is the cost of creating a new dispatch with capacity at least q_k. We check to see if this candidate state is dominated by an existing state, and if not, update the set of non-dominated states and add the state to the frontier.

If we order the frontier in ascending order of c, then we can terminate when we select a state

(d_k, e, c) with $e \leq l_k$ to reach from. Furthermore, whenever we consider a candidate state (d_k, e, c) with $e \leq l_k$, we can conclude that c is an upper-bound on the cheapest marginal cost path, and only insert states having $c' < c$ into the frontier.

Algorithm 11 Path Finding Algorithm

```
1: function FindPath(Commodity $k$)
2:     AddState$(o_k, e_k, 0)$
3:     $U \leftarrow \{(o_k, e_k, 0)\}$
4:     while $U \neq \emptyset$ do
5:         Take $(l, e, c) \in \arg\min_{(l', a', c') \in U}\{c'\}$
6:         $U \leftarrow U \setminus (l, e, c)$
7:         if $l = d_k$ and $a \leq l_k$ then
8:             return BackTrack$(l, e, c)$
9:         $\tilde{D} \leftarrow \{(u, v, a, b, Q) \in D : u = l, e \leq b\}$
10:        for $(u, v, a, b, Q) \in \tilde{D}$ do
11:            $\tilde{c} \leftarrow C_{uv}(Q + q_k) - C_{uv}(q_k)$
12:            $\tilde{a} \leftarrow \max\{e, a\}$
13:            if Not IsDominated$(v, \tilde{a} + \tau_{uv}, \tilde{c} + c)$ then
14:                AddState$(v, \tilde{a} + \tau_{uv}, \tilde{c} + c)$
15:                $U \leftarrow U \cup (v, \tilde{a} + \tau_{uv}, \tilde{c} + c)$
16:        for $(u, v) \in \delta_F(l)$ do
17:            $\tilde{c} \leftarrow C_{uv}(q_k)$
18:            if Not IsDominated$(v, e + \tau_{uv}, \tilde{c} + c)$ then
19:                AddState$(v, e + \tau_{uv}, \tilde{c} + c)$
20:                $U \leftarrow U \cup (v, e + \tau_{uv}, \tilde{c} + c)$
21: function IsDominated$((\tilde{l}, \tilde{a}, \tilde{c}))$
22:     if $\tilde{a} < \min_{(a', c') \in S_{\tilde{l}}}\{a'\}$ then
23:         return false
24:     $(a, c) \leftarrow \arg\max_{(a', c') \in S_{\tilde{l}}}\{a' : a' \leq \tilde{a}\}$
25:     if $c \geq \tilde{c}$ then
26:         return true
27:     else
28:         return false
29: function AddState$((\tilde{l}, \tilde{a}, \tilde{c}))$
30:     $S_{\tilde{l}} \leftarrow S_{\tilde{l}} \setminus \{(a', c') \in S_{\tilde{l}} : a' \geq \tilde{a}, c' \geq \tilde{c}\}$
31:     $U \leftarrow U \setminus \{(\tilde{l}, a', c') : (a', c') \in S_{\tilde{l}}, a' \geq \tilde{a}, c' \geq \tilde{c}\}$
32:     $S_{\tilde{l}} \leftarrow S_{\tilde{l}} \cup (\tilde{a}, \tilde{c})$
```

During the execution of the algorithm, we maintain a set of non-dominated states. Let $S_u =$

(a, c) represent a set of non-dominated states at terminal u where a is the earliest arrival time at cost c. Since we assume there are no waiting costs, and a commodity can wait at a terminal indefinitely, the set of non-dominated states $S_u = \{(a, c)\}$ ordered by ascending a must be decreasing in c. This makes checking whether a new state is dominated easy, and updating the set of non-dominated states easy.

For each terminal u, we can store the set of non-dominated states $S_u = \{(a, c)\}$ in ascending order of a. To check whether a candidate state (u, a', c') is dominated, we simply need to find $(\bar{a}, \bar{c}) = \arg\max_{(a,c)\in S_u}\{a : a \leq a'\}$ which can be done in $O(\log |S_u|)$ time. Then the candidate state is dominated if and only if $\bar{c} \leq c'$.

Once a non-dominated state (u, a', c') is found, S_u can be revised to include it. Let $S_u \leftarrow \{(a', c')\} \cup (S_u \setminus \{(\bar{a}, \bar{c}) : \bar{a} \geq a', \bar{c} \geq c'\})$. Since S_u ordered in ascending values of a is decreasing in c, we simply need to start with value a' and continue deleting states until we find a state having $c < c'$.

3.4.2 Dispatch Departure Intervals

When a path is found for a commodity, it is composed of new and existing dispatches. Due to the release and due time constraints of the commodity, the feasible dispatch time intervals may change (e.g. shrink). We face the problem of updating these intervals. We would like them to be as wide as possible so that commodities in subsequent iterations can find paths using them. The problem of updating the dispatch departure intervals can be modelled as an optimization problem on a graph encoding precedence relations between commodities and dispatches. If commodity k uses the path (d_1, d_2) where d_1 and d_2 are dispatches, then d_1 cannot depart before k is released, d_2 cannot depart before d_1 arrives, and d_2 must arrive before k is due. These types of relations govern the dispatch departure windows.

Dispatch Precedence Graph Structure

We construct and maintain a graph G, which we call the dispatch precedence graph, to encode the precedence relations among commodities and dispatches. The vertices of G, $V(G) = E \sqcup V_D \sqcup L$ consist of three distinct types. There exists a vertex $v_d \in V_D$ for every existing dispatch $d \in D$ in the current solution. For each commodity $k \in \tilde{K}$ which has been assigned a path, there exists an origin vertex $n_k \in E$ and a destination vertex $n'_k \in L$. We let k_n denote the commodity corresponding to the vertex $n \in E \cup L$.

For any two dispatches $d_1, d_2 \in V_D$, the arc (d_1, d_2) belongs to G if and only if some commodity uses dispatch d_1 and d_2 in succession on its path. This encodes the information that the earliest departure time of d_2 depends on the earliest departure time of d_1. It also encodes the information that the latest departure time of d_1 depends on that of d_2.

Table 3.1: Example partial solution

Commodity	Path
1	(d_1, d_2)
2	(d_3, d_2)
3	(d_3, d_4, d_5)

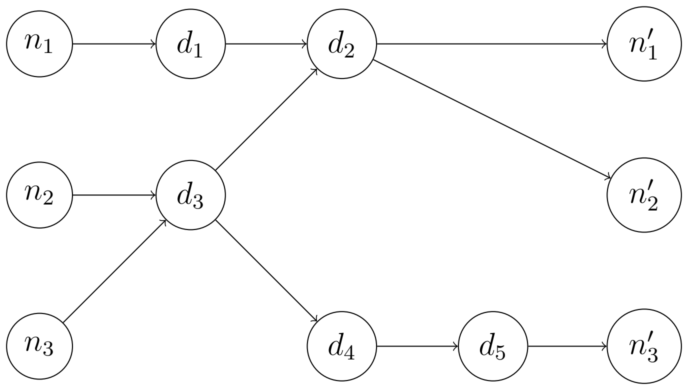


Figure 3.1: Precedence graph corresponding to partial solution

Every origin vertex $n \in E$ has indegree 0 and outdegree 1. The arc (n, v_d) for $v_d \in V_D$ belongs to the graph if and only if d is the first dispatch on the path selected for the commodity k_n. This signifies that the earliest departure time of d depends on the release time of k_n. Every destination vertex $n \in L$ has indegree 1 and outdegree 0. The arc (v_d, n) for $v_d \in V_D$ belongs to the graph if and only if d is the last dispatch on the path select for commodity k_n. This signifies that the latest departure time depends on the due time of k_n. There are no arcs among vertices in E, and there are no arcs among vertices in L.

This representation of a solution is useful in several ways. Fast labelling algorithms can be applied on the graph to determine departure windows for all of the dispatches. Furthermore, it encodes the dependency structure of the solution.

The components of this graph are exactly those “clusters” defined by [31]. In their work, they present a set partitioning formulation with a binary variable for each cluster and suggest a column generation approach to determine which clusters should be included in the solution. They generate high quality solutions using simple cluster templates where few commodities and dispatches are contained in the same component. This suggests that other graph algorithms could be applied on the precedence graph to analyze or improve solutions.

Dispatch Interval Modelling

We consider two possible options for modelling dispatch intervals on the precedence graph which we refer to as rigid windows and flexible windows. We will loosely use $[a_d, b_d]_{d \in D}$ when referring to rigid windows and $[\alpha_d, \beta_d]_{d \in D}$ when referring to flexible windows. A set of rigid windows

$[a,b]_{d\in D}$ is a solution to the system

$$b_u + \tau_u \leq a_v \qquad \forall (u,v) \in A[V_D] \tag{3.3}$$

$$a_u \leq b_u \qquad \forall u \in V_D \tag{3.4}$$

$$e_{k_n} \leq a_u \qquad \forall (n,u) \in \delta(E) \tag{3.5}$$

$$b_u + \tau_u \leq l_{k_n} \qquad \forall (u,n) \in \delta(L) \tag{3.6}$$

whereas a set of flexible windows $[\alpha,\beta]_{d\in D}$ is a solution to

$$\alpha_u + \tau_u \leq \alpha_v \qquad \forall (u,v) \in A[V_D] \tag{3.7}$$

$$\beta_u + \tau_u \leq \beta_v \qquad \forall (u,v) \in A[V_D] \tag{3.8}$$

$$\alpha_u \leq \beta_u \qquad \forall u \in V_D \tag{3.9}$$

$$e_{k_n} \leq \alpha_u \qquad \forall (n,u) \in \delta(E) \tag{3.10}$$

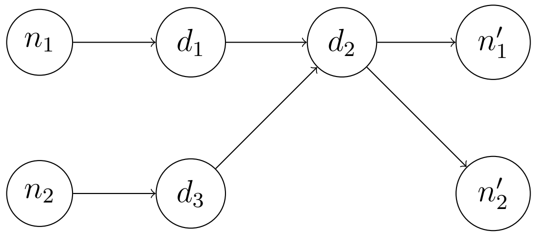
$$\beta_u + \tau_u \leq l_{k_n} \qquad \forall (u,n) \in \delta(L) \tag{3.11}$$

.

We will provide a small example and a few results to provide intuition regarding the difference between the two types of intervals. Consider an example where the commodities given in Table 3.2 have been pathed so far. Suppose commodity k_1 traveled from A to B via terminal C using dispatches d_1 and d_2 in succession, and that commodity k_2 traveled from D to B via terminal C using dispatches d_3 and d_2 in succession. That is, k_1 and k_2 meet at terminal B and travel together in dispatch d_2 to C. Assume $\tau_{d_i} = 1$ for $i = 1,2,3$. The dispatch precedence graph for this solution is given in Figure 3.2.

Table 3.2: Interval modelling example data

k	o_k	d_k	e_k	l_k
1	A	B	0	4
2	D	B	0	4



$$\begin{tikzcd} n_1 \arrow[r] & d_1 \arrow[r] & d_2 \arrow[r] \arrow[dr] & n_1' \\ n_2 \arrow[r] & d_3 \arrow[ur] & & n_2' \end{tikzcd}$$

Figure 3.2: Interval modelling example graph

We desire dispatch intervals as wide as possible, so that commodities in later iterations may consolidate to find cheaper paths. Two obvious objective functions to consider are

$$\max \sum_{u \in V_d} b_d - a_d \tag{3.12}$$

which we refer to as the max-sum objective, and

$$\max \min_{u \in V_d} b_d - a_d \tag{3.13}$$

which we call the max-min objective. Optimal intervals for the max-sum objective are shown in Figure 3.3, while optimal intervals for the max-min objective are shown in Figure 3.4.

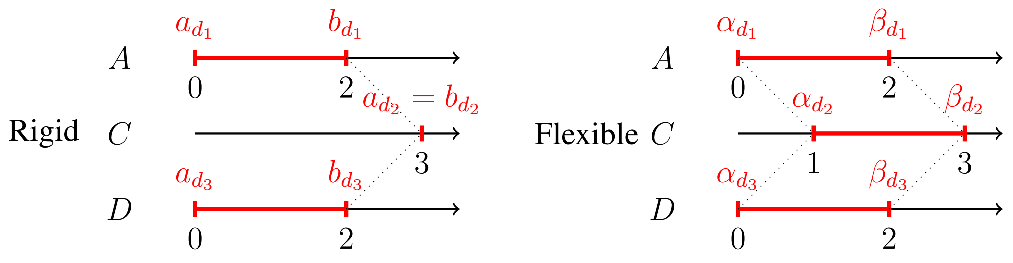


Figure 3.3: Optimal dispatch intervals under max-sum objective

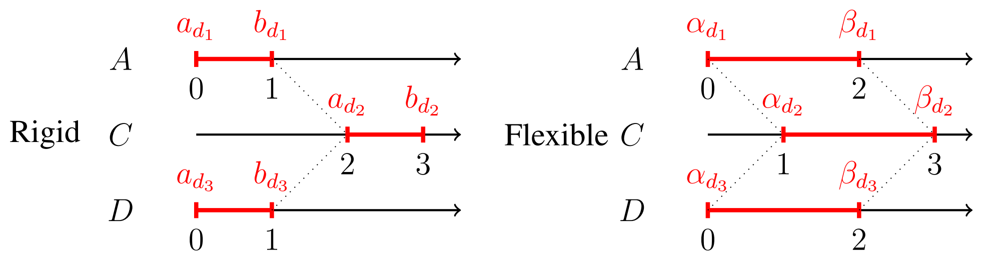


Figure 3.4: Optimal dispatch intervals under max-min objective

There are several properties of these intervals to point out. The first is that the optimal flexible windows are the same for both objective functions, while the optimal rigid windows are different. Furthermore, all of the flexible window widths take the same value, 2. There is a property that the two sets of rigid windows have in common: the sum of the window widths along any $E - L$ path in G is the same under either objective: 2. This is a first glance at the main difference between the two models of intervals. Along $E - L$ paths in G there is some inherent *slack* time. Each $E - L$ path $(n_1, n_2, ..., n_m)$ has an earliest release time corresponding to $e_{k_{n_1}}$ for the first node n_1 in the path and a due time $l_{k_{n_m}}$ for the last node n_m in the path. The internal vertices of an $E - L$ path must lie in V_d. We denote the travel time of the dispatch d_{n_i} corresponding to the node n_i as simply τ_{d_i}.

The slack represents how much time the dispatches along the path can delay departing while enforcing the path is completed in $[e_{k_{n_1}}, l_{l_{k_m}}]$. Imprecisely, flexible window widths take the value of this slack, while rigid windows are a partition this slack. First formally define the slack of an

$E - L$ path $p = (n_1, n_2, ..., n_m)$ as

$$\gamma_p := l_{k_{n_m}} - e_{k_{n_1}} - \sum_{i=2}^{m-1} \tau_{d_i} \tag{3.14}$$

. Note that $\gamma_p = 2$ for all $E - L$ paths p in our example. We provide the following results to further develop this intuition.

Proposition 1. *Consider the case where G is an $E - L$ path $p = (n_1, n_2, ..., n_m)$.*

(a) The flexible window widths for every dispatch will take the value γ_p under either objective. That is,

$$\beta_d^* - \alpha_d^* = \gamma_p \forall d \in V_d$$

(b) The rigid windows under either the max-sum or the max-min objective will satisfy

$$\sum_{i=2}^{m-1} b_{n_i} - a_{n_i} = \gamma_p$$

(c) Under the max-min objective, the rigid windows will satisfy

$$b_d - a_d = \frac{\gamma_p}{m-2} = \frac{\gamma_p}{|V_d|} \forall d \in V_d$$

Proof. (a) Taking $\alpha_{n_i} = e_{k_{n_1}} + \sum_{j=1}^{i-1} \tau_{d_j}$ and $\beta_{n_i} = l_{k_{n_m}} - \sum_{j=i+1}^{m} \tau_{d_j}$ yields feasible windows all having width γ_p.

Aggregating constraints 3.8 and 3.11 yields $\beta_{d_i} \leq l_{k_{n_m}} - \sum_{j=i}^{m} \tau_{d_j}$ for $i = 2, ..., m-1$. Aggregating constraints 3.7 and 3.10 yields $-\alpha_{d_i} \leq -e_{k_{n_1}} - \sum_{j=1}^{i-1} \tau_{d_j}$ for $i = 2, ..., m-1$. Hence,

$$\beta_{d_i} - \alpha_{d_i} \leq l_{k_{n_m}} - e_{k_{n_1}} - \sum_{i=2}^{m-1} \tau_{d_i} = \gamma_p$$

.

(b) A similar aggregation of constraints yields

$$e_{k_{n_1}} + \sum_{i=2}^{m-1} (b_{d_i} - a_{d_i}) + \sum_{i=2}^{m-1} \tau_{d_i} \leq l_k$$

, so $\sum_{i=2}^{m-1} (b_{d_i} - a_{d_i}) \leq \gamma_p$. Taking $a_{d_2} = e_{k_{n_1}}$ and $b_{d_2} = a_{d_2} + \gamma_p$ and $a_{d_i} = b_{d_i} = e_{k_{n_1}} + \gamma_p$ yields a feasible solution achieving this bound.

(c) The bound $\sum_{i=2}^{m-1} (b_{d_i} - a_{d_i}) \leq \gamma_p$ from (b) holds. Take $a_{d_i} = e_{k_{n_1}} + (i-2)\frac{\gamma_p}{m-2}$ and $b_{d_i} = a_{d_i} + \frac{\gamma_p}{m-2}$ for $i = 2, ..., m-1$. This solution is feasible and has $b_d - a_d = \frac{\gamma_p}{m-2}\ \forall\ d \in V_d$.

The result follows from the property that any $\max\min_{i=2,...,m-1}\{x_i\}$ satisfying $x_i \geq 0\ i = 2, ..., m$ and $\sum_{i=2}^{m-1} s_i \leq \gamma_p$ will take value $\frac{\gamma_p}{m-2}$. $\square$

Extending these results to general G is more complicated; however, one can show the following:

Claim 1. *For a general precedence graph G,*

(a) Let P_{EL} denote the set of all $E-L$ paths in G. The optimal flexible windows under the max-sum objective will satisfy

$$\beta_u^* - \alpha_u^* = \min_{p \in P_{EL}: p \ni u} \gamma_p$$

(b) For any optimal set of rigid windows under either the max-sum or the max-min objective, there will exist an $E-L$ path $p = (n_1, n_2, ..., n_m)$ such that

$$\sum_{i=2}^{m-1} b_{n_i} - a_{n_i} = \gamma_p$$

We prove a more general version of (b) in Proposition 3. We leave (a) as a claim but a construction similar to the proof of Proposition 1 (a) taking the maximum of incoming paths for α and the minimum across outgoing paths for β should yield this result. These results reinforce the intuition that finding optimal rigid windows involves partitioning slack across dispatches in the precedence graph.

Another difference between the two models of dispatch departure intervals regards the sets of schedules that adhere to the windows. We use the term schedule to refer to the assignment of a specific departure time to every dispatch in the solution. For some sets of optimal rigid windows, there exist feasible schedules where dispatches depart outside of the windows. This is shown in Figure 3.5 where the optimal rigid windows under the max-min objective from the earlier example fail to accommodate the time feasible dispatch schedule shown in blue. Dispatches d_1 and d_3 depart outside of their optimal rigid windows, but the schedule is still time feasible.

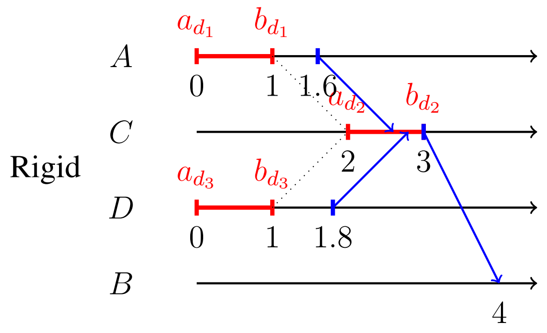


Figure 3.5: A feasible schedule not captured by optimal rigid windows

This is a concern, because our pathing algorithm assumes all dispatches must depart within their provided interval. Using rigid windows in the pathing algorithm can yield sub-optimal solutions to the marginal cost pathing problem, because we are not considering all feasible consolidations. For example, suppose we have an instance with two commodities, each with the same origin and destination. The release and due times of the commodities are given in Table 3.3. Suppose we first path commodity 1 and it takes a path using two dispatches, each of travel time 1 unit. Then, in Figure 3.6, the optimal flexible windows under the max-sum objective are given in blue, and the rigid windows for the max-min objective are given in red. We can see that the rigid windows prevent commodity 2 from being consolidated with commodity 1, even though this consolidation is time feasible.

We can relate planning on a time expanded network with any discretization to planning with rigid windows of width zero where the set of possible dispatches must depart at a point in the time discretization. When a commodity uses a time expanded arc, it is assumed to depart in a window of width zero. Therefore, using windows generalizes sequential heuristic planning on time expanded networks.

On the other hand, the flexible windows under the max-sum objective have the property that dispatches depart in the optimal windows for any feasible schedule. In the example, commodities 1 and 2 can consolidate by having d_1 depart at 3 and d_2 depart at 4. Since flexible windows can accommodate all feasible schedules, we would like to use them whenever possible.

Table 3.3: Example of rigid window suboptimal path finding: commodity data

k	e_k	l_k
1	0	6
2	3	6

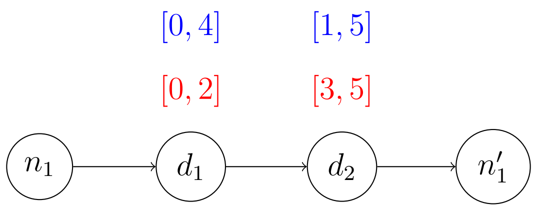


Figure 3.6: Example of rigid window suboptimal path finding: precedence graph and dispatch intervals

The final difference between the two models which we will consider is how feasible schedules can be constructed from the windows. Rigid windows have the following property

Property 1. *For a set of rigid windows $[a_d, b_d]_{d \in D}$, jointly fixing departure times $\{t_d\}_{d \in D}$ such that $t_d \in [a_d, b_d] \; \forall \, d \in D$ will always yield a feasible schedule.*

This is useful, because the path finding algorithm used in our approach leaves the dispatch intervals fixed during the execution of the algorithm. Flexible windows do not have this property. Figure 3.7 shows a schedule in blue, corresponding to the solution graph in Figure 3.2, where every dispatch departs within the flexible windows, but it is not time feasible, because dispatch d_2 departs before d_1 and d_3 arrive.

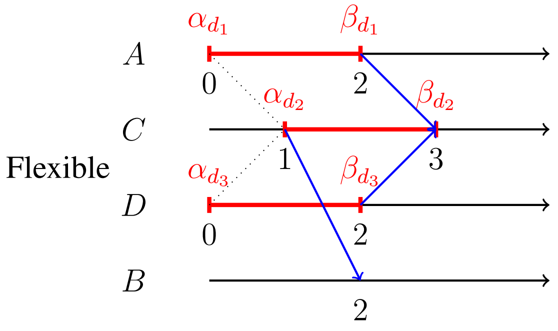


Figure 3.7: An infeasible schedule captured by optimal flexible windows

Flexible windows have the weaker property:

Property 2. *Let $[\alpha_d, \beta_d]_{d \in D}$ be a set of flexible windows. Choose any $\tilde{d} \in D$ and fix $t_{\tilde{d}} \in [\alpha_{\tilde{d}}, \beta_{\tilde{d}}]$. Then there will exist some $t_d \in [\alpha_d, \beta_d] \; \forall \; d \in D \setminus \{\tilde{d}\}$ such that $\{t_d\}_{d \in D}$ is a feasible schedule.*

This weaker property says that we can generate a feasible schedule by sequentially fixing a dispatch departure time and updating the windows in every iteration. However, our path finding algorithm does not update the intervals during the execution of the algorithm. If the path finding algorithm is provided flexible windows, there exist instances where the algorithm will return a time infeasible path. To illustrate this phenomenon we provide the following more complicated example. The geography of the logistics network F is shown in Figure 3.8; assume all travel times are 1, all vehicle costs are 1, and all vehicle capacities are 1.

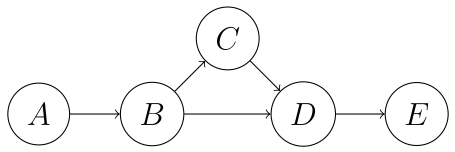


Figure 3.8: Example logistics network

The commodity data is given in Table 3.4. Assume that the commodities are to be pathed in order of k. In the first iteration, commodity 1 will take a direct from B to C at a cost of 1. Call this dispatch d_2. In the second iteration, commodity 2 will take a direct from C to D at a cost of 1. Call this dispatch d_3. In the third iteration, commodity 3 will travel from A to B on a new dispatch d_1, consolidate with commodity 1 to C on d_2, consolidate with commodity 2 to D on d_3, and travel from D to E on a new dispatch d_4 for a total marginal cost of 2.

Table 3.4: Example commodity data

k	o_k	d_k	e_k	l_k	q_k
1	B	C	0	100	0.5
2	C	D	0	100	0.5
3	A	E	0	100	0.5
4	A	E	1	4	0.5

After iteration 3, the existing dispatches are shown in Figure 3.9, and the precedence graph with this solution is shown in Figure 3.10 with optimal flexible dispatch windows shown in blue.

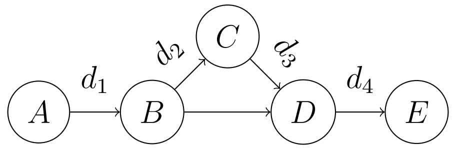


Figure 3.9: Logistics network and existing dispatches after iteration 3

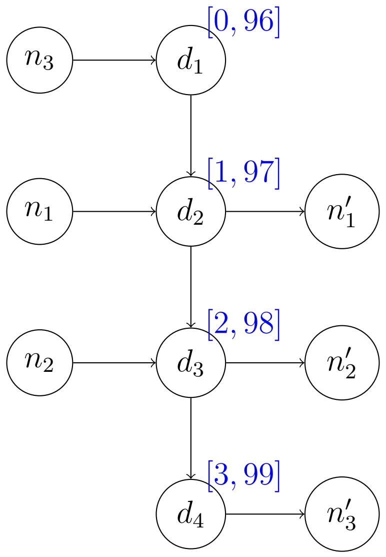


Figure 3.10: Precedence graph and optimal flexible windows after iteration 3

If the path finding algorithm is provided the windows in Figure 3.10, it will return a path (d_1, d_5, d_4) for commodity 4 where d_1 can depart at 1, d_5 is a new dispatch which can depart at 2, and d_4 can depart at 3. Note: the existing dispatches used in this path, d_1 and d_4, can depart within their optimal flexible windows provided in Figure 3.10; however, assigning commodity 4 to this path yields a time infeasible solution. Updating the precedence graph and finding the new optimal flexible windows yields Figure 3.11. These windows are due to the undirected cycle $d_1, d_2, d_3, d_4, d_5, d_1$. Commodity 4 shifted the window of dispatch d_1 ahead to have an earliest release of 1, and shifted the window of dispatch d_4 to have a latest departure of 3. When these changes are propagated through the network, one obtains nonsensical departure intervals.

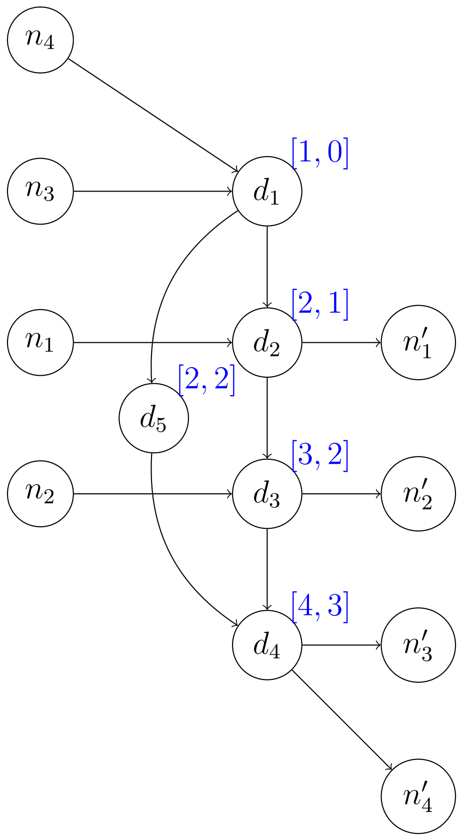


Figure 3.11: Precedence graph and flexible windows after iteration 4

Therefore, to guarantee the path finding algorithm will return a time-feasible path, it must be provided with rigid windows. One could develop an algorithm that updates flexible windows after each reach of the labelling algorithm; however, window updates require solving their own shortest path problems resulting in time inefficiency. Furthermore, a set of windows would need to be maintained for each leaf in the solution tree which could be memory inefficient. Given the heuristic aspect of the approach, we decided to leave the intervals fixed in the path finding algorithm.

Finding optimal dispatch intervals

Based on the results and intuition presented in the previous section, we chose to use the max-sum objective for finding flexible windows, and the max-min objective for rigid windows. Max-sum optimal solutions for the flexible window problem dominate solutions to the max-min problem;

however, for rigid windows, many optimal solutions to the max-sum objective have dispatches with 0-width-intervals. Indeed this is shown in the earlier example in Figure 3.3, and we prefer having the slack partitioned more equally since we can not predict in future iterations which consolidations are more beneficial.

Assuming that there exists a time feasible solution, optimal flexible windows are solutions to

$$\begin{aligned}
\max \sum_{u \in V_D} & \beta_u - \alpha_u \\
\text{s.t.} & \\
& \alpha_u + \tau_u \le \alpha_v && \forall (u,v) \in A[V_D] \\
& \beta_u + \tau_u \le \beta_v && \forall (u,v) \in A[V_D] \\
& e_{k_n} \le \alpha_u && \forall (n,u) \in \delta(E) \\
& \beta_u + \tau_u \le l_{k_n} && \forall (u,n) \in \delta(L)
\end{aligned}$$

. This problem is dual to the intersection of two shortest path problems; α and β are dual labels for independent shortest path problems. Therefore we can solve for windows using a shortest path algorithm on G.

At each iteration, we add an $E - L$ path $p = (n_1, ..., n_m)$ (containing new and existing vertices and arcs) to G. We do not need to solve the above problem from scratch at every iteration. For the release vertex $n_1 \in E$, we initialize the window to $[e_{k_{n_1}}, \infty]$ where $e_{k_{n_1}}$ is the release time of the commodity corresponding to n_1. For a commodity destination vertex n_m, we initialize the window to $[-\infty, l_{k_{n_m}}]$ where $l_{k_{n_m}}$ is the due time of the commodity corresponding to n_m. Finally, for dispatch vertices, we initialize the window to $[-\infty, \infty]$. We initialize the frontier to the set of nodes in the new path, and propagate the labels changes. Algorithm 12 gives this procedure.

Algorithm 12 α, β Updates (Constructive)

```
1:  procedure α-UPDATE(New $E - L$ path $p = (n_1, ..., n_m)$)
2:      $U \leftarrow p$
3:      while $U \neq \emptyset$ do
4:          $u \leftarrow U$.pop()
5:          for $v \in \delta^+(u)$ do
6:              if $\alpha_v < \alpha_u + \tau_u$ then
7:                  $\alpha_v \leftarrow \alpha_u + \tau_u$
8:                  $U$.push($v$)
9:  procedure β-UPDATE(New $E - L$ path $p = (n_1, ..., n_m)$)
10:     $U \leftarrow p$
11:     while $U \neq \emptyset$ do
12:         $v \leftarrow U$.pop()
13:         for $u \in \delta^-(v)$ do
14:             if $\beta_u + \tau_u > \beta_v$ then
15:                 $\beta_u \leftarrow \beta_v - \tau_u$
16:                 $U$.push($u$)
```

Finding optimal rigid windows is not as simple. Again, assuming there exists a time feasible solution, optimal rigid windows are solutions to

$$\begin{aligned} &\max z \\ &\quad \text{s.t.} \\ &\qquad z \leq b_u - a_u && \forall u \in V_D \\ &\qquad b_u + \tau_u \leq a_v && \forall (u,v) \in A[V_D] \\ &\qquad e_{k_n} \leq a_u && \forall (n,u) \in \delta(E) \\ &\qquad b_u + \tau_u \leq l_{k_n} && \forall (u,n) \in \delta(L) \end{aligned}$$

.

Define $\gamma_u = b_u - a_u$ to be the dispatch interval width for $u \in V_D$. We rewrite with variables a_u and γ_u, because, with these variables, it is easier to see why the dual is equivalent to solving a minimum mean cost $E - L$ path problem.

$$\begin{aligned}
&\max z \\
&\text{s.t. } a_u + \gamma_u + \tau_u \leq a_v && \forall (u,v) \in A[V_D] && (3.15)\\
&e_{k_n} \leq a_u && \forall (n,u) \in \delta(E) && (3.16)\\
&a_u + \gamma_u + \tau_u \leq l_{k_n} && \forall (u,n) \in \delta(L) && (3.17)\\
&z \leq \gamma_u && \forall u \in V_D && (3.18)\\
&\gamma_u \geq 0 && \forall u \in V_D
\end{aligned}$$

Associating dual variables x with constraints 3.15, q with 3.16, r with 3.17, and y with 3.18, the dual problem is:

$$\min \sum_{(n,u)\in\delta(E)} (-e_{k_n}) q_{nu} + \sum_{(u,v)\in A[V_D]} (-\tau_u) x_{uv} + \sum_{(u,n)\in\delta(L)} (l_{k_n} - \tau_u) r_{un}$$

$$\text{s.t} \sum_{u \in V_D} y_u = 1 \qquad (3.19)$$

$$- \sum_{(n,u)\in\delta^-_{E-V_D}(u)} q_{nu} - \sum_{(v,u)\in\delta^-_{V_D}(u)} x_{vu} + \sum_{(u,v)\in\delta^+_{V_D}(u)} x_{uv} + \sum_{(u,n)\in\delta^+_{V_D-L}(u)} r_{un} = 0 \quad \forall u \in V_D \qquad (3.20)$$

$$- y_u + \sum_{(u,v)\in\delta^+_{V_D}(u)} x_{uv} + \sum_{(u,n)\in\delta^+_{V_D-L}(u)} r_{un} \geq 0 \qquad \forall u \in V_D \qquad (3.21)$$

$$x, q, r, y \geq 0$$

The x, q, and r variables act as flow variables; q and r are just flow variables leaving E and entering L respectively. One can view the y variables as supplying the network with one unit of supply, which can be propagated on a node's forward star according to 3.21. Constraints 3.20 act as flow balance, but only on V_D. Any nodes u having $y_u > 0$ must push the flow in its forward star and reverse star until the flow is absorbed by nodes in E and L. Any node u with flow on its forward star must also have $y_u > 0$.

Therefore, we intuit that extreme points of this feasible region will have an $E - L$ path $p = (n_1, ..., n_m)$ with $y_u = \frac{1}{m-2} \; \forall \, u \in P \cap V_D$, and hence the x, q, and r variables for the arcs of p will also take the value $\frac{1}{m-2}$. It appears that the problem is to find the $E - L$ path of minimum mean cost according to the dual objective. Furthermore, from complementary slackness we know that $y_u > 0 \Leftrightarrow z = \gamma_u$, so the vertices on such a path could be precisely those whose window widths are equal to the max-min objective. This intuition agrees with the idea that finding optimal rigid windows is to find an optimal partition of slack along a path. The objective function of the dual is accumulating the slack γ_p (see equation 3.14) along an $E - L$ path p, and finds the path whose slack, when partitioned evenly among the dispatches in the path, is minimized.

We begin by proving the following

Proposition 2. *In any optimal solution to the primal, there will exist an $E - L$ path $\tilde{p} = (n_1, ..., n_m)$ having*

$$
\begin{aligned}
z^* &= b^*_{n_i} - a^*_{n_i} && i = 2, ..., m-1 \\
e_{n_1} &= a^*_{n_2} \\
l_{n_m} &= b^*_{n_{m-1}} + \tau_{n_{m-1}} \\
a^*_{n_i+1} &= b^*_{n_i} + \tau_{n_i} && i = 2, ..., m-2
\end{aligned}
$$

To prove this we give the following lemma:

Lemma 1. *Take an optimal solution such that $M = \{u \in V_D : b^*_u - a^*_u = z^*\}$ is minimal.*

(a) For every $u \in M$ there exists $(u, v) \in \delta^+(u)$ such that either $v \in M$ or $v \in L$.

(b) *For every $u \in M$ there exists $(v, u) \in \delta^-(u)$ such that either $v \in M$ or $v \in E$.*

(c) *There exists an $E - L$ path $(n_1, ..., n_m)$ having*

$$n_i \in M \qquad i = 2, ..., n-1 \tag{3.22}$$

$$e_{n_1} = a^*_{n_2} \tag{3.23}$$

$$l_{n_m} = b^*_{n_{m-1}} + \tau_{n_{m-1}} \tag{3.24}$$

$$a^*_{n_i+1} = b^*_{n_i} + \tau_{n_i} \qquad i = 2, ..., m-2 \tag{3.25}$$

Proof. (a) Suppose for contradiction there exists a vertex $u \in M$ such that $\forall\ (u, v) \in \delta^+(u)$ $v \notin M$ and $v \notin L$. Then $\forall\ (u, v) \in \delta^+(u)$ $v \in V_D$ and $b_v - a_v > b_u - a_u$. Let $\delta = \min_{(u,v)\in\delta^+(u)}\{b_v - a_v\} - (b_u - a_u)$.

Taking

$$a'_n = \begin{cases} a_n + \frac{\delta}{2} & \forall n : \exists (u, n) \in \delta^+(u) \\ a_n & \text{otherwise} \end{cases}$$

and

$$b'_n = \begin{cases} b_n + \frac{\delta}{2} & \text{if } n = u \\ b_n & \text{otherwise} \end{cases}$$

yields a feasible solution with $b'_u - a'_u > z^*$ and $b'_v - a'_v > z^*\ \forall\ (u, v) \in \delta^+(u)$. In the case where u was the single vertex achieving the optimal, we have increased the optimal objective by $\frac{\delta}{2}$, a contradiction. Otherwise, we have reduced the number of vertices in M, contradicting our assumption that M is minimal.

(b) Similar to (a).

(c) The existence of an $E - L$ path $(n_1, ..., n_m)$ having $n_i \in M$ for $i = 2, .., m-1$ follows from (a) and (b). If there are multiple such paths, let $(n_1, ..., n_m)$ be the path such that n_2

has no upstream neighbour in M and n_{m-1} has no downstream neighbour in M. That is, $\nexists(u, n_2) \in \delta^-(n_2) : u \in M$ and $\nexists(n_{m-1}, v) \in \delta^+(n_{m-1}) : v \in M$.

Since n_{m-1} has no downstream neighbour in M, if it does not have a neighbour $v \in L$ such that $l_v = b^*_{n_{m-1}} + \tau_{n_{m-1}}$, then its interval width can be extended contradicting the minimality of M as in the proof of (a). Relabel n_m to be this vertex v if necessary. Similarly, since n_2 has no upstream neighbour in M, it must have an upstream neighbour $u \in E$ with $e_u = a^*_{n_2}$.

We have shown that there exists an $E - L$ path $(n_1, ..., n_m)$ satisfying 3.22-3.24. Suppose for contradiction that no path satisfying 3.22-3.24 also satisfies 3.25. Take an $E - L$ path $(n_1, ..., n_m)$ satisfying 3.22-3.24. Starting at $n \leftarrow n_2$, move to neighbours in $n' \in \delta^{out}(n)$ where $n' \in L$ and $l_{n'} = b_n + \tau_n$ or $n' \in M$ and $a_{n'} = b_n + \tau_n$ until no such neighbour exists. If we reach $n \in L$ we have produced a path satisfying 3.22-3.25. If we have not, we have a node $n \in M$ such that there exists $\delta > 0$ such that $a_{n'} \geq b_n + \tau_n + \delta \; \forall \; (n, n') \in \delta^{out}(n) : n' \in M$ and $l_{n'} \geq b_n + \tau_n + \delta \; \forall \; (n, n') \in \delta^{out}(n) : n' \in L$, and $b_{n'} - a_{n'} - \delta \geq z^* \; \forall \; (n, n') \in \delta^{out}(n) : n' \notin M, n' \notin L$. It follows that we can extend the interval of n as above contradicting the minimality of M.

$\square$

Proposition 2 follows from Lemma 1, because such a path is contained in the set M for every minimal set M. Expanding M by adding new vertices preserves the existence of the path.

Proposition 3. *Consider an optimal solution to the primal, and let $\tilde{p} = (n_1, n_2, ..., n_m)$ be an $E - L$ path having the properties from Proposition 2. Recall the slack of $\tilde{p}$ is*

$$\gamma_{\tilde{P}} = (l_{n_m} - e_{n_1}) - \sum_{i=2}^{m-1} \tau_{n_i}$$

.

Then

$$b^*_{n_i} - a^*_{n_i} = \frac{\gamma_{\tilde{p}}}{m-2},\ i = 2, .., m-1$$

Proof.

$$\sum_{i=2}^{m-1} b^*_{n_i} - a^*_{n_i} = (m-2)(b^*_{n_j} - a^*_{n_j}) \qquad j = 2, .., m-1$$

Then for $j = 2, ..., m-1$

$$(m-2)(b^*_{n_j} - a^*_{n_j}) = \sum_{i=2}^{m-1} b^*_{n_i} - a^*_{n_i} \tag{3.26}$$

$$= \sum_{i=2}^{m-2} b^*_{n_i} + l_{n_m} - \tau_{n_{m-1}} - \sum_{i=3}^{m-1} a^*_{n_i} - e_{n_1} \tag{3.27}$$

$$= l_{n_m} - e_{n_1} - \tau_{n_{m-1}} - \sum_{i=2}^{m-2} (a^*_{n_{i+1}} - b^*_{n_i}) \tag{3.28}$$

$$= (l_{n_m} - e_{n_1}) - \sum_{i=2}^{m-1} \tau_{n_i} \tag{3.29}$$

□

We have shown that, in any optimal solution, the max-min rigid window width is achieved at each node along some $E - L$ path $\tilde{p}$, and that the optimal solution value is equivalent to the slack of the path averaged over the dispatch vertices in the path. We have not shown that the path $\tilde{p}$ is the minimum mean cost path with costs accumulating slack (as in the dual problem); however, we guess that this is the case, propose an algorithm for finding the minimum mean cost path, then prove the algorithm is correct.

Since the precedence graph G is directed acyclic, we can adapt labelling algorithms for finding minimum mean cost cycles [32] to solve our problem. We define costs on the arcs of the precedence graph that accumulate slack:

$$C_{uv} = \begin{cases} -e_{k_u} & \text{if } (u,v) \in \delta(E) \\ -\tau_u & \text{if } u \in A[D] \\ -\tau_u + l_{k_v} & \text{if } (u,v) \in \delta(L) \end{cases}$$

.

For any $E - L$ path $\tilde{p}$

$$\sum_{a \in A[\tilde{p}]} C_a = \gamma_{\tilde{p}}$$

.

We use a Bellman-Ford style algorithm to find labels $d_i(u)$ representing the minimum cost $E - u$ path containing exactly i arcs. For an overview of the Bellman-Ford algorithm with correctness and complexity results see [33]. Then the minimum mean cost $E - L$ path can be found by finding the label achieving $\min_{v \in L, m \in \{1,\ldots,n\}}\{\frac{d_m(v)}{m-1}\}$. Note $m - 1$ is in the denominator, because for an $E - L$ path having m arcs, there are $m - 1$ vertices in the path that lie in V_d which slack must be allocated over.

Algorithm 13 Window Labeling

```
1: function GenerateLabels($G = (V, A = E \sqcup D \sqcup L)$)
2:     $d_0(u) \leftarrow \begin{cases} 0 & \text{if } u \in E \\ \infty & \text{otherwise} \end{cases} \ \forall\, u \in V$
3:     for $i = 1, \ldots, |V|$ do
4:         for $v \in V$ do
5:             $d_i(v) \leftarrow \begin{cases} \infty & \text{if } d_{i-1}(u) = \infty \forall (u,v) \in A \\ \min_{(u,v) \in A}\{d_{i-1}(u) + C_{uv}\} & \text{otherwise} \end{cases}$
6:     $Opt \leftarrow \min_{v \in L, m \in \{1,\ldots,n\}}\{\frac{d_m(v)}{m-1}\}$
7:     return $\{d_i(v)\}_{v \in V, i=1,\ldots,|V|}$
```

From the shortest path labels, there is a convenient way to recover feasible windows for all vertices, not just those on the path having the optimal window width. Suppose for some $u \in V_D$

we have already fixed the value b_u. The value $d_m(u) + b_u$ represents the minimum slack of any $E - u$ path containing exactly m arcs. Then the value $\min_{m=1,2,..,n}\{\frac{d_m(u)+b_u}{m}\}$ is the minimum mean slack of any $E - u$ path for any $u \in V_d$. Note the denominator is m, because for any $E - u$ path with $u \in V_d$ having m arcs, there are m vertices on the path that lie in V_d. We assign γ_u, the width of the interval of u, this value. Algorithm 14 works by considering nodes whose downstream neighbours have their windows fixed (i.e. their downstream neighbours are contained in F), and working backwards along G. Once all downstream neighbours of a vertex u have fixed windows, the latest departure time b_u is known. We assign γ_u, fix $a_u = b_u - \gamma_u$, add u to F and update the b_x for x in the reverse start of u. Note that it may be possible to extend the widths of intervals of some dispatches, but we show the algorithm is correct and produces an optimal solution to the max-min rigid window problem.

Algorithm 14 Recover Feasible Windows

```
1: procedure RecoverWindows
2:     $b_u \leftarrow \infty \; \forall \, u \in V[P]$
3:     $b_u \leftarrow l_u \; \forall \, u \in L$
4:     $U \leftarrow L$
5:     $F \leftarrow \emptyset$
6:     while $U \neq \emptyset$ do
7:         Take $u \in U$
8:         $U \leftarrow U \setminus u$
9:         $\gamma_u \leftarrow \begin{cases} \min_{m=1,2,..,n}\{\frac{d_m(u)+b_u}{m}\} & \text{if } u \in V[D] \\ 0 & \text{otherwise} \end{cases}$
10:        $a_u \leftarrow b_u - \gamma_u$
11:        $F \leftarrow F \cup u$
12:        for $(x, u) \in \delta^{in}(u)$ do
13:            $b_x \leftarrow \min\{b_x, a_u - \tau_x\}$
14:            if $N^{out}(x) \subseteq F]$ then
15:                $U \leftarrow U \cup x$
```

Proposition 4. *Algorithm 14 yields* $\{(a_u, \gamma_u, b_u)\}_{u \in V_D}$ *satisfying*

$$a_u + \gamma_u + \tau_u \leq a_v \qquad \forall (u,v) \in A[V_D] \tag{3.30}$$

$$b_u + \tau_u \leq l_k \qquad \forall (u,k) \in \delta(L) \tag{3.31}$$

$$e_k \leq a_u \qquad \forall (k,u) \in \delta(E) \tag{3.32}$$

Proof. Constraints 3.30 hold since line 13 of Algorithm 14 guarantees

$$b_u = \min_{(u,v) \in \delta^+(u)} \{a_v - \tau_u\}$$

.

Constraints 3.31 hold since, for all $k \in L$, $a_k = l_k$, thus line 13 guarantees

$$b_u = \min_{(u,k) \in \delta^+(u) \cap \delta(L)} \{l_k - \tau_u\}$$

.

Consider any node $u \in V_D$. γ_u is chosen such that, for any $E-u$ path $p = (n_1, n_2, ..., n_m, n_{m+1} = u)$

$$\gamma_u \leq \frac{b_u - e_{n_1} - \sum_{i=2}^{m} \tau_i}{m}$$

. Rearranging,

$$e_{n_1} \leq b_u - m\gamma_u - \sum_{i=2}^{m} \tau_i$$

Then restricting to paths that consist of a single arc we have

$$e_k \leq b_u - \gamma_u = a_u$$

□

Proposition 5. *Let γ^* be optimal objective of the minimum mean cost path problem. Then Algorithm 14 yields $\{\gamma_u\}_{u\in V_D}$ satisfying*

$$\gamma_u \geq \gamma^* \; \forall u \in V_D$$

Lemma 2. *For any $(u, v) \in A[V_D]$, after Algorithm 14 terminates, if $b_u = a_v - \tau_u$, then $\gamma_u \geq \gamma_v$.*

Proof. Fix $v \in V_D$. Note that for some particular $m*$, the algorithm assigns γ_v as $\gamma_v = \frac{d_{m^*}(v)+b_v}{m^*}$ for $m^* \in \{1, 2, ..., |V[G]|\}$.

Note that for any $m \in 1, ..., |V[G]|$

$$d_{m+1}(v) \leq d_m(u) - \tau_u$$

and

$$\gamma_v \leq \frac{d_m(v) + b_v}{m} \; \forall m \in \{1, 2, .., |V[G]|\} \Rightarrow d_{m+1}(v) + b_v \geq (m+1)\gamma_v \; \forall m \in \{1, .., |V[G]| - 1\}$$

Assume $b_u = a_v - \tau_u = b_v - \gamma_v - \tau_u$. For any $m = 1, 2, .., |V[G]| - 1$ we have

$$\begin{aligned} \frac{d_m(u) + b_u}{m} &= \frac{d_m(u) + b_v - \gamma_v - \tau_u}{m} \\ &\geq \frac{d_{m+1}(v) + b_v - \gamma_v}{m} \\ &\geq \frac{(m+1)\gamma_v - \gamma_v}{m} \\ &= \gamma_v \end{aligned}$$

Hence since γ_u is achieved at some $m \in \{1, 2, .., |V[G]| - 1\}$, $\gamma_u \geq \gamma_v$. □

The following is a proof of Proposition 5:

Proof. From any node $u \in V_D$ there is a path $(n_1 = u, n_2, .., n_m)$ with $n_m \in L$ and $b_{n_i} = a_{n_{i+1}} - \tau_{n_i}$

for $i = 1, .., m-1$. Then applying Lemma 2 we have $\gamma_u \geq \gamma_{n_2} \geq ... \geq \gamma_{n_k}$. Since $\gamma_x \geq \gamma^*$ for all $x \in L$, we conclude that $\gamma_u \geq \gamma^*$ for all $u \in V_D$. □

At this point, we have shown that our algorithm produces a feasible solution to the rigid window problem. The next result proves this solution is also optimal.

Proposition 6. *Let z^* be the optimal solution to the max-min rigid window problem. Then Algorithm 14 yields $\{\gamma_u\}_{u \in V_D}$ satisfying*

$$\gamma_u \geq z^* \; \forall u \in V_D$$

Proof. For any $E - L$ path $(n_1, n_2, ..., n_m)$, aggregating constraints in the max-min rigid window problem yields
$$e_{k_{n_1}} + \sum_{i=2}^{m-1} \gamma_{n_i} + \sum_{i=2}^{m-1} \tau_{n_i} \leq l_{k_{n_m}}$$
. For the minimum mean cost path $(n_1, n_2, ..., n_m)$ we have, by construction,

$$\begin{aligned} e_{k_{n_1}} + \sum_{i=2}^{m-1} \gamma_{n_i} + \sum_{i=2}^{m-1} \tau_{n_i} &= e_{k_{n_1}} + (m-2)\gamma^* + \sum_{i=2}^{m-1} \tau_{n_i} \\ &= l_{k_{n_m}} \end{aligned}$$

Suppose for contradiction that $z^* > \gamma^*$, then

$$\begin{aligned} e_{k_{n_1}} + \sum_{i=2}^{m-1} \gamma_{n_i} + \sum_{i=2}^{m-1} \tau_{n_i} &\geq e_{k_{n_1}} + (m-2)z^* + \sum_{i=2}^{m-1} \tau_{n_i} \\ &> e_{k_{n_1}} + (m-2)\gamma^* + \sum_{i=2}^{m-1} \tau_{n_i} \\ &= l_{k_{n_m}} \end{aligned}$$

Then there is no feasible solution. So $z^* \leq \gamma^*$. The result follows from Proposition 5: Algorithm

14 produces solutions with $\gamma_u \geq \gamma^* \geq z^* \; \forall \; u \in V_D$. $\square$

3.5 Computational Results

3.5.1 Synthetic Instance Generation

An instance generator was developed to test the performance of the algorithm on instances of different sizes. The main parameters of the instances are

1. The earliest date and time at which commodities can arrive to the system $\underline{t}$
2. The latest date and time at which commodities can arrive to the system $\bar{t}$
3. The number of vehicle types n_V
4. The dimensions of the rectangular map on which terminals are placed
5. The number of regions in the map
6. The number of terminals in the map
7. The number of hubs in each region
8. The number of commodities

The topology of the flat network was developed as follows. Each terminal was assigned random coordinates following a uniform distribution across the entire rectangular map. The terminals were then clustered into regions using k-means, with k being the given number of regions. Terminals were randomly chosen to be hubs, with weights proportional to the inverse of the distance between the terminal and the mean of the coordinates of all terminals. This was done to bias hubs towards connecting regions. The terminals are the nodes of the flat network. The set of arcs consists of the union of all pairs of terminals in the same region and all pairs of hubs. That is, the hubs induce a complete graph and all terminals in any region induce a complete graph. The travel time between terminals u and v was set to be $\tau_{uv} = \lceil 10 d_{uv} \rceil$ minutes, where d_{uv} is the euclidean distance between the coordinates of u and v.

Given the number of vehicles n_V, we set $V = [n_V]$, $Q_v = 50 + 50v$, and $C_{av} = \sqrt{Q_v}\tau_a \; \forall \; a \in A, v \in V$.

The origin and destination of every commodity were chosen randomly with equal probability for any terminal. The weight of every commodity q_k was chosen random according to a $Unif[1, 20]$ distribution. The earliest release time of each commodity, e_k, was chosen uniformly in $[\underline{t}, \overline{t}]$. Let $\tilde{\tau}_{o_k d_k}$ represent the minimum travel time between the origin and destination of commodity k. The due time of every commodity was set to be $l_k \leftarrow e_k + \tilde{\tau}_{o_k d_k}(1 + Unif[0.05, 0.25])$.

Comparison to industry instance

The structure of instances can be effectively manipulated to exaggerate the performance of certain approaches. Therefore, we demonstrate that our synthetically generated instances are comparable to our industry instance across a variety of metrics. To do this, we compare the industry instance to a synthetic instance of similar size. The high level characteristics of these instances are shown in Table 3.5.

Table 3.5: Instance properties

Property	Industry	Synthetic
Num. Commodities	100,281	100,000
Total Weight	19,549,964	1,051,715
Number of Locations	69	70
Number of Lanes	2,507	2,078

We first show in Figure 3.12 that the economies of scale in vehicle pricing has the desired structure. Instances having larger vehicles that are less cost efficient are easier to solve, because the larger vehicles will never be used and hence could be excluded from the problem.

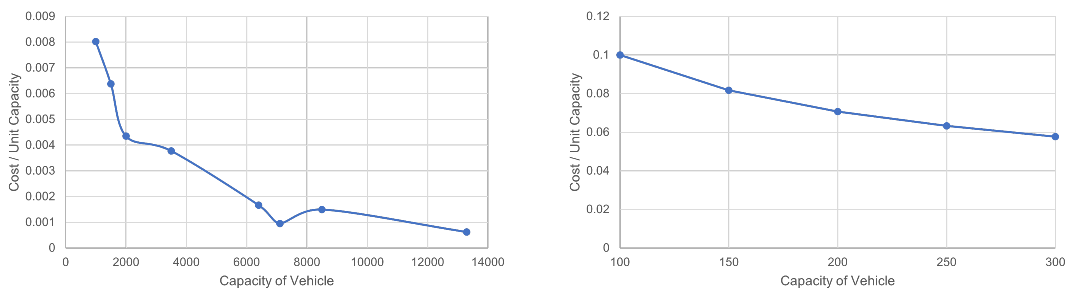


Figure 3.12: Cost per unit capacity for industry (left) and synthetic (right) instances

Intuitively, an instance having fewer timed paths available for each commodity will be easier to solve. One metric affecting this is the commodity slack, taking value $l_k - e_k - \underline{\tau}_k$ where $\underline{\tau}_k$ is the duration of the minimum travel time path from o_k to d_k. As commodity slack decreases, fewer time feasible paths exist and the instance becomes easier to solve. Similarly, if the degree of the flat network $F = (B, A)$ is reduced, the problem becomes easier. Combining the effects of slack time a geographic degree, we can also measure and compare the number of buildings $b \in B$ that each commodity can visit during $[e_k, l_k]$ and still make service. These metrics are shown in Table 3.6 for the industry instance and Table 3.7 for the synthetic instance. Among the properties we believe impact difficulty, the instances are sufficiently similar for comparison.

Table 3.6: Industry instance metrics

Metric	Min	Avg	Max
Commodity Weight	0.1	194.95	18021.5
Commodity Slack	0	395.43	4,205
Arc Travel Time	0	311.83	1,284
Out-Degree of F	0	36.33	56
Shortest Path Time ($\underline{\tau}_k$)	0	286.18	1,686
Originating Flow	0	283,332.80	1,812,445.43
Num. Buildings Reachable	2	33.14	66

Table 3.7: Synthetic instance metrics

Metric	Min	Avg	Max
Commodity Weight	1	10.52	20
Commodity Slack	1	452.02	3,649
Arc Travel Time	5	425.56	1,230
Out-Degree of F	8	29.69	69
Shortest Path Time ($\underline{\tau}_k$)	5	573.15	1,226
Originating Flow	0	15,024.50	16,149
Num. Buildings Reachable	2	29.64	70

3.5.2 Rigid-Only Results

To guarantee we obtain a time-feasible path, we can not exclusively rely on flexible windows. A simple first implementation is to exclusively use rigid intervals and ignore flexible intervals entirely. Although this excludes feasible consolidations from being considered and although the algorithm for updating rigid windows takes longer, we are guaranteed to be provided with a time-feasible path. An initial construction algorithm was implemented using only the rigid windows. The pseudocode is given in Algorithm 15. Note this implementation solves for the rigid windows from scratch at each iteration. It does not save Bellman labels from the previous iteration.

Algorithm 15 Initial Test Algorithm

```
1: procedure RIGIDONLY($K$)
2:     $G \leftarrow \emptyset$                                ▷ Precedence Graph
3:     $\mathcal{L} \leftarrow \emptyset$                                ▷ Labels
4:     $W \leftarrow \emptyset$                                ▷ Windows
5:     for $k \in K$ do
6:         $p \leftarrow$FindPath$(k, W)$
7:         AddPath$(G, p)$
8:         $\mathcal{L} \leftarrow \emptyset$
9:         $\mathcal{L} \leftarrow$GenerateLabels$(G)$
10:        $W \leftarrow \emptyset$
11:        $W \leftarrow$RecoverWindows$(\mathcal{L}, G)$
```

Table 3.8: Discretized vs Discretizationless construction performance

$\lvert R\rvert, \lvert B\rvert, \lvert H\rvert, \lvert K\rvert$	Discless Cost	Discless Time (s)	Discretized Cost	Discretized Time (s)	Cost Imp %
3,25,3,5000	5,935,030	49.77	6,795,270	0.77	12.66
5,50,3,5000	7,134,320	66.19	7,807,630	1.81	8.62
3,25,3,10000	8,805,150	237.50	10,008,400	1.76	12.02
5,50,3,10000	11,226,900	318.46	12,412,100	3.32	9.55
3,25,3,15000	9,174,220	615.41	10,124,300	3.51	9.38
5,50,3,15000	14,380,600	791.29	15,570,000	5.46	7.64
3,25,3,30000	17,133,200	2,548.02	18,311,000	7.10	6.43
5,50,3,30000	19,639,600	3,733.90	20,977,400	15.15	6.38
3,25,3,50000	22,882,500	7,598.26	24,111,800	12.82	5.10
5,50,3,50000	28,256,700	11,045.30	30,059,400	35.29	6.00

We denote the rigid-only algorithm as “Discless”. We benchmark against a sequential marginal cost path heuristic on a fixed time discretization with heuristic dynamic discretization discovery, denoted “Discretized”. From a solution perspective, such an approach is equivalent to planning with rigid windows of width zero, but avoids the expanded state space in the path finding algorithm and avoids spending time updating windows. However, it requires dynamically inserting new time points if the mapping error yields a time infeasibility. Here we are comparing only the construction phase of the approaches; no local search improvement is performed.

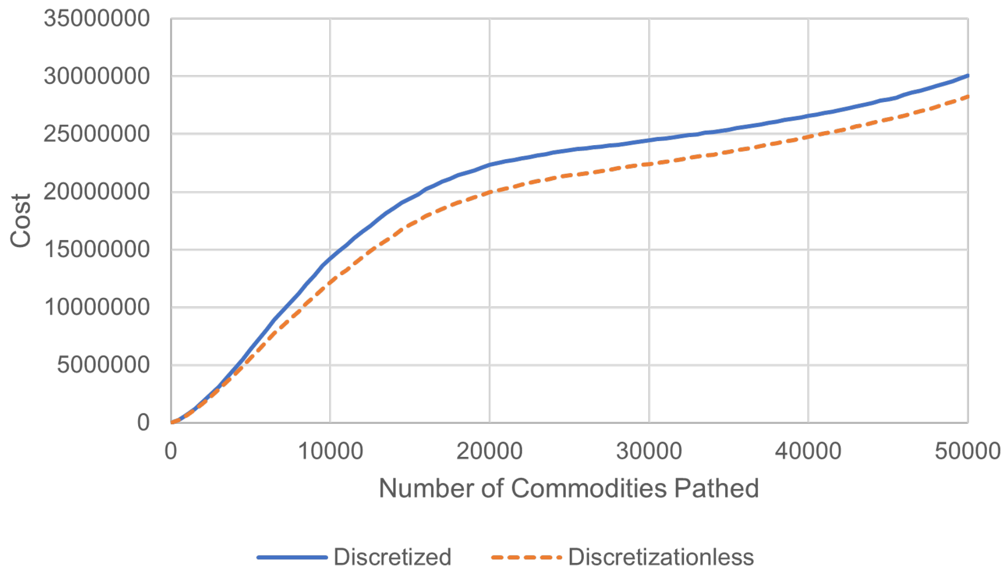


Figure 3.13: Cost vs iteration for the 5,50,3,50000 instance

We can see that the rigid window approach outperforms MCPH in terms of solution quality, but the solution time is orders of magnitude larger than MCPH. Furthermore, this solution time gap grows rapidly with $|K|$.

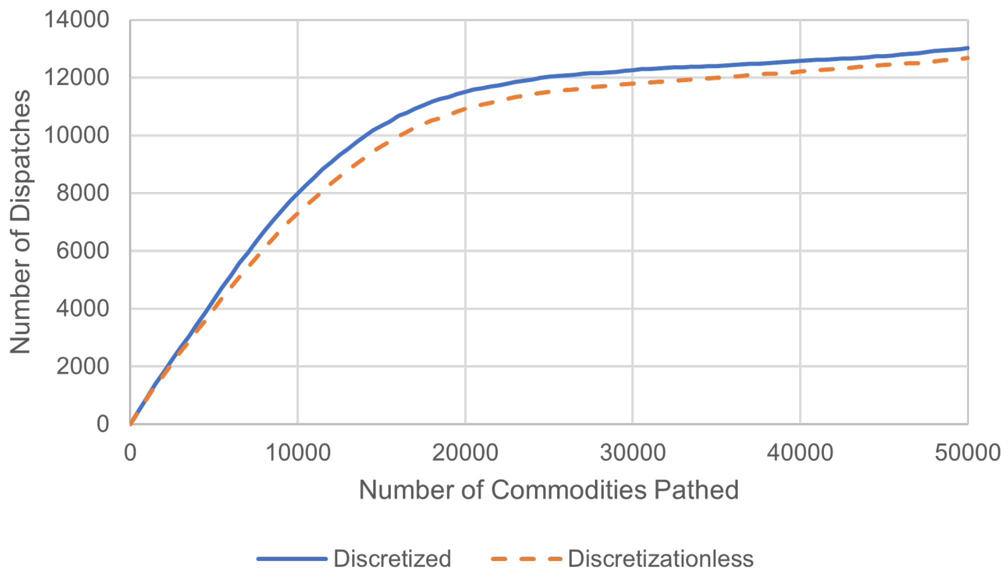


Figure 3.14: Number of dispatches vs iteration for the 5,50,3,50000 instance

Examining the progress over iterations, we see the bulk of the gains in solution cost were made early in the execution, suggesting that MCPH could be suffering from a lack of flexibility, and that this downside dwindles as the entire network is populated with commodities.

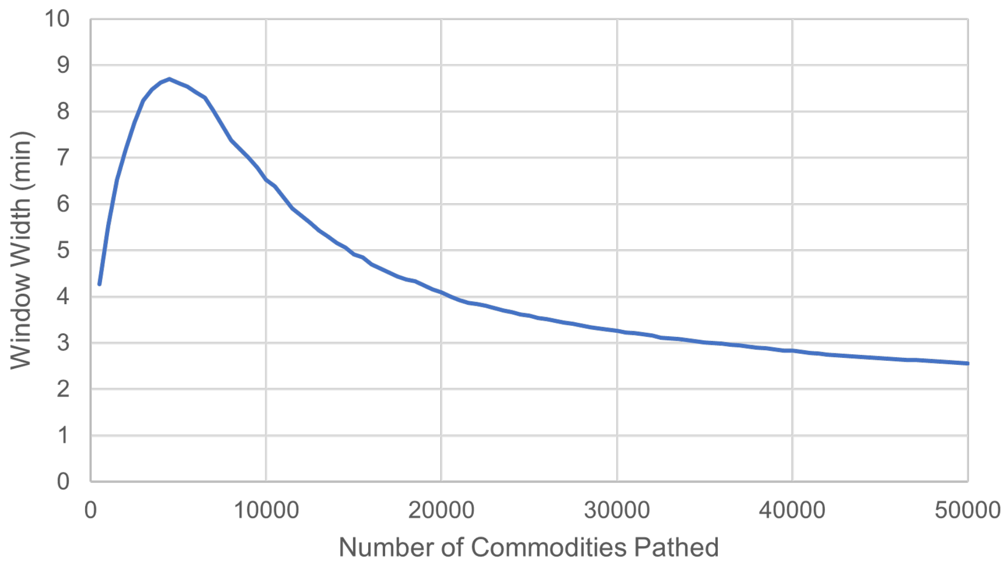


Figure 3.15: Average window width vs iteration for the 5,50,3,50000 instance

We see the average window width grows for a time before tailing off. The commodities are added in ascending order of slack time, so this behaviour makes sense. Later on, commodities squeeze in where they can, driving windows smaller as they add additional timing constraints.

3.5.3 Enhancing the Algorithm

Because the solution time of the discretizationless approach is so poor, we consider two improvements designed to boost efficiency. The first aims to use flexible windows wherever possible and use rigid windows only when necessary. The second aims to reduce the time spent finding rigid windows by preserving Bellman labels between path iterations. Based on the structural changes of the precedence graph, only a subset of the Bellman labels change between path finding iterations.

Defaulting to Flexible Windows

We would like to drive the solution time lower. Updating flexible windows is much more time efficient. We can attempt to find a minimum marginal cost path using the flexible windows. If adding it to the current solution is feasible, we do so; otherwise, we find a minimum marginal cost path using the rigid windows.

The method of detecting feasibility of the solution can work as follows: we maintain two sets of windows, new and old. While updating the new windows, if we find some window has $\beta_d^{new} < \alpha_d^{new}$, we declare the solution infeasible, revert to the old windows, and find a path using the rigid windows. The algorithmic changes are shown in Algorithm 16.

Algorithm 16 New Flexible Window Update

```
procedure $\alpha$-UPDATE(List of precedence vertices of the new path $X$)
    $U \leftarrow X$
    while $U \neq \emptyset$ do
        $u \leftarrow U$.pop()
        for $v \in \delta^+(u)$ do
            if $\alpha_v^{new} < \alpha_u^{new} + \tau_u$ then
                $\alpha_v^{new} \leftarrow \alpha_u^{new} + \tau_u$
                $U$.push($v$)
function $\beta$-UPDATE(List of precedence vertices of the new path $X$)
    $U \leftarrow X$
    while $U \neq \emptyset$ do
        $v \leftarrow U$.pop()
        for $u \in \delta^-(v)$ do
            if $\beta_u^{new} + \tau_u > \beta_v^{new}$ then
                $\beta_u^{new} \leftarrow \beta_v^{new} - \tau_u$
                $U$.push($u$)
                if $\beta_u^{new} < \alpha_u^{new}$ then
                    return False                                ▷ Adding this path is not feasible
    return True                                                 ▷ Adding this path is feasible
procedure FLEXWITHRIGID($K$)
    $G \leftarrow \emptyset$                                              ▷ Precedence Graph
    $\mathcal{L} \leftarrow \emptyset$                                              ▷ Labels
    $W \leftarrow \emptyset$                                              ▷ Rigid Windows
    for $k \in K$ do
        $p \leftarrow$FindPath($k, \alpha^{old}, \beta^{old}$)
        AddPath($G, p$)
        $\alpha$-Update($p$)
        $Feas \leftarrow \beta$-Update($p$)
        if $Feas$ then
            $(\alpha^{old}, \beta^{old}) \leftarrow (\alpha^{new}, \beta^{new})$
        else
            $(\alpha^{new}, \beta^{new}) \leftarrow (\alpha^{old}, \beta^{old})$
            RemovePath($G, p$)
            $\mathcal{L} \leftarrow \emptyset$
            $\mathcal{L} \leftarrow$GenerateLabels($G$)
            $W \leftarrow \emptyset$
            $W \leftarrow$RecoverWindows($\mathcal{L}, G$)
            $p \leftarrow$FindPath($k, W$)
            AddPath($G, p$)
            $\alpha$-Update($p$)
            $\beta$-Update($p$)
```

Maintaining Rigid Window Labels Between Iterations

Suppose we would like to reduce the amount of computation spent generating rigid windows by maintaining as much of the label table as possible between iterations. We maintain a set of queues $Q_0, Q_1, .., Q_{|V|}$ which will denote vertices from which we must reach to update the labels. The subscript i of Q_i denotes the column of the label table from which we are reaching. In ascending order of i, we will process nodes in $u \in Q_i$ by checking if $d_i(u) + C_{uv} < d_{i+1}(v)$ for $v \in \delta^{out}(u)$. If so then we update $d_{i+1}(v)$ and insert v into Q_{i+1}.

We must initialize $\{Q_i\}$ with the nodes being added. Let V^{new} denote the set of new vertices in the precedence graph, and V_E^{new} denote the new vertices in E. We initialize $Q_0 \leftarrow V_E^{new}$.

Let A^{new} denote the set of new arcs. We claim that we need only insert, into the appropriate $\{Q_i\}$, tail vertices of A^{new} which are also contained in V^{old}. If the tail u of $(u, v) \in A^{new}$ is in V^{new} we will eventually consider the reach updates along (u, v), because there is either an $V_E^{new} - u$ path comprised of all new vertices, or there is a $x - u$ path with $x \in V^{old}$ having all internal vertices in V^{new}.

Thus we consider vertices in $X = \{u \in V^{old} : \exists (u, v) \in A^{new}\}$. We initialize $Q_i \leftarrow \{x \in X : d_i(x) \neq \infty\}$ for $i = 1, ..., |V|$, from which we proceed with the label update described above. The algorithm is shown in Algorithm 17.

Improved Results

To measure the effects of the algorithmic enhancements, we compare four possible configurations of the discretizationless approach, along with the construction phase of MCPH. We compare these configurations on both the industry and synthetic instances described in Section 3.5.1.

The first configuration, denoted "Rigid Only" has none of the enhancements. The "Efficient Rigid Only" implementation has the improved Bellman label updates from Algorithm 17. The "Flex with Rigid" implementation defaults to using flexible windows, but if necessary uses regu-

Algorithm 17 Window Labeling

```
1: function UpdateLabels($G = (V^{old} \cup V^{new}, A^{old} \cup A^{new})$, $\{d_i(u) | u \in V^{old}, i = 0, 1, ..., |V^{old}|\}$)
2:     $d_0(u) \leftarrow \begin{cases} 0 & \text{if } u \in E \\ \infty & \text{otherwise} \end{cases} \forall\, u \in V^{new}$
3:     $Q_0 \leftarrow V^{new} \cap E$
4:     $X = \{u \in V^{old} : \exists (u, v) \in A^{new}\}$
5:     $Q_i \leftarrow \{x \in X : d_i(x) \neq \infty\}$ for $i = 1, ..., |V^{old}|$
6:     $Q_i \leftarrow \emptyset$ for $i = |V^{old}| + 1, ..., |V^{old}| + |V^{new}|$
7:     for $i = 0, ..., |V|$ do
8:         for $u \in Q_i$ do
9:             for $v \in \delta^{out}(u)$ do
10:                if $d_i(u) + C_{uv} < d_{i+1}(v)$ then
11:                    $d_{i+1}(v) \leftarrow d_i(u) + C_{uv}$
12:                    $Q_{i+1} \leftarrow Q_{i+1} \cup \{v\}$
13:    $Opt \leftarrow \min_{v \in L, k \in \{2,...,n\}} \{\frac{d_k(v)}{k}\}$
14:    return $\{d_i(v)\}_{v \in V, i=1,...,|V|}$
```

lar rigid window updates as described in Algorithm 16. Finally, the "Flex with Efficient Rigid" configuration has both enhancements.

Table 3.9 shows a comparison of the implementations on the synthetic instance, and Table 3.10 shows the results for the industry instance. We observe that both enhancements provide significant improvements in solution time. Furthermore, defaulting to flexible windows has a noticeable beneficial impact on the plan cost. We proceed in the experiments using the "Flex with Efficient Rigid" implementation of the discretizationless approach.

Table 3.9: Synthetic instance results

Method	Solution Time (s)	Plan Cost
Discretized - 15 mins	7,271.31	47,681,261.84
Rigid Only	72,487.46	46,164,844.28
Efficient Rigid Only	15,773.47	46,231,170.34
Flex With Rigid	647.92	45,676,587.93
Flex With Efficient Rigid	518.69	45,624,297.36

Table 3.10: Industry instance results

Method	Solution Time (s)	Plan Cost
Discretized - 15 mins	3,198.98	4,583,283.36
Rigid Only	258,309.00	4,301,214.81
Efficient Rigid Only	13,432.40	4,294,044.50
Flex with Rigid	307.06	4,149,184.26
Flex with Efficient Rigid	249.29	4,140,846.12

3.5.4 Enhanced Algorithm Results

In this section, we provide an extensive computational study to benchmark the discretizationless approach using all enhancements described earlier. We compare the approach to four other solution approaches. The first, denoted MCPH, is a marginal cost pathing approach on a fixed discretization containing both a construction phase and improvement phases. The second approach, denoted LPRound, is to solve the linear programming relaxation of the problem on a fully time expanded network and round up to the cheapest integral capacities that accommodate this flow. The third alternative approach, denoted SlpSc, is to use a slope scaling heuristic. Note that the first iteration of SlpSc is equivalent to LPRound. The final alternative approach we consider is to solve the problem exactly using a mixed integer programming formulation on a fully time expanded network. This approach is denoted FullMIP.

Since practically sized instances are intractable for FullMIP, we divide the set of test instances into small (400 or fewer commodities) and large (10,000-100,000 commodities) categories. We give each approach a time limit of one hour to begin their final improvement iteration on any instance. We provide FullMIP results only for small instances. We investigate the solution time, plan cost, and planned number of dispatches for each approach. Furthermore, we provide metrics on the utilization and path length distributions of the plans produced by each method.

Results on Small Instances

We provide results for all five solution approaches, including the exact approach FullMIP, for instances having 400 or fewer commodities. FullMIP was only able to produce and certify optimality for the instances with 100 commodities. For other instances, we use the best solution found by the solver during one hour. Table 3.11 shows the solution time in seconds, as well as the cost and number of dispatches of the plans produced by each method. We can see that MCPH, LPRound, and Discless are dominant in terms of solution time. The plans produced by LPRound and SlpSc are consistently worse than MCPH and Discless.

Table 3.11: Cost, solution time, and number of dispatches for small instances

Instance					Method	Cost	Time	Number of
Type	n_R	$\lvert L\rvert$	n_H	$\lvert K\rvert$				Dispatches
Synthetic	3	25	3	100	FullMIP	373,220.00	252.03	128
Synthetic	3	25	3	100	LPRound	581,590.00	0.84	161
Synthetic	3	25	3	100	Discless	396,300.00	0.01	121
Synthetic	3	25	3	100	MCPH	425,580.00	0.09	126
Synthetic	3	25	3	100	SlpSc	494,910.00	33.94	149
Synthetic	5	50	3	100	FullMIP	455,570.00	520.59	182
Synthetic	5	50	3	100	LPRound	662,280.00	2.37	221
Synthetic	5	50	3	100	Discless	475,970.00	0.01	178
Synthetic	5	50	3	100	MCPH	499,760.00	0.17	184
Synthetic	5	50	3	100	SlpSc	548,940.00	123.88	199
Synthetic	3	25	3	200	FullMIP	573,840.00	1433.52	224
Synthetic	3	25	3	200	LPRound	1,200,810.00	0.86	376
Synthetic	3	25	3	200	Discless	618,340.00	0.02	224
Synthetic	3	25	3	200	MCPH	654,010.00	0.54	230
Synthetic	3	25	3	200	SlpSc	816,940.00	43.70	287
Synthetic	5	50	3	200	FullMIP	698,310.00	3615.30	334
Synthetic	5	50	3	200	LPRound	1,238,310.00	2.19	420
Synthetic	5	50	3	200	Discless	777,890.00	0.03	317
Synthetic	5	50	3	200	MCPH	786,260.00	0.75	315
Synthetic	5	50	3	200	SlpSc	971,520.00	128.18	373
Synthetic	3	25	3	300	FullMIP	814,540.00	3609.98	319
Synthetic	3	25	3	300	LPRound	1,954,810.00	1.16	548
Synthetic	3	25	3	300	Discless	927,509.39	0.03	323
Synthetic	3	25	3	300	MCPH	903,800.00	1.96	309
Synthetic	3	25	3	300	SlpSc	1,236,616.19	116.17	398
Synthetic	5	50	3	300	FullMIP	895,440.00	3625.23	501
Synthetic	5	50	3	300	LPRound	1,639,160.00	3.16	609
Synthetic	5	50	3	300	Discless	937,440.00	0.05	441
Synthetic	5	50	3	300	MCPH	918,209.18	3.94	430
Synthetic	5	50	3	300	SlpSc	1,103,453.08	438.09	517
Synthetic	3	25	3	400	FullMIP	1,068,858.49	3610.20	436
Synthetic	3	25	3	400	LPRound	2,497,570.00	1.02	753
Synthetic	3	25	3	400	Discless	1,075,903.55	0.04	393
Synthetic	3	25	3	400	MCPH	1,072,573.93	3.53	382
Synthetic	3	25	3	400	SlpSc	1,408,734.74	83.42	504
Synthetic	5	50	3	400	FullMIP	1,256,124.03	3633.34	645
Synthetic	5	50	3	400	LPRound	2,431,420.00	2.98	852
Synthetic	5	50	3	400	Discless	1,273,300.00	0.07	566
Synthetic	5	50	3	400	MCPH	1,293,690.00	4.03	563
Synthetic	5	50	3	400	SlpSc	1,631,618.16	759.15	679

Table 3.12 shows the linear programming lower bound, the bounds produced by FullMIP, and the plan costs for MCPH and Discless for each instance. One observes that the LP bound is

consistently weak.

Table 3.12: MCPH vs Discless comparison with bounds

Type	n_R	$\|L\|$	n_H	$\|K\|$	LP Bound	MIP LB	MIP UB	MCPH	Discless
Synthetic	3	25	3	100	5,860.90	373,220.00	373,220.00	425,580.00	396,300.00
Synthetic	5	50	3	100	6,622.80	455,540.00	455,570.00	499,760.00	475,970.00
Synthetic	3	25	3	200	12,085.20	573,785.00	573,840.00	654,010.00	618,340.00
Synthetic	5	50	3	200	12,383.10	695,060.00	698,310.00	786,260.00	777,890.00
Synthetic	3	25	3	300	19,704.60	712,963.00	814,540.00	903,800.00	927,509.39
Synthetic	5	50	3	300	16,391.60	650,470.00	895,440.00	918,209.18	937,440.00
Synthetic	3	25	3	400	25,057.10	699,275.00	1,068,860.00	1,072,573.93	1,075,903.55
Synthetic	5	50	3	400	24,352.90	837,096.00	1,256,120.00	1,293,690.00	1,273,300.00

Table 3.13 outlines the gap of the MCPH and Discless approaches. Because we could not produce an optimal solution for every instance, we provide gap calculated with the FullMIP lower bound, as well as gap calculated with the average of the two FullMIP bounds.

Table 3.13: MCPH vs Discless gap

					To MIP LB		To Avg MIP Bound	
Type	n_R	$\|L\|$	n_H	$\|K\|$	MCPH Gap	Discless Gap	MCPH Gap	Discless Gap
Synthetic	3	25	3	100	12.30	5.82	12.30	5.82
Synthetic	5	50	3	100	8.85	4.29	8.85	4.29
Synthetic	3	25	3	200	12.27	7.21	12.26	7.20
Synthetic	5	50	3	200	11.60	10.65	11.39	10.44
Synthetic	3	25	3	300	21.11	23.13	15.50	17.66
Synthetic	5	50	3	300	29.16	30.61	15.82	17.55
Synthetic	3	25	3	400	34.80	35.01	17.58	17.83
Synthetic	5	50	3	400	35.29	34.26	19.10	17.80

We note that Discless produces plans between 4-18% worse than the average bound, and MCPH produces plans between 8-19% worse than the average bound. Discless produces the best plan of the non-exact methods on 5/8 instances, while MCPH produces the best plan on the remaining 3/8 instances. We attribute this to the fact that MCPH has an improvement phase, while Discless is a pure construction heuristic. Furthermore, Discless produces a plan faster than MCPH on every instance.

Table 3.14 shows the minimum, average, and maximum trailer utilization of the plans produced by each method. The average plan utilization roughly tracks the plan cost as expected.

Table 3.15 shows the minimum, average, and maximum length of a commodity's path measured in number of dispatches. Longer average path lengths can result in higher consolidation, but plans containing longer paths may be more difficult to operate in practice and may be more susceptible to later deliveries from random delays or weather shutdowns. As expected, LPRound produces plans with the lowest average path length. This is because the LP relaxation can be solved by putting commodities independently on their minimum cost paths where the arc-costs are the cost per unit capacity of vehicles. The LP relaxation decomposes by commodity, and there is no incentive for consolidation. We also note that FullMIP produced the best solution on every instance and also has the largest average path length on every instance.

Table 3.14: Plan utilization for small instances

Instance								
Type	n_R	$\|L\|$	n_H	$\|K\|$	Method	MinUtil	AvgUtil	MaxUtil
Synthetic	3	25	3	100	FullMIP	0.010	0.163	0.580
Synthetic	3	25	3	100	LPRound	0.010	0.103	0.310
Synthetic	3	25	3	100	Discless	0.010	0.155	0.580
Synthetic	3	25	3	100	MCPH	0.010	0.147	0.580
Synthetic	3	25	3	100	SlpSc	0.010	0.126	0.430
Synthetic	5	50	3	100	FullMIP	0.010	0.153	0.530
Synthetic	5	50	3	100	LPRound	0.010	0.105	0.200
Synthetic	5	50	3	100	Discless	0.010	0.150	0.530
Synthetic	5	50	3	100	MCPH	0.010	0.138	0.430
Synthetic	5	50	3	100	SlpSc	0.010	0.132	0.710
Synthetic	3	25	3	200	FullMIP	0.010	0.217	0.980
Synthetic	3	25	3	200	LPRound	0.010	0.105	0.390
Synthetic	3	25	3	200	Discless	0.010	0.192	0.890
Synthetic	3	25	3	200	MCPH	0.010	0.193	0.900
Synthetic	3	25	3	200	SlpSc	0.010	0.157	0.890
Synthetic	5	50	3	200	FullMIP	0.010	0.192	0.900
Synthetic	5	50	3	200	LPRound	0.010	0.110	0.200
Synthetic	5	50	3	200	Discless	0.010	0.173	0.900
Synthetic	5	50	3	200	MCPH	0.010	0.171	0.690
Synthetic	5	50	3	200	SlpSc	0.010	0.141	0.980
Synthetic	3	25	3	300	FullMIP	0.010	0.240	1.000
Synthetic	3	25	3	300	LPRound	0.010	0.103	0.250
Synthetic	3	25	3	300	Discless	0.010	0.215	1.000
Synthetic	3	25	3	300	MCPH	0.010	0.221	0.930
Synthetic	3	25	3	300	SlpSc	0.010	0.170	0.887
Synthetic	5	50	3	300	FullMIP	0.010	0.194	0.900
Synthetic	5	50	3	300	LPRound	0.010	0.104	0.200
Synthetic	5	50	3	300	Discless	0.010	0.182	0.810
Synthetic	5	50	3	300	MCPH	0.010	0.196	0.950
Synthetic	5	50	3	300	SlpSc	0.010	0.166	0.980
Synthetic	3	25	3	400	FullMIP	0.010	0.222	1.000
Synthetic	3	25	3	400	LPRound	0.010	0.099	0.200
Synthetic	3	25	3	400	Discless	0.010	0.222	1.000
Synthetic	3	25	3	400	MCPH	0.010	0.227	0.980
Synthetic	3	25	3	400	SlpSc	0.010	0.171	0.940
Synthetic	5	50	3	400	FullMIP	0.010	0.205	0.910
Synthetic	5	50	3	400	LPRound	0.010	0.105	0.200
Synthetic	5	50	3	400	Discless	0.010	0.201	0.880
Synthetic	5	50	3	400	MCPH	0.010	0.201	0.990
Synthetic	5	50	3	400	SlpSc	0.010	0.164	0.920

Table 3.15: Plan path lengths for small instances

Instance								
Type	n_R	$\|L\|$	n_H	$\|K\|$	Method	MinPathLength	AvgPathLength	MaxPathLength
Synthetic	3	25	3	100	FullMIP	1	2.02	5
Synthetic	3	25	3	100	LPRound	1	1.63	3
Synthetic	3	25	3	100	Discless	1	1.83	5
Synthetic	3	25	3	100	MCPH	1	1.82	4
Synthetic	3	25	3	100	SlpSc	1	1.86	5
Synthetic	5	50	3	100	FullMIP	1	2.62	6
Synthetic	5	50	3	100	LPRound	1	2.21	3
Synthetic	5	50	3	100	Discless	1	2.49	5
Synthetic	5	50	3	100	MCPH	1	2.41	5
Synthetic	5	50	3	100	SlpSc	1	2.45	5
Synthetic	3	25	3	200	FullMIP	1	2.32	6
Synthetic	3	25	3	200	LPRound	1	1.89	3
Synthetic	3	25	3	200	Discless	1	2.10	6
Synthetic	3	25	3	200	MCPH	1	2.12	5
Synthetic	3	25	3	200	SlpSc	1	2.17	6
Synthetic	5	50	3	200	FullMIP	1	2.90	6
Synthetic	5	50	3	200	LPRound	1	2.10	3
Synthetic	5	50	3	200	Discless	1	2.52	5
Synthetic	5	50	3	200	MCPH	1	2.45	5
Synthetic	5	50	3	200	SlpSc	1	2.41	6
Synthetic	3	25	3	300	FullMIP	1	2.46	7
Synthetic	3	25	3	300	LPRound	1	1.84	4
Synthetic	3	25	3	300	Discless	1	2.25	6
Synthetic	3	25	3	300	MCPH	1	2.19	6
Synthetic	3	25	3	300	SlpSc	1	2.22	6
Synthetic	5	50	3	300	FullMIP	1	3.08	8
Synthetic	5	50	3	300	LPRound	1	2.03	3
Synthetic	5	50	3	300	Discless	1	2.56	6
Synthetic	5	50	3	300	MCPH	1	2.69	8
Synthetic	5	50	3	300	SlpSc	1	2.73	7
Synthetic	3	25	3	400	FullMIP	1	2.51	6
Synthetic	3	25	3	400	LPRound	1	1.89	3
Synthetic	3	25	3	400	Discless	1	2.23	5
Synthetic	3	25	3	400	MCPH	1	2.20	5
Synthetic	3	25	3	400	SlpSc	1	2.26	5
Synthetic	5	50	3	400	FullMIP	1	3.14	8
Synthetic	5	50	3	400	LPRound	1	2.14	4
Synthetic	5	50	3	400	Discless	1	2.68	7
Synthetic	5	50	3	400	MCPH	1	2.64	6
Synthetic	5	50	3	400	SlpSc	1	2.64	9

Results on Large Instances

In this section, we test the approaches on instances having 10,000 - 100,000 commodities. For these instances, FullMIP can not produce helpful bounds in under an hour, so we exclude it. We give each method a time limit of one hour to begin its final improvement iteration. Some methods took much longer than one hour to produce any solution, but we still list these results.

Table 3.16 shows the solution time in seconds and the cost and number of dispatches of the plan produced by each method. Similar to the small instances, MCPH and Discless are producing the best plans among the non-exact methods. However, for these large instances, MCPH is using the entire hour without finding a local optimum.

Table 3.16: Cost, solution time, and number of dispatches for large instances

Instance					Method	Cost	Time	Number of
Type	n_R	$\lvert L\rvert$	n_H	$\lvert K\rvert$				Dispatches
Synthetic	3	25	3	10,000	LPRound	54,333,270.00	172.55	13,917
Synthetic	3	25	3	10,000	Discless	5,771,684.34	5.35	1,889
Synthetic	3	25	3	10,000	MCPH	6,247,893.28	3,611.17	2,098
Synthetic	3	25	3	10,000	SlpSc	13,782,760.44	3,626.50	3,770
Synthetic	5	50	3	10,000	LPRound	57,201,570.00	524.83	15,950
Synthetic	5	50	3	10,000	Discless	7,172,489.00	9.80	3,116
Synthetic	5	50	3	10,000	MCPH	8,012,073.80	3,601.40	3,461
Synthetic	5	50	3	10,000	SlpSc	21,311,544.19	3,706.10	6,181
Synthetic	3	25	3	25,000	LPRound	127,436,850.00	473.61	31,990
Synthetic	3	25	3	25,000	Discless	11,772,327.72	21.50	3,064
Synthetic	3	25	3	25,000	MCPH	12,415,424.16	3,746.13	3,501
Synthetic	3	25	3	25,000	SlpSc	24,850,067.27	3,662.22	5,826
Synthetic	5	50	3	25,000	LPRound	134,086,370.00	1,296.06	38,726
Synthetic	5	50	3	25,000	Discless	13,309,596.68	41.69	5,235
Synthetic	5	50	3	25,000	MCPH	14,622,307.66	3,657.94	5,936
Synthetic	5	50	3	25,000	SlpSc	36,401,429.04	3,642.84	10,441
Synthetic	3	25	3	50,000	LPRound	240,469,557.64	944.25	55,656
Synthetic	3	25	3	50,000	Discless	22,223,570.96	58.76	4,088
Synthetic	3	25	3	50,000	MCPH	22,910,389.80	3938.46	4,835
Synthetic	3	25	3	50,000	SlpSc	38,784,586.82	3676.10	7,090
Synthetic	5	50	3	50,000	LPRound	257,378,460.00	2753.55	73,127
Synthetic	5	50	3	50,000	Discless	23,307,341.66	121.87	7,747
Synthetic	5	50	3	50,000	MCPH	24,805,878.60	3752.34	8,959
Synthetic	5	50	3	50,000	SlpSc	59,833,330.36	4081.40	16,631
Synthetic	3	25	3	100,000	LPRound	386,973,917.94	1750.89	92,702
Synthetic	3	25	3	100,000	Discless	40,802,297.05	168.70	5,803
Synthetic	3	25	3	100,000	MCPH	40,637,138.35	3627.67	6,842
Synthetic	3	25	3	100,000	SlpSc	62,330,476.18	3623.98	9,914
Synthetic	5	50	3	100,000	LPRound	411,962,356.66	4861.10	127,450
Synthetic	5	50	3	100,000	Discless	39,746,801.24	358.24	10,988
Synthetic	5	50	3	100,000	MCPH	41,320,338.04	3642.30	12,840
Synthetic	5	50	3	100,000	SlpSc	283,162,039.85	3751.25	81,285
South China	N/A	69	N/A	100,281	LPRound	71,802,412.77	8,166.92	64,479
South China	N/A	69	N/A	100,281	Discless	4,155,073.91	246.15	6,841
South China	N/A	69	N/A	100,281	MCPH	4,632,395.45	4,016.47	8,286
South China	N/A	69	N/A	100,281	SlpSc	71,802,412.77	4,426.55	64,479

Table 3.17 shows a comparison of MCPH and Discless. Despite having no improvement phase, Discless produces the best solution on all but one instance. The improvement over MCPH rangers

from -0.41% to 10.48%. Furthermore, its solution time is orders of magnitude smaller than that of MCPH.

Table 3.17: MCPH vs Discless on large instances

					MCPH		Discless		% Better	
Type	n_R	$\|L\|$	n_H	$\|K\|$	Cost	Time	Cost	Time	Cost	Time
Synthetic	3	25	3	10000	6,247,893.28	3611.17	5,771,684.34	5.35	7.62	99.85
Synthetic	5	50	3	10000	8,012,073.80	3601.40	7,172,489.00	9.80	10.48	99.73
Synthetic	3	25	3	25000	12,415,424.16	3746.13	11,772,327.72	21.50	5.18	99.43
Synthetic	5	50	3	25000	14,622,307.66	3657.94	13,309,596.68	41.69	8.98	98.86
Synthetic	3	25	3	50000	22,910,389.80	3938.46	22,223,570.96	58.76	3.00	98.51
Synthetic	5	50	3	50000	24,805,878.60	3752.34	23,307,341.66	121.87	6.04	96.75
Synthetic	3	25	3	100000	40,637,138.35	3627.67	40,802,297.05	168.70	-0.41	95.35
Synthetic	5	50	3	100000	41,320,338.04	3642.30	39,746,801.24	358.24	3.81	90.16
South China	N/A	69	N/A	100,281	4,632,395.45	4016.47	4,155,073.91	246.15	10.30	93.87

Table 3.18 shows the LP bound and the best solution found for each instance. Similar to the small instances, the LP bound is weak.

Table 3.18: LP bound on large instances

Instance						
Type	n_R	$\lvert L \rvert$	n_H	$\lvert K \rvert$	LP Bound	Best Solution
Synthetic	3	25	3	10,000	571,154.00	5,771,684.34
Synthetic	5	50	3	10,000	585,525.00	7,172,489.00
Synthetic	3	25	3	25,000	1,454,760.00	11,772,327.72
Synthetic	5	50	3	25,000	1,417,230.00	13,309,596.68
Synthetic	3	25	3	50,000	3,094,360.00	22,223,570.96
Synthetic	5	50	3	50,000	2,861,080.00	23,307,341.66
Synthetic	3	25	3	100,000	5,816,920.00	40,637,138.35
Synthetic	5	50	3	100,000	5,275,080.00	39,746,801.24
South China	N/A	69	N/A	100,281	26,546.80	4,155,073.91

Table 3.19 shows the minimum, average, and maximum trailer utilization of the plans produced by each method. The average utilization roughly tracks the efficiency of the plan. For the South China instance, SlpSc was only able to perform one iteration, resulting in it producing a plan equivalent to LPRound.

Table 3.20 shows the minimum, average, and maximum length of each commodity's path measured in number of dispatches. Again, the average path length is a proxy for the level of consolidation. We see the maximum path length achieving numbers unlikely to be useful in any practical plan, suggesting that in practice companies should add penalties or constrain such solutions from being produced.

Table 3.19: Plan utilization for large instances

Instance								
Type	n_R	$\|L\|$	n_H	$\|K\|$	Method	MinUtil	AvgUtil	MaxUtil
Synthetic	3	25	3	10,000	LPRound	0.010	0.112	0.500
Synthetic	3	25	3	10,000	Discless	0.010	0.848	1.000
Synthetic	3	25	3	10,000	MCPH	0.010	0.801	1.000
Synthetic	3	25	3	10,000	SlpSc	0.010	0.243	1.000
Synthetic	5	50	3	10,000	LPRound	0.010	0.108	0.570
Synthetic	5	50	3	10,000	Discless	0.010	0.735	1.000
Synthetic	5	50	3	10,000	MCPH	0.010	0.709	1.000
Synthetic	5	50	3	10,000	SlpSc	0.010	0.194	1.000
Synthetic	3	25	3	25,000	LPRound	0.010	0.122	0.790
Synthetic	3	25	3	25,000	Discless	0.010	0.934	1.000
Synthetic	3	25	3	25,000	MCPH	0.010	0.911	1.000
Synthetic	3	25	3	25,000	SlpSc	0.010	0.257	1.000
Synthetic	5	50	3	25,000	LPRound	0.010	0.114	0.730
Synthetic	5	50	3	25,000	Discless	0.010	0.870	1.000
Synthetic	5	50	3	25,000	MCPH	0.010	0.849	1.000
Synthetic	5	50	3	25,000	SlpSc	0.010	0.213	1.000
Synthetic	3	25	3	50,000	LPRound	0.010	0.142	1.000
Synthetic	3	25	3	50,000	Discless	0.060	0.983	1.000
Synthetic	3	25	3	50,000	MCPH	0.100	0.967	1.000
Synthetic	3	25	3	50,000	SlpSc	0.010	0.296	1.000
Synthetic	5	50	3	50,000	LPRound	0.010	0.122	0.800
Synthetic	5	50	3	50,000	Discless	0.010	0.926	1.000
Synthetic	5	50	3	50,000	MCPH	0.010	0.908	1.000
Synthetic	5	50	3	50,000	SlpSc	0.010	0.223	1.000
Synthetic	3	25	3	100,000	LPRound	0.010	0.169	1.000
Synthetic	3	25	3	100,000	Discless	0.200	0.993	1.000
Synthetic	3	25	3	100,000	MCPH	0.080	0.980	1.000
Synthetic	3	25	3	100,000	SlpSc	0.010	0.308	1.000
Synthetic	5	50	3	100,000	LPRound	0.010	0.142	1.000
Synthetic	5	50	3	100,000	Discless	0.040	0.972	1.000
Synthetic	5	50	3	100,000	MCPH	0.010	0.946	1.000
Synthetic	5	50	3	100,000	SlpSc	0.010	0.191	1.000
South China	N/A	69	N/A	100,281	LPRound	1.4E-05	0.074	1.000
South China	N/A	69	N/A	100,281	Discless	1.4E-04	0.776	1.000
South China	N/A	69	N/A	100,281	MCPH	1.4E-04	0.751	1.000
South China	N/A	69	N/A	100,281	SlpSc	1.4E-05	0.074	1.000

Table 3.20: Plan path lengths for large instances

Instance								
Type	n_R	$\|L\|$	n_H	$\|K\|$	Method	MinPathLength	AvgPathLength	MaxPathLength
Synthetic	3	25	3	10,000	LPRound	1	1.48	3
Synthetic	3	25	3	10,000	Discless	1	2.43	11
Synthetic	3	25	3	10,000	MCPH	1	2.37	10
Synthetic	3	25	3	10,000	SlpSc	1	2.28	7
Synthetic	5	50	3	10,000	LPRound	1	1.65	3
Synthetic	5	50	3	10,000	Discless	1	2.89	13
Synthetic	5	50	3	10,000	MCPH	1	2.91	11
Synthetic	5	50	3	10,000	SlpSc	1	2.62	8
Synthetic	3	25	3	25,000	LPRound	1	1.49	3
Synthetic	3	25	3	25,000	Discless	1	2.59	11
Synthetic	3	25	3	25,000	MCPH	1	2.34	11
Synthetic	3	25	3	25,000	SlpSc	1	2.14	7
Synthetic	5	50	3	25,000	LPRound	1	1.68	3
Synthetic	5	50	3	25,000	Discless	1	3.16	15
Synthetic	5	50	3	25,000	MCPH	1	3.04	15
Synthetic	5	50	3	25,000	SlpSc	1	2.62	8
Synthetic	3	25	3	50,000	LPRound	1	1.50	3
Synthetic	3	25	3	50,000	Discless	1	3.02	14
Synthetic	3	25	3	50,000	MCPH	1	2.63	22
Synthetic	3	25	3	50,000	SlpSc	1	2.15	6
Synthetic	5	50	3	50,000	LPRound	1	1.70	5
Synthetic	5	50	3	50,000	Discless	1	3.46	15
Synthetic	5	50	3	50,000	MCPH	1	3.20	18
Synthetic	5	50	3	50,000	SlpSc	1	2.39	7
Synthetic	3	25	3	100,000	LPRound	1	1.49	3
Synthetic	3	25	3	100,000	Discless	1	3.01	12
Synthetic	3	25	3	100,000	MCPH	1	2.59	26
Synthetic	3	25	3	100,000	SlpSc	1	1.99	7
Synthetic	5	50	3	100,000	LPRound	1	1.72	4
Synthetic	5	50	3	100,000	Discless	1	3.60	20
Synthetic	5	50	3	100,000	MCPH	1	3.25	21
Synthetic	5	50	3	100,000	SlpSc	1	1.74	5
South China	N/A	69	N/A	100,281	LPRound	1	1.93	6
South China	N/A	69	N/A	100,281	Discless	1	2.79	14
South China	N/A	69	N/A	100,281	MCPH	1	3.64	21
South China	N/A	69	N/A	100,281	SlpSc	1	1.93	6

To investigate further into the types of plans produced by each method, we plot the utilization distributions in Figure 3.16 and the path length distributions in Figure 3.17 for the plans produced for the South China instance. Table 3.21 gives the path length distributions for the plans produced by each method on the South China instance. Despite being a better plan, the path length distribution produced by Discless is more practical than that of MCPH. We note that the SlpSc plan, equivalent to the LPRound plan since it was only able to perform one iteration, has a terrible utilization distribution, but the most realistic path length distribution.

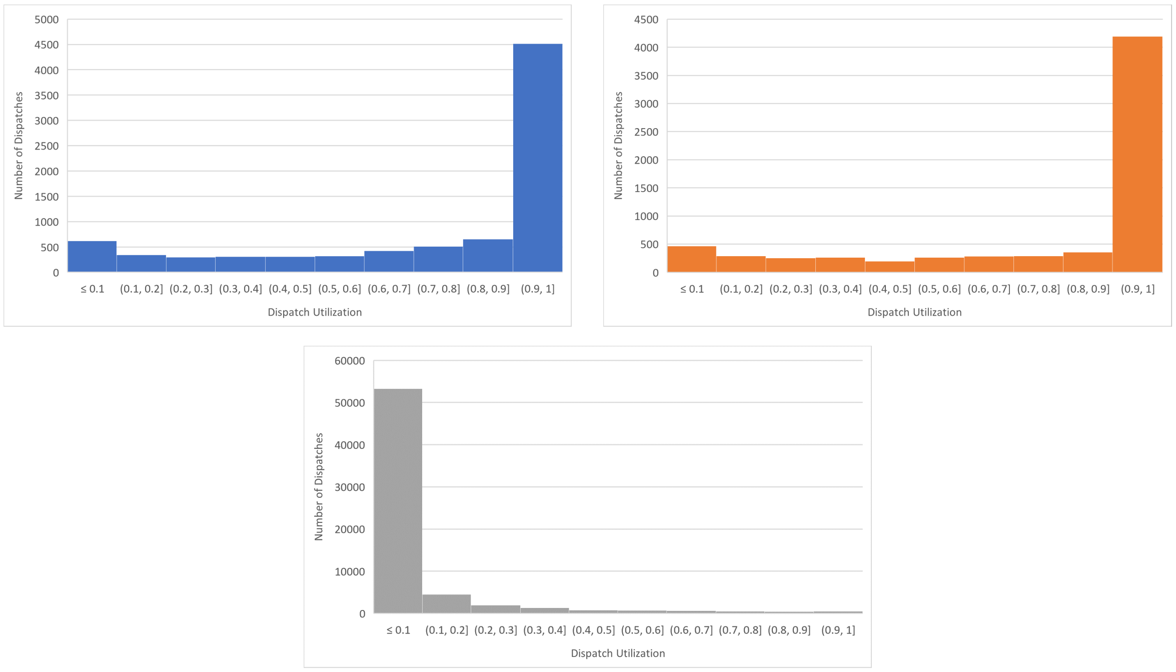


Figure 3.16: Utilization distributions for South China instance: MCPH(blue), Discless(orange), SlpSc(grey)

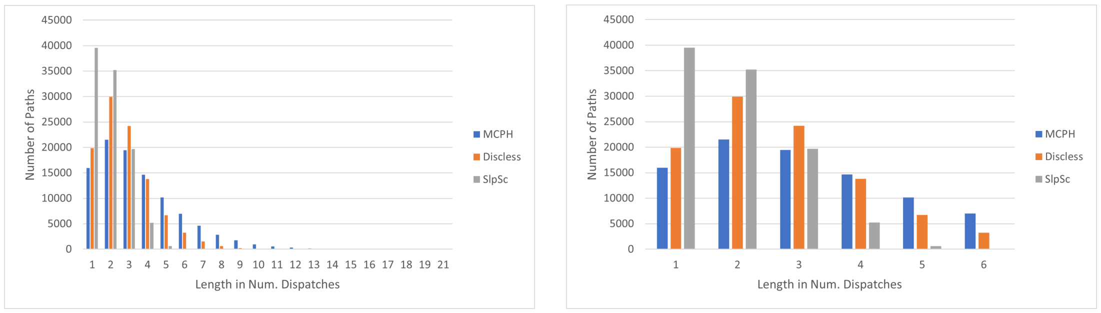


Figure 3.17: Path length distributions: South China instance

Table 3.21: Path lengths: South China instance

	Number of Paths		
Path Length	MCPH	Discless	SlpSc
1	15,969	19,853	39,524
2	21,524	29,925	35,205
3	19,462	24,226	19,674
4	14,648	13,800	5,217
5	10,164	6,708	597
6	6,988	3,229	64
7	4,623	1,535	0
8	2,876	643	0
9	1,787	229	0
10	984	102	0
11	558	22	0
12	330	7	0
13	169	1	0
14	85	1	0
15	49	0	0
16	33	0	0
17	16	0	0
18	9	0	0
19	5	0	0
21	2	0	0

In Figure 3.18, we show the distribution of the flexible window widths for the Discless solution to the South China instance. We observe that the plan is incredibly constrained with the most frequent window width (comprising over 2000 dispatches) being 0 minutes. Fewer than 250 dispatches have a departure interval longer than 30 minutes. This demonstrates both the time sensitivity of the problem, but also the potential fragility of the solution. In practice, it may be wise to enforce a minimum dispatch interval width to generate more practical plans.

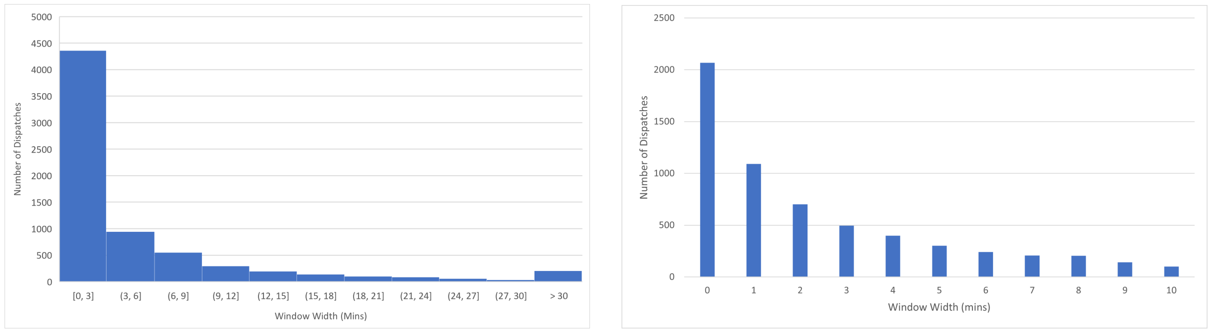


Figure 3.18: Dispatch window widths: South China instance

3.6 Conclusion

We provide a sequential greedy approach that does not rely on a fixed discretization. In such an approach, one does not need to store the entire time expanded network; only the dispatches used by some commodity are stored. Furthermore, the approach allows dispatch departure times to shift or move within a known time interval allowing consolidations that are not possible in sequential pathing on a fixed discretization.

We provide an in-depth analysis on the properties and methods of updating two types of departure intervals, each having its own benefits and trade offs. This type of interval modelling and updating can be applied to other domains where intervals are fixed during and updated after some planning iteration. For example, this work could be applied to some kind of sequential time-based route cover problem.

We demonstrate that the modelling and algorithms are capable of outperforming the best known heuristics for large scale flow and load planning. The approach outperforms a marginal cost path heuristic on industry scale instances in terms of cost and solution time.

On small instances, the discretizationless construction heuristic was able to produce solutions 4-35% worse than the best lower bound produced, and 4-18% worse than the average of the bounds produced by an exact approach. On the instances solved to near optimality, the gap was only 4-11%. The discretizationless approach was able to produce these solutions in less than one second.

On practically sized instances, the discretizationless approach produced the best solution on 8/9 instances with -0.41 to 10.48% improvement in solution cost over a marginal cost path heuristic despite not having an improvement phase. The discretizationless approach produced these solutions in under 360 seconds while MCPH used an hour to construct and improve each solution.

Having dispatch windows of positive width in sequential pathing will dominate a fixed discretization approach, because once a commodity is assigned to an arc, the departure time of that arc can not change. Therefore, another commodity originating one minute later will not be able to consider consolidating on this arc, because the departure time of the earlier commodity cannot shift later. However, departure intervals allow time shifts as described to be considered.

One can view the use of rigid departure intervals as having a similar effect to pessimistic mapping error in fixed discretization approaches. The dispatch time being constrained more than necessary is similar in effect to the assumption that the travel time is longer than it truly is.

One way to eliminate all interval/mapping error entirely would be to dynamically update the flexible windows after each reach in the path finding algorithm. This would come at the cost of an exponential increase in memory and solution time due to the copying required at each branch of the solution tree.

Future work on this topic should include the development of a local search improvement scheme. We have also motivated a method of constraining the maximum path length and a minimum dispatch interval width in order to enhance the practicality of the solutions to this problem.

CHAPTER 4
THE FLOW RULE PROBLEM

4.1 Introduction

This work focuses on modelling and solution approaches for joint flow and load planning at a major parcel express company. Consider the operations of a parcel delivery company. There exists a set of packages which must be delivered within their individual time constraints. They can be transported to their destinations sharing common vehicles. They can stop at intermediate locations at which they accrue a sort cost and occupy some sort capacity.

We concern ourselves in this work with a particular method of sort operations. Sort operations occur in shifts throughout the day. We refer to any particular shift as a sort. When a trailer arrives to a building during a sort, it is unloaded, and the contents of the trailer are inducted into the sort process. The sort process directs each package from the unloaded trailer to some outbound door. In the actual operations of many parcel express companies, packages are directed to an outbound door based on their destination terminal and service class. A *flow rule* specifies that flow currently at building b_1 at sort s_1 destined for building b_{dest} of class c may be sent next to building b_2 at sort s_2. Historically, the determination of all flow rules was called *flow planning* and this process was performed before *load planning* which referred to determining the departure times and contents of trailers following the flow plan. The flow plan specifies sequences of building sort pairs, which we call *sort paths*, that can carry packages to their destination, while the load plan specifies a specific timed-path that adheres to the flow plan that carries packages to their destinations.

Note that there may be multiple flow rules that apply to packages at building b_1 at sort s_1 destined for building b_{dest} of class c; however, there will always be at least one. At each building b_1 and sort s_1, one particular rule for each destination and service class is singled out as the default mode of operation and referred to as the *primary flow rule*. Each package has a unique sort path to its destination comprised only of primary flow rules, which we call a *primary sort path*.

Typically, at the flow and load planning stage, the granularity at which the operations are modelled can be coarse. Planning tools built on this modelling yield plans that are not implementable and need post-processing or manual adjustment. In particular, many models assume each package, or any group of packages (however defined), can be individually directed through the logistics network.

However, package express companies may have only a subset of their buildings sufficiently automated to accomplish this. Many buildings, particularly those with insufficient volume to justify full automation, operate via human beings who direct packages to an outbound loading door based on readable criteria on the package label: for instance the final destination and service class of the package. In a fully automated facility, each individual package could be assigned a specific door. However, in industry sort processes do not direct packages in this way. Such sort processes were designed for humans to be able to direct packages. Humans can not read a package ID of several digits and recognize which door it was assigned to. Instead, packages are typically directed to a door based on only their final destination and service class. On many parcel labels, the data in the largest font will be an indicator of the final destination and the service class, so that a human can quickly recognize this information and direct it to an outbound door.

Existing approaches in the literature have used in-tree constraints to model the case where each destination and service class have exactly one applicable flow rule at each sort. When generalizing to more than one rule, a particular challenge arises when enforcing that each package must be delivered before its due date; we refer to this as the time feasibility of the solution. For any particular destination and service class, there may exist several outbound doors available. However, for any particular package, one can not control which outbound door it is directed to. Even though two arriving packages have the same destination and service class, they may be due on different days. This leads to a particular phenomenon not addressed by existing models in the literature: all outbound door selections must be time feasible for any arriving package. Not only this, but at all subsequent stops, the available outbound doors must be time feasible for any package that *could have* arrived there.

In this operational context, models which can direct packages on other criteria, such as their due date, can make considerably cheaper non-implementable plans by ignoring such constraints. This work will model flow and loading planning where predefined flow classes (e.g. those sharing a common final destination and service class) must directed jointly at each sort in such a way that any particular package belonging to the class is guaranteed to be delivered on time.

Smooth operations during a sort are a highly desirable trait in a plan. Extending a sort shift can be expensive or impossible; however, this situation is faced when too much volume arrives late in the sort. Thus incorporating some aspect of time-based sort capacity is necessary to generate practically implementable plans. This work proposes a model where the sorts are discretized into time intervals, each containing a specified sort capacity.

The output of a flow and loading planning model is a set of timed loads and a list specifying how much of each flow class must be loaded into each load. In operations, volume is inducted into a sort when trailers are unloaded at unload doors. The sort then directs the volume to a loading door where it is immediately loaded into a trailer. In order to guarantee a plan is implementable, flow must depart every sort in the same order it was inducted.

In Section 4.3 we mathematically define the problem and model the practical constraints. In Section 4.4 we provide a general scheme for developing improvement heuristics that maintain time-feasibility, sort-feasibility, and FIFO-feasibility, and outline specific implementations. Section 4.5 contains computational experiments to gauge the performance of the improvement heuristics on large scale industry instances. Finally, Section 4.6 contains concluding remarks.

4.2 Literature Review

[5] provide a recent comprehensive overview of service network design and flow and load planning. [19] and [20] provide an overviews of service network design freight transportation modelling. [34] provides an early overview of various problems arising in network design. They detail various solution approaches including optimization-based approaches and heuristics.

The root of many solution approaches for flow and load planning or service network design lies in work done to solve multicommodity fixed charge flow. Approaches combining optimization and heuristics to solve this problem include [35] and [28].

[36] describe the postal and express shipment setting describing the interplay between automated and user planning for multi modal shipment planning.

[37] give a load planning formulation with practical constraints. They enforce that their solutions form in-trees; that is, each flow class can only have one rule to follow along each step of its path. [14] use a solution approach containing tree-based variables that can solve instances with hundreds of commodities. They pre-generate sets of geographic flow paths that are used to construct solutions. They make use of a slope scaling heuristic and column generation to introduce new variables.

The incorporation of practical constraints and downstream decisions has been observed in a variety of applications. [21] explain the hierarchy of decision making in air crew scheduling and also model the problem where the departure time of flights can vary within time windows.

[31] consider the problem of simultaneously performing load-matching and routing with equipment balancing. They define load-matching and routing as determining the timed-paths each load will take on the movement leg network and determining which pairs of trailers will be pulled by a common tractor. They decompose the problem into two sub-problems; the first determines which loads and empties are matched, and their physical routes. The second determines the scheduling of the loads. They define a cluster as a group of loads that interact by using common tractors and

trailers on their paths to their final destination. They present a set partitioning formulation with a binary variable for each cluster to determine whether or not it is included in the solution.

[38] integrate the timing of local pickup and delivery into a SND model which produces delivery routes. Customers are assigned a-priori to origin and destination terminals. Customer shipments are sent direct to their origin terminal, and vehicle routes are generated from their destination terminal. The approach incorporates the synchronization of the pickup, transsshipment, and delivery route timings into a single model. The output of their approach is a flow and load plan along with a set of delivery routes to deliver shipments from their destination terminal to the final customer location. They provided a route based model and an arc based model and use DDD to accommodate a fine time granulation. In the route based model, they enumerate all delivery routes beforehand and add a virtual node for each route to the time-space network. They then connect the route nodes to commodity destination nodes which they can cover.

[39] use time expanded network modelling to develop several heuristics for decision support tools to perform dynamic loadplanning.

A recent work exploring plans containing multiple alts, rather than in-tree plans, is given in [40]. They consider $\mathbf{p}$-alt plans, where $\mathbf{p}$ is a vector indexed by each (i, d) building-destination pair indicating the number of possible next stops for a commodity at building i destined for d. Typical load planning with in-tree constraints is the case of a $\mathbf{1}$-alt plan. They examine a two stage problem where in the first stage, the number of trailers moving between buildings is specified, and the specific p_{id} alternates for each building-destination pair (i, d) are chosen. In the second stage, random commodity sizes are realized, and commodities are routed through the network on paths adhering to the capacities set in the first stage and the $\mathbf{p}$-alt structure specified in the first stage. The authors show that significant cost savings can be achieved by allowing $\mathbf{2}$-alt operations. They also note that creating a $\mathbf{2}$-alt plan makes the solution less sensitive to demand uncertainty.

4.3 Problem Description

The logistics system we wish to plan for is built upon a geographic network $F = (B, A)$, sometimes referred to as the flat network, where every $b \in B$ is a building in the consolidation network and an arc $(u, v) \in A$ represents the ability to send vehicles and packages from u to v.

We want to determine the flows and loads on this network over a planning interval T. Buildings are not able to accept, sort, and dispatch volume during the entire planning interval. Instead, each building $b \in B$ operates its own set of sort shifts, which we denote S_b. Every sort $s \in S_b$ has a start time e_s and and end time l_s during which packages can originate in the building, packages can be transshipped through the building, and vehicles can arrive or depart from the building. We assume $[e_s, l_s]_{s \in S_b}$ are disjoint for every $b \in B$. Let $S = \cup_{b \in B} S_b$.

All volume that arrives to a sort shift must be sorted during that shift, but may be held in a trailer and dispatched at a later shift. Each sort shift $s \in S$ can process at most r_s units of volume per unit time.

As in the typical service network design setting, packages originate at a particular time, and must be delivered to some other building by a due date. We aggregate packages into sets which have common origin, destination, release (available) time, due time, and service class. For each commodity $k \in K$, we denote the origin $o_k \in B$, destination $d_k \in B$, release time $e_k \in T$, due time $l_k \in T$, service class λ_k, and the weight q_k equal to the sum of the weights of all packages composing the commodity. Commodities travel through time and space from their origins to their destinations.

In reality, packages can originate and be due at homes, private businesses, and small pickup locations. However, we only plan the flow of volume from the time it arrives at its pre-specified origin sortation building to its pre-specified destination sortation building. The choice of origin or destination building, sometimes included in other works on service network design, is not included in this model.

In a typical service network design approach, we would assume that each commodity $k \in K$ could be directed independently through the network. That is, putting each commodity $k \in K$ on a time-feasible path to its destination would yield a feasible solution, and that coordination in the planning of the commodities is performed only to increase consolidation and drive down cost. However, in the operations of many sortation hubs, particularly manual ones, commodities may not necessarily be distinguishable. A human worker may not be able to read o_k, d_k, e_k, l_k and quickly recall the outbound door assignment for such packages, for every $k \in K$ that flows through the particular sort shift.

Therefore, we aggregate commodities further into *flow classes*. Each commodity $k \in K$ belongs to a flow class $c_k \in C$. In practice, every $c \in C$ is the set of commodities sharing a common final destination and service class. In operations, each flow class can be assigned to one or more outbound doors at a sort shift; however, which constituents of the class get sent to which door can not be controlled.

The crux of the problem is that the constituents of a flow class at a particular sort may have different due dates, but we may not control which path any constituent actually takes. If there are two possible outbound door assignments for a particular class at a particular sort, we can not assume that the commodities with earlier due dates will be assigned to the door that will result in an earlier delivery. Therefore, every possible downstream path a commodity could possibly take must be time-feasible.

In practice, there exist a set of rules that specifies a default next sort $s_2 \in S$ for every sort $s_1 \in S$ and every flow class $c \in C$. This is called a primary flow rule. The flow rules have been constructed such that if every class follows primary flow rules from its origin sort, it will arrive on-time to its destination. Therefore, they can be used to construct initial feasible solutions, and also to provide possible local changes for improvement heuristics.

Besides determining the possible flow between sorts, we must also specify departure times and arrival times of loads to move between those sorts, and specify how much of each flow class is to be loaded into each load. Oftentimes, if a load departs at the end of its origin sort, it will arrive during its destination sort. If there are capacity issues at the destination sort, then the optimal solution may have the load departing early from the origin sort. Incorporating these decisions into the model is essential for making practically implementable plans.

4.3.1 Time Expanded Modelling

From the geographic network and the sort schedule, we construct a time-expanded network. At each sort, there are a set of induction nodes and dispatch nodes occurring at a regular interval, called the time discretization, and denoted Δ. Packages originating in the sort appear at induction nodes. Induction nodes are connected to dispatch nodes via sort arcs, which are used to model the sort capacity. Dispatch nodes are connected to induction nodes at other sorts via travel arcs, which are used to model the transportation of packages in vehicles.

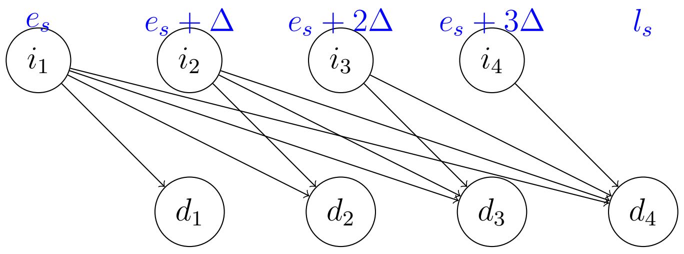


Figure 4.1: Time-space representation of a sort

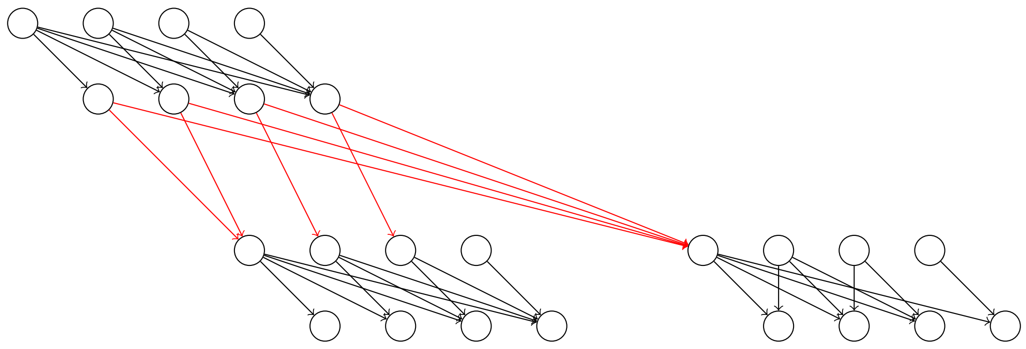


Figure 4.2: Connections between sorts

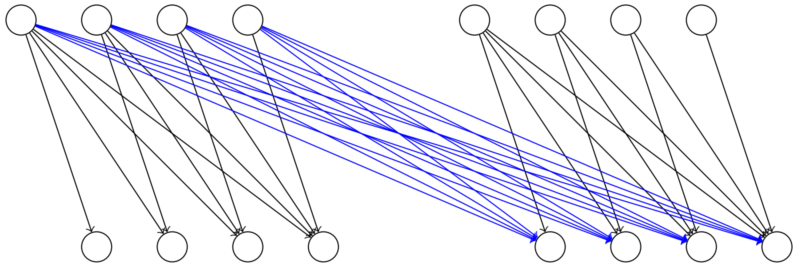


Figure 4.3: Hold arcs

The main benefit of this model over a model that uses a single node for induction and dispatch is that this modelling allows control over the maximum holding time to be represented in the structure of the network. It also allows a visual interpretation of FIFO violations which occur when crossing sort arcs contain the same flow class.

Each commodity can be delivered to its destination at a variety of times before its due time. We denote the set of candidate destination nodes for commodity k as V^k. A typical flow and load planning model that ignores the practical constraints we introduce in this work is given below.

$$\min \sum_{s \in S} [\sum_{a \in A_s^{disp}} c_a y_a + h_s \sum_{a \in A_s^{sort}} \sum_{k \in K} f_a^k]$$

$$\text{s.t.} \sum_{a \in \delta^{out}(v)} f_a^k - \sum_{a \in \delta^{in}} f_a^k = \delta_v^k \qquad \forall v \in V \setminus V^k, k \in K$$

$$\sum_{k \in K} q_k f_a^k \leq Q y_a \qquad \forall a \in A_s^{disp}, s \in S$$

$$0 \leq f_a^k \leq 1 \qquad \forall a \in A_T, k \in K$$

$$y_a \in \mathbb{Z}_+ \qquad \forall a \in A_s^{disp}, s \in S$$

4.3.2 Modelling Flow Rules

The practical application which we wish to model is a system where parcels are directed only by their final destination and service class. We model this more generally by declaring that every commodity $k \in K$ belongs to some flow class $c_k \in C$ that can be directed.

At any sort, packages are directed to a next stop based on their flow class. For every sort $s_1 \in S$, let

$$z^c_{s_1 s_2} = \begin{cases} 1 & \text{if packages of class } c \text{ may travel to sort } s_2 \text{ as a next stop} \\ 0 & \text{otherwise.} \end{cases}$$

This can be enforced through constraints

$$f^k_{uv} \leq z^{c_k}_{s_u s_v} \qquad \forall k \in K, (u,v) \in A^{disp}$$

Suppose $z^c_{s,s_1} = 1$ and $z^c_{s,s_2} = 1$ in some solution. We can not control which packages of class c are directed to s_1 and which are directed to s_2. This is concerning, because packages of class c may have different due dates, depending on how they arrived at s. We model the problem so that any possible assignment of c-flow to s_1 and s_2 must be time-feasible. We assume that we can control the proportion of c-flow sent to each door, but not the specific packages.

We want a necessary and sufficient condition for any possible flow to be time feasible for a given set of flow rules. In order to present this condition, we introduce two additional quantities. For any sort s, let α^c_s represent the minimum due date of class c that *could* flow through s. Then

$$\begin{aligned} \alpha^c_{s_2} &\leq \alpha^c_{s_1} z^c_{s_1 s_2} && \forall (s_1, s_2) \in A^{conn}, c \in C \\ \alpha^{c_k}_s &\leq l_k && \forall k \in K : s_{o_k} = s, s \in S \end{aligned}$$

.These constraints take the minimum due times at the origin of every commodity and propagates this minimum value to downstream sorts that can be reached by active flow rules.

Similarly, for any sort s and any flow class c, let β^c_s be the latest arrival time of any path of flow class c that contains s. Then

$$\beta^c_{s_1} \geq \beta^c_{s_2} z^c_{s_1 s_2} \qquad \forall (s_1, s_2) \in A^{conn}$$

$$\beta^c_s \geq l_s \qquad \forall s \in S$$

We claim that any flow following the flow rules must be time feasible if and only if we can find α and β satisfying the above and

$$\beta^c_s \leq \alpha^c_s \qquad \forall s \in S, c \in C$$

4.3.3 Modelling FIFO Loading

A practically implementable solution must admit a suitable trailer-loading procedure. If two packages x_1 and x_2 arrive at times t_1 and t_2, respectively, with $t_1 < t_2$, and are both to be loaded to the same next destination, then x_2 should never depart strictly earlier than x_1. We constrain our solutions in order to preserve this property by adding conflict constrains for sets of sort arcs.

For any sort $s \in S$, and any sort arc $(u, v) \in A^{sort}_s$, let

$$A^{conflict}_{uv} = \{(x, y) \in A^{sort}_s : t_x < t_u, t_y > t_v\} \cup \{(x, y) \in A^{sort}_s : t_x > t_u, t_y < t_v\}$$

If flow to the same next sort is simultaneously present on a and any arc in $A^{conflict}_a$ then there is a FIFO loading violation.

For a sort arc $(u, v) \in A^{sort}_s$, let

$$g^{s_2}_{uv} = \begin{cases} 1 & \text{if } v \text{ has a dispatch to sort } s_2 \\ 0 & \text{otherwise} \end{cases}$$

Then we can enforce FIFO loading by adding constraints

$$g_a^s + \sum_{a' \in A_a^{conflict}} g_{a'}^s \leq 1 \qquad \forall a \in A_s^{sort}, s \in S \tag{4.1}$$

$$M g_{uv}^{s_2} \geq \sum_{c \in C} \sum_{(v,w) \in \delta^{out}(v): s_w = s_2} \mathbb{1}\{f_{uv}^c > 0, f_{vw}^c > 0\} \quad \forall (u,v) \in A_{s_1}^{sort}, (s_1, s_2) \in A^{conn} \tag{4.2}$$

4.3.4 Modelling Sort Capacity

In order to develop practically implementable plans for package express, the sorting operations must be modelled carefully. Sorts are typically 4-6 hours long, during which vehicles arrive over time and are unloaded. The contents are then loaded onto different outbound vehicles, or onto package cars for local delivery. If not enough vehicles have arrived at the beginning of the sort, then the staff in the building are idle, and the capacity is lost.

$$\sum_{k \in K} \sum_{(x,y) \in A_s^{sort}: t_y \leq t_v, t_x \geq t_u} q_k f_{xy}^k \leq r_s(t_v - t_u) \qquad \forall (u,v) \in A_s^{sort}, s \in S$$

Any sort arc $(u, v) \in A_s^{sort}$ can represent the time interval $[t_u, t_v]$. On the left-hand-side of the constraints above, there is the total volume both inducted and dispatched during the interval $[t_u, t_v]$. The right-hand-side is the sort capacity during the interval.

Since the flow rule modelling guarantees that all flow arrives on time, we can move from flow variables defined on K to flow variables defined on C. Because K contains all origin, release time, destination, and due time based commodities, it is typically much larger that the set C which usually is the product of the set of final destinations with the set of service offerings. An arc based formulation of the problem is given below.

$$
\begin{aligned}
\min \sum_{s\in S}[\sum_{a\in A_s^{disp}} c_a y_a + h_s \sum_{a\in A_s^{sort}} \sum_{c\in C} f_a^c] & \\
\text{s.t.} \sum_{a\in\delta^{out}(v)} f_a^c - \sum_{a\in\delta^{in}} f_a^c = \delta_v^c & \qquad \forall v \in V \setminus V^c, c \in C \\
\sum_{c\in C} f_a^c \le Q y_a & \qquad \forall a \in A_s^{disp}, s \in S \\
f_{uv}^c \le M_{uvc}^1 z_{s_u s_v}^c & \qquad \forall c \in C, (u,v) \in A^{disp} \\
\alpha_{s_1}^c \ge \alpha_{s_2}^c - M^2(1 - z_{s_1 s_2}^c) & \qquad \forall (s_1, s_2) \in A^{conn}, c \in C \\
\alpha_s^c \le l_s^c & \qquad \forall c \in C, s \in S \\
\beta_{s_1}^c \ge \beta_{s_2}^c - M^2(1 - z_{s_1 s_2}^c) & \qquad \forall (s_1, s_2) \in A^{conn}, c \in C \\
\beta_s^c \ge l_s & \qquad \forall s \in S \\
\beta_s^c \le \alpha_s^c & \qquad \forall s \in S, c \in C \\
g_a^s + \sum_{a' \in A_a^{conflict}} g_{a'}^s \le 1 & \qquad \forall a \in A_s^{sort}, s \in S \\
M_v^3 g_{uv}^{s_2} \ge \sum_{c\in C} \sum_{(v,w)\in\delta^{out}(v): s_w = s_2} \mathbb{1}\{f_{uv}^c > 0, f_{vw}^c > 0\} & \qquad \forall (u,v) \in A_{s_1}^{sort}, (s_1, s_2) \in A^{conn} \\
\sum_{c\in C} \sum_{(x,y)\in A_s^{sort}: t_y \le t_v, t_x \ge t_u} f_{xy}^c \le r_s(t_v - t_u) & \qquad \forall (u,v) \in A_s^{sort}, s \in S \\
f_a^c \ge 0 & \qquad \forall a \in A_T, c \in C \\
y_a \in \mathbb{Z}_+ & \qquad \forall a \in A_s^{disp}, s \in S \\
\alpha_s^c, \beta_s^c \in \mathbb{R}_+ & \qquad \forall s \in S, c \in C \\
z_{s_1 s_2}^c \in \{0,1\} & \qquad \forall (s_1, s_2) \in A^{conn}, c \in C \\
g_a^s \in \{0,1\} & \qquad \forall a \in A_s^{sort}, s \in S
\end{aligned}
$$

$$M^1_{uvc} = \min\{Q_c, r_{s_u}(l_{s_u} - e_{s_u}), r_{s_v}(l_{s_v} - e_{s_v})\}$$

$$M^2 = |T|$$

$$M^3_v = |C||\delta^{out}(v)|$$

Note that this model contains the difficulties of service network design which is known to be difficult on instances having 500 or more commodities. In addition, this model contains several more binary variables to model flow rules and FIFO. Therefore, we can conclude that we will not be able to solve this problem exactly using a MIP solver for practically sized instances.

4.4 Improvement Heuristics

Every improvement heuristic we present has local improvement iterations following the same general template:

1. Randomly select a sort $\tilde{s} \in S$
2. Select some subset $\tilde{C}$ of the flow classes processed in $\tilde{s}$
3. For each $n \in V^{induct}_{\tilde{s}}$, remove flow belonging to $\tilde{C}$ from $\tilde{s}$ and downstream sorts by using Algorithm 18. Record the weight of the flow of class c removed downstream from n as o^c_n.
4. Find a new $n \to V^c$ flow of size o^c_n for each $n \in V^{induct}_{\tilde{s}}, c \in \tilde{C}$
5. (If Required) Run FIFOFix procedure to locally resolve FIFO violations, and reject improvement if it violations sort capacity constraints.
6. If the cost increased during iteration, reject improvement.

Algorithm 18 Remove Downstream Flow

```
1: function RemoveFlow($\tilde{s}, \tilde{C}$)
2:     $o_n^c \leftarrow 0 \; \forall \, n \in V_{\tilde{s}}^{induct}, c \in \tilde{C}$        ▷ Flow source after removal
3:     for $n \in V_{\tilde{s}}^{induct}$ do
4:         for $a \in \delta^{out}(n)$ do
5:             for $c \in \tilde{C}$ do
6:                 $o_n^c \leftarrow o_n^c + f_a^c$
7:                 RemoveFlowRecurse($a$, $c$, $f_a^c$)
8:     return $o_n^c \; \forall \, n \in V_{\tilde{s}}^{induct}, c \in \tilde{C}$
9: procedure RemoveFlowRecurse(Time-Space Arc $(u, v)$, Flow-Class $c$, Amount-To-Remove $x$)
10:    $f_{uv}^c \leftarrow f_{uv}^c - x$
11:    if $v \in V^c$ then        ▷ If $v$ is a destination node for $c$
12:        return
13:    $r \leftarrow 0$
14:    for $(v, w) \in \delta^{out}(v)$ do
15:        $z \leftarrow \min\{f_{vw}^c, x - r\}$
16:        RemoveFlowRecurse($(v, w)$, $c$, $z$)
17:        $r \leftarrow r + z$
18:        if $r = x$ then
19:            return
```

A time-feasible primary solution can be constructed by sending every package along a primary sort path from it's origin to it's destination. During this process, the only possible sources of infeasibility are FIFO violations and sort capacity violations. To address the former, we will outline a process that can mend a FIFO violation. To address the latter, it may possible to penalize sort capacity violations in the objective function. However, we assume that we are provided a fully feasible initial solution.

In order to discuss the heuristics, the following additional notation is useful:

- $C(a) := \{c \in C : f_a^c > 0\} \; \forall \, a \in A_T$
- $S^{out}(n) := \{s \in S : \exists (u, v) \in \delta^{out}(n) : s_v = s, \exists c \in C : f_{uv}^c > 0\} \; \forall \, n \in V^{disp}$
- $S^{in}(n) := \{s \in S : \exists (u, v) \in \delta^{in}(n) : s_u = s, \exists c \in C : f_{uv}^c > 0\} \; \forall \, n \in V^{induct}$

- $S^{out}((u,v)) := \{s \in S : \exists (v,w) \in \delta^{out}(v) : s_w = s, \exists c \in C : f^c_{uv} > 0, f^c_{vw} > 0\} \; \forall \; (u,v) \in A^{sort}$
- $S^{out}(s) = \cup_{n \in V^{disp}_s} S^{out}(n) \; \forall \; s \in S$
- $S^{in}(s) = \cup_{n \in V^{induct}_s} S^{in}(n) \; \forall \; s \in S$

$C(a)$ represents the set of flow classes with positive flow on a. The notation $S^{out}(\cdot)$ in general represents the set of sorts that the argument builds to. In the case of a node argument n, $S^{out}(n)$ represents the set of sorts that n dispatches a positive amount of flow to. In the case of a sort arc argument a, $S^{out}(a)$ represents the set of sorts that a could be building to. For any sort arc a, $g^s_a = 0$ is feasible in our model if and only if $s \notin S^{out}(a)$.

We assume that the network has been constructed such that the following property holds

Property 3. *For any dispatch node $n \in V^{disp}$ and any sort $s \in S(n)$, there exists exactly one arc $a = (n, x)$ having $x \in V^{induct}_s$.*

Each improvement heuristic avoids binary modelling for flow rule feasibility and FIFO feasibility. Feasible values of g and z depend not on the quantity flowed, but on which arcs the flow is positive. We have the following useful property for maintaining flow rule feasibility.

Proposition 7. *Fix feasible values $\tilde{\alpha}$ and $\tilde{\beta}$ at their bounds. Adding c-flow to a timed-path $p = (n_1, ..., n_{2m+1})$ following a sort path $p^{sort} = (s_1, ..., s_m)$ will be flow rule feasible if and only if $\min_{j=1,...,i}\{\tilde{\alpha_{s_j}}^c\} \geq \tilde{\beta_{s_i}}^c$ for $i = 1, ..., m$.*

Proof. ($\Leftarrow$) Suppose $\min_{j=1,...,i}\{\tilde{\alpha_{s_j}}^c\} < \tilde{\beta_{s_i}}^c$ for some i. Adding flow cannot increase α or decrease β. Then $\alpha^c_{s_i} \leq \min_{j=1,...,i}\{\tilde{\alpha_{s_j}}^c\} < \tilde{\beta_{s_i}}^c \leq \beta^c_{s_i}$.

($\Rightarrow$) After adding the flow we will have $\alpha^c_{s_i} = \min_{j=1,2,..,i}\{\tilde{\alpha_{s_j}}^c\}$ and $\beta^c_{s_i} = \max_{j=i,i+1,...,m}\{\tilde{\beta_{s_j}}^c\}$. It remains to show that

$$\min_{j=1,...,i}\{\tilde{\alpha_{s_j}}^c\} \geq \tilde{\beta_{s_i}}^c \Rightarrow \min_{j=1,...,i}\{\tilde{\alpha_{s_j}}^c\} \geq \max_{j=i,i+1,...,m}\{\tilde{\beta_{s_j}}^c\}$$

$i = 1, ..., m$ to prove the vertices along the path are flow rule feasible. Take any $i \in \{1, ..., m-1\}$ and $k \in \{i+1, ..., m\}$. Then

$$\min_{j=1,\ldots,i} \{\tilde{\alpha_{s_j}}^c\} \geq \min_{j=1,\ldots,k} \{\tilde{\alpha_{s_j}}^c\} \geq \tilde{\beta_{s_k}}^c$$

from which the above follows.

□

Using this, we may add flows one path at a time, and guarantee that the result will adhere to flow-rule constraints.

Knowing beforehand whether adding c-flow to a new timed path p will violate FIFO is possible, but difficult to describe in adequate notation. Some of the proposed heuristics opt to allow but track FIFO violations, and locally fix the violations using a provided procedure.

4.4.1 Maintaining α and β Labels

The heuristics we introduce use feasible α_s^c and β_s^c values to determine the feasibility of resulting flow. Whenever flow is added or removed from travel arcs, an update to these labels may be necessary. If f_{uv}^c moves from 0 to a positive value, $\alpha_{s_v}^c$ may need to become smaller. This change would then propagate to $S^{out}(s_v)$ and so on. Similarly, $\beta_{s_u}^c$ may need to increase, and this change would propagate to $S^{in}(s_u)$ and so on. The reverse updates may be required if f_{uv}^c moves from a positive value to 0. Algorithm 19 contains pseudocode for how these updates could be implemented.

Algorithm 19 α, β updates

```
1: procedure UPDATEALPHA(Sort $s$, Flow class $c$)
2:     $\tilde{\alpha} \leftarrow \min\{l_s^c, \min_{s' \in S^{in}(s)}\{\alpha_{s'}^c\}\}$
3:     if $\alpha_s^c = \tilde{\alpha}$ then
4:         return
5:     $\alpha_s^c \leftarrow \tilde{\alpha}$
6:     for $s' \in S^{out}(s)$ do
7:         UpdateAlpha($s'$, $c$)
8: procedure UPDATEBETA(Sort $s$, Flow class $c$)
9:     $\tilde{\beta} \leftarrow \max\{l_s, \max_{s' \in S^{out}(s)}\{\beta_{s'}^c\}\}$
10:    if $\beta_s^c = \tilde{\beta}$ then
11:        return
12:    $\beta_s^c \leftarrow \tilde{\beta}$
13:    for $s' \in S^{in}(s)$ do
14:        UpdateBeta($s'$, $c$)
```

4.4.2 Maintaining g_a^s labels

Similarly, in order to detect FIFO violations so that they may be fixed, the improvement heuristics presented may make use of a feasible set of $g_a^s, a \in A_s^{sort}, s \in S$ values. The value of g_a^s depends not only on the flows on $a = (u, v)$, but also on the arc (v, w) where $s_w = s$. Whenever flow moves from zero to positive, or vice versa, on any arc (sort, travel, or hold), the values of g may need to be updated. We outline the updates for each case below.

- If f_a^c moves from zero to positive and a is a sort arc, then g_a^s should be set to 1 for all s where there exists an arc (v, w) with $s_w = s$ and $f_{vw}^c > 0$.
- If f_a^c moves from zero to positive and $a = (v, w)$ is a travel arc, then for all $a' = (u, v) \in A_{s_v}^{sort}$ where $f_{a'}^c > 0$, $g_{a'}^{s_v}$ should be set to 1.
- If f_a^c moves from positive to zero and $a = (u, v)$ is a sort arc, then consider all s where $g_a^s = 1$. Consider the unique arc $a' = (v, w)$ where $s_w = s$. If $C(a) \cap C(a') \neq \emptyset$ set $g_a^s = 0$.
- If f_a^c moves from positive to zero and $a = (v, w)$ is a dispatch arc, then consider all $a' = (u, v)$ where $g_{a'}^{s_w} = 1$. If $C(a) \cap C(a') = \emptyset$, set $g_{a'}^{s_w} = 0$.

4.4.3 FIFOFix Procedure

Consider a general FIFO violation from arcs $a_1, a_2 \in A^{sort}_{s_1}$ where $a_2 \in A^{conflict}_{a_1}$ and $g^{s_2}_{a_1} = g^{s_2}_{a_2} = 1$. The situation is depicted in Figure 4.4. For clarity note that the arcs d_1 and d_2 are the only travel arcs, they connect s_1 to s_2, and that $u_1, u_2, u_3 \in V^{induct}_{s_1}$, $v_1, v_2 \in V^{disp}_{s_1}$, $w_1, w_2 \in V^{induct}_{s_2}$, and $x_1, x_2, x_3 \in V^{disp}_{s_2}$. Note that according to Property 3, d_2 is the only arc connecting v_2 to s_2. Thus, we know $C(a_2) \cap C(d_2) \neq \emptyset$. If we alter the flow such that $C(a_2) \cap C(d_2) = \emptyset$, then $g^{s_2}_{a_2} = 0$ would be feasible. The FIFOFix procedure takes flow of a class in $C(a_2) \cap C(d_2)$ traveling on the subpath (a_2, d_2, y_i), and moves it to the subpath (a_3, d_1, y'_i). Note the origin and destination nodes of these subpaths are identical. The procedure continues moving flow until $C(a_2) \cap C(d_2) = \emptyset$.

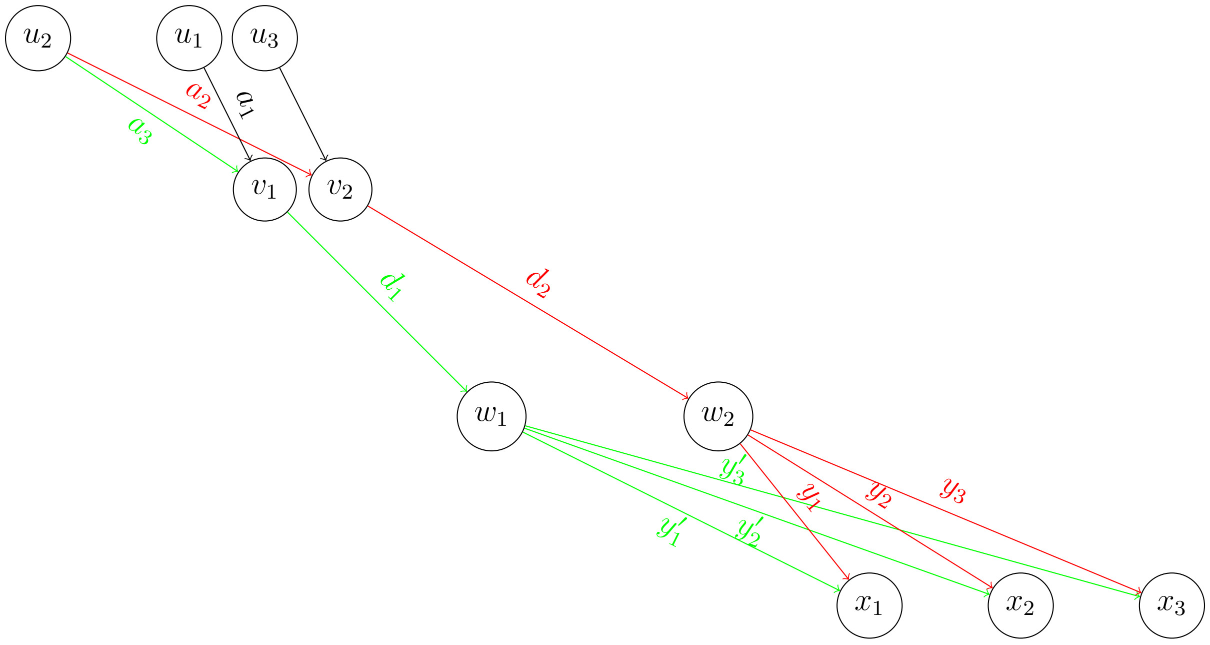


Figure 4.4: Resolving FIFO violations with FIFOFix: flow moves from red to green subpaths

Algorithm 20 FIFO Fix Procedure

```
1: procedure FIFOFix(Sort arcs violating FIFO $a_1 = (u_1, v_1), a_2 = (u_2, v_2)$)    ▷ Assume WLOG $t_{v_2} > t_{v_1}$
2:     for $s \in S^{out}(a_1) \cap S^{out}(a_2)$ do
3:         FIFOFixSort($a_1, a_2, s$)
4: procedure FIFOFixSort(Sort arcs violating FIFO $a_1 = (u_1, v_1), a_2 = (u_2, v_2)$, sort $s$)
5:     Let $d_1 = (v_1, w_1)$ denote the unique $v_1 - s$ arc
6:     Let $d_2 = (v_2, w_2)$ denote the unique $v_2 - s$ arc
7:     Let $a_3 = (u_2, v_1)$
8:     for $c \in C(a_2) \cap C(d_2)$ do
9:         $F \leftarrow \min\{f^c_{a_2}, f^c_{d_2}\}$
10:        $f^c_{a_2} \leftarrow f^c_{a_2} - F$
11:        $f^c_{d_2} \leftarrow f^c_{d_2} - F$
12:        $f^c_{a_3} \leftarrow f^c_{a_3} + F$
13:        $f^c_{d_1} \leftarrow f^c_{d_1} + F$
14:        $r \leftarrow 0$                    ▷ Tracks flow removed downstream from $w_2$
15:        for $y \in \delta^{out}(w_2) : C(y) \ni c$ do
16:            Let $x$ denote the head of $y$
17:            Let $y' = (w_1, x)$
18:            $f \leftarrow \min\{F - r, f^c_y\}$
19:            $f^c_y \leftarrow f^c_y - f$
20:            $f^c_{y'} \leftarrow f^c_{y'} + f$
21:            $r \leftarrow r + f$
```

Proposition 8. *After the procedure FIFOFix, $g^s_{a_2} = 0$ is feasible for all $s \in S^{out}(a_1)$.*

Proof. After the procedure FIFOFixSort, $C(a_2) \cap C(d_2) = \emptyset$. Then $S^{out}(a_2) \not\ni s$, because d_2 is the unique arc connecting v_2 to s. Since this is performed for all $s \in S^{out}(a_1) \cap S^{out}(a_2)$, we can conclude that $S^{out}(a_1) \cap S^{out}(a_2) = \emptyset$ after FIFOFix. □

Proposition 9. *After finitely many iterations of FIFOFix, performed in arbitrary order, the solution will contain no FIFO violations.*

Proof. FIFOFix pairwise moves positive quantities of flow from subpaths (a_2, d_2, y_i) to subpaths (a_1, d_1, y_i) where the end nodes are identical, but the internal nodes of the subpath (a_1, d_1, y_i) are strictly early than the internal nodes of (a_2, d_2, y_i). Thus, FIFOFix could not run for an infinite amount of iterations, because all flow would eventually be allocated to the earliest sort arcs reachable by a flow source, and such a solution is guaranteed to be FIFO feasible. □

4.4.4 Alt

This heuristic is implemented as follows: For step (2) of the generic local improvement procedure, this heuristic randomly selects a destination sort $s' \in S^{out}(\tilde{s})$ and chooses $\tilde{C}$ to be the set of classes having positive flow on any arc connecting $\tilde{s}$ to s'. To add this flow back, the algorithm iterates over $c \in \tilde{C}$, and then iterates over any possible s'' that $\tilde{s}$ could build to. It then records a sort path $p^{sort} = (\tilde{s}, s'', s_1, s_2, .., s_m)$ where $(s'', s_1, s_2, ..., s_m)$ is the primary sort path from s'' to V^c. If this sort path is feasible according to the conditions of Proposition 7 it continues; otherwise, the algorithm skips to the next possible s''. Next, the algorithm iterates over n where $o_n^c > 0$ and finds a cheapest cost path of capacity at least o_n^c following the sort path p^{sort}, records this path as $p_{s''}^{nc}$, adds the flow to this path, and continues. The algorithm uses the sum of the costs of these flow paths as the joint price of *alting* to s'', removes the flow from the candidate paths $p_{s''}^{nc}$, and records this cost as $c_{s''}$. The algorithm then chooses $s^* \in \arg\min_{s''}\{c_{s''}\}$, and re-adds o_n^c units of c-flow to the paths $p_{s''}^{nc}$. The practicality of this approach is that it adds at most $|\tilde{C}|$ new flow rules, simplifying the changes required to operate the new plan.

Algorithm 21 Alt Heuristic

```
1: procedure AltHeuristic(Sort $\tilde{s}$, Flow classes $\tilde{C}$, Flow source $o_n^c$ for $n \in V_{\tilde{s}}^{induct}, c \in \tilde{C}$)
2:     for $c \in \tilde{C}$ do
3:         for $(\tilde{s}, s'') \in \delta_{sort}^{out}(\tilde{s}$ do
4:             $(s'', s_1, s_2, ..., s_m) \leftarrow$PrimarySortPath$(s'', c)$
5:             $p_{s''}^{sort} \leftarrow (\tilde{s}, s'', s_1, s_2, ..., s_m)$
6:             if $\alpha_{\tilde{s}}^c \geq \beta_{s''}^c$ and $\min\{\alpha_{\tilde{s}}^c, \alpha_{s''}^c, \min_{j=1,...,i}\{\alpha_{s_i}^c\}\} \geq \beta_{s_i}^c$ for $i = 1, ..., m$ then
7:                 $cost \leftarrow$ CurrentPlanCost()
8:                 for $n \in V_{\tilde{s}}^{induct} : o_n^c > 0$ do
9:                     $p_{s''}^{nc} \leftarrow$FindMinCostPathFollowingSortPath$(n, o_n^c, p_{s''}^{sort}, c)$
10:                    for $a \in p_{s''}^{nc}$ do
11:                        $f_a^c \leftarrow f_a^c + o_n^c$
12:                $\Delta C_{s''} \leftarrow$CurrentPlanCost() $-cost$
13:                for $n \in V_{\tilde{s}}^{induct} : o_n^c > 0$ do
14:                    for $a \in p_{s''}^{nc}$ do
15:                        $f_a^c \leftarrow f_a^c - o_n^c$
16:            else
17:                $\Delta C_{s''} \leftarrow \infty$
18:        Take $s^* \in \arg\min_{s''}\{\Delta C_{s''}\}$
19:        for $n \in V_{\tilde{s}}^{induct} : o_n^c > 0$ do
20:            for $a \in p_{s^*}^{nc}$ do
21:                $f_a^c \leftarrow f_a^c + o_n^c$
```

```
22: function FINDMINCOSTPATHFOLLOWINGSORTPATH(Origin node $o \in V_T$, Flow Quantity
    $o_n^c$, Sort Path $p_{s''}^{sort} = (s_1, s_2, ..., s_m)$, Flow Class $c$)
23:     Next($s_{i+1}$) $\leftarrow s_i$ for $i = 1, 2, ..., m-1$
24:     $l(n) \leftarrow \infty \; \forall \, n \in V_T$
25:     $l(o) \leftarrow 0$
26:     $U \leftarrow \{o\}$
27:     while $U \neq \emptyset$ do
28:         Take $u \in \arg\min_{u \in U}\{l(u)\}$
29:         $U \leftarrow U \setminus \{u\}$
30:         if $u \in V_{s_m}^{disp}$ then
31:             return BackTrack($parent, u, o$)
32:         if $u \in V_{s_u}^{induct}$ then
33:             $s^{next} \leftarrow s_u$
34:         else
35:             $s^{next} \leftarrow$Next($s_u$)
36:         for $(u, v) \in \delta^{out}(u) : s_v - s^{next}$ do
37:             $cost \leftarrow$ MarginalCost $((u, v), o_n^c, c)$
38:             if $l(u) + cost < l(v)$ then
39:                 $parent(v) \leftarrow u$
40:                 $l(v) \leftarrow l(u) + cost$
41:                 $U \leftarrow U \cup \{v\}$
```

4.4.5 Reflow

We propose another local improvement heuristic for this problem that also avoids incorporating the z_{uv}^{dc} variables into the model. At each iteration, a building b, sort $s \in S_b$, and a subset of the destination-service classes with positive flow through s, denoted $\tilde{C}$, are somehow chosen. All flow belonging to a class in $\tilde{C}$ moving through s is removed at the current sort s and from all arcs downstream from s. The pseudo-code for flow removal is shown in Algorithm 18.

Once the flow is removed, the goal is to find a new flow that is flow rule feasible, sort feasible, loads vehicles in FIFO order, and delivers each flow class on time. We rely on a heuristic procedure to add flow back iteratively, class by class. We specify a minimum-splitable-quantity q. For each class, we find minimum marginal cost path that can accommodate at least q and respects due times, FIFO, and flow rule feasibility. We then saturate this path, and continue finding more if necessary.

The pseudocode for this is given in Algorithm 22.

Algorithm 22 Reflow Loop

```
1: function FLOWLOOP(TimeSpaceNode $v$, Flow-Class $c$, Amount-To-Path $q$)
2:     $flowed \leftarrow 0$
3:     while $flowed \neq q$ do
4:         $q' \leftarrow \begin{cases} q - flowed & \text{if } (q - flowed) < 2q' \\ q' & \text{otherwise} \end{cases}$
5:         $p \leftarrow$FINDPATH$(v, c, q')$
6:         $y \leftarrow$ minimum available capacity on $p$
7:         if $y \geq q - flowed$ then
8:             $f \leftarrow q - flowed$
9:         else
10:            $r \leftarrow q - flowed - y$
11:            if $r < q'$ then
12:                $f \leftarrow y - (q' - r)$
13:            else
14:                $f \leftarrow y$
15:        Add $f$ units of flow to $p$
16:        $flowed \leftarrow flowed + f$
```

We can find the minimum marginal cost path satisfying the conditions of Proposition 7 by solving a shortest path problem on an expanded state space. Then we can saturate the path, update α, β, and the marginal costs and repeat until the entire flow-class has been added back.

Finding a path satisfying the conditions of Proposition 7 can be accomplished by a labelling algorithm on a state space of (b, a, α, c), where b is a building, a is an arrival time, α is the due time of the flow, and c is the cost at which this state can be reached. When moving from one state to another, the arrival time and cost increase, and the due time of the flow decreases. The pseudocode for this algorithm is given in Algorithm 23.

At each iteration, a single flow class c is removed from the solution and re-flowed. When the flow class is removed, the values α and β are updated. Our heuristic works by sequentially introducing paths satisfying this property to the flow basis, and updating the α and β.

Algorithm 23 Path Finding Algorithm

```
1: function FINDPATH(TimeSpaceNode $v$, Flow-Class $k$, Amount-To-Path $q$)
2:     $U \leftarrow \{(b_v, t_v, 0)\}$
3:     while $U \neq \emptyset$ do
4:         Take $(u, \alpha, c) \in \arg\min_{(b', a', c') \in U}\{c'\}$
5:         if $u = (d_k, t) : t \geq \alpha$ then
6:             return BackTrack$(u, \alpha, c)$
7:         $U \leftarrow U \setminus \{(u, \alpha, c)\}$
8:         for $(u, v) \in \delta^{out}(u)$ do
9:             $\bar{\alpha} \leftarrow \min\{\alpha^k_{s_v}, \alpha\}$
10:            $\bar{c} \leftarrow c+$MarginalCost$((u, v), q, k)$
11:            if $\bar{\alpha} \geq \beta^k_{s_v}$ then                    ▷ If state is flow rule feasible
12:                $U \leftarrow U \cup \{(v, \bar{\alpha}, \bar{c})\}$
13: function MARGINALCOST(TimeSpaceArc $(u, v)$, Quantity $q$, Flow-Class $k$)
14:     $c \leftarrow 0$
15:     if $(u, v) \in A^{sort}_{s_u}$ then
16:         $c \leftarrow c + hq$
17:         $o \leftarrow$SortViolation$((u, v), q)$
18:         if $o > 0$ then
19:             $c \leftarrow c + M$
20:         if $\exists k \in K, a \in A^{conflict}_{(u,v)} : f^k_a > 0$ then
21:             $c \leftarrow c + M$
22:     else
23:         $c \leftarrow c+$
24:     return 0
25: function SORTVIOLATION(TimeSpaceArc $(\mu, \nu)$, Quantity $q$)
26:     $o \leftarrow 0$
27:     for $(u, v) \in A^{sort}_{\mu}$ do
28:         $q_{uv} \leftarrow \begin{cases} q & \text{if } t_u \leq t_\mu \text{ and } t_v \geq t_\nu \\ 0 & \text{otherwise} \end{cases}$
29:         $o \leftarrow o + \max\{q_{uv} + \sum_{k \in K} \sum_{(x,y) \in A^{sort}_s : t_y \leq t_v, t_x \geq t_u} q_k f^k_{xy} - r_s(t_v - t_u), 0\}$
30:     return $o$
```

4.4.6 MIP-Based Heuristics

The bulk of the computational difficulty of using mixed integer programming to find feasible paths arises from the variables **z** and **g** used to model flow rule feasibility and FIFO feasibility. However, we have demonstrated and used Proposition 7 to maintain flow rule feasibility in other heuristics.

The core idea of the proposed MIP-based heuristics is to construct path sets $\tilde{P}^c$ for $c \in \tilde{C}$ connecting $\tilde{s}$ to V^c, the destination vertices for c. The set of paths $\tilde{P}^c$ will be "jointly" flow rule feasible and FIFO feasible. That is, even if there was positive flow added to *every* path in $\tilde{P}^c$, the resulting plan would be flow rule feasible and FIFO feasible. Furthermore, if we guarantee $\tilde{P}^c$ contains every path from which flow was removed in step 3 of the general improvement heuristic template, we can guarantee that a path-based MIP using variables restricted to $\tilde{P}^c$ could recover the initial feasible solution.

Optimizing on such a restricted set of paths (those that are jointly flow rule and FIFO feasible) can limit the possible improvement at each iteration, but allows us to model the problem as the MIP below. This MIP completely avoids the difficult variables and constraints. In the instances we solve, Gurobi can solve this MIP to optimality for reasonable sized $\tilde{C}$ in seconds.

$$\min \sum_{s \in S} \Big[\sum_{a \in A_s^{disp}} c_a y_a + h_s \sum_{a \in A_s^{sort}} \sum_{c \in C} \sum_{p \in \tilde{P}^c : p \ni a} f_p^c \Big]$$

$$\text{s.t.} \quad \sum_{p \in \tilde{P}^c : p = (i, n_2, n_3, \ldots, n_m)} f_p^c = \delta_i^c \qquad \forall i \in N, c \in C$$

$$Q y_a \geq \sum_{c \in C} \sum_{p \in \tilde{P}^c : p \ni a} f_p^c \qquad \forall a \in A_s^{disp}, s \in S$$

$$\sum_{c \in C} \sum_{(x,y) \in A_s^{sort}} \sum_{p \in \tilde{P}^c : p \ni xy} \sum_{: t_y \leq t_v, t_x \geq t_u} f_p^c \leq r_s (t_v - t_u) \qquad \forall (u,v) \in A_s^{sort}, s \in S$$

$$f_p^c \geq 0 \qquad \forall p \in \tilde{P}^c, c \in C$$

$$y_a \in \mathbb{Z}_+ \qquad \forall a \in A_s^{disp}, s \in S$$

Key to the success or failure of this approach is the method of constructing $\tilde{P}^c$. We provide two possible implementations for testing, but the design of better methods may be warranted. Both implementations initialize $\tilde{P}^c$ to the set of paths from which flow is removed in Step 3 of the general improvement heuristic template. They then use temporary $\alpha, \beta,$ and g values to sequentially add paths to $\tilde{P}^c$ that are guaranteed to be flow rule and FIFO feasible. To do this, one can assume there is positive flow on every path in $\tilde{P}^c$ and set the temporary $\alpha, \beta,$ and g accordingly, and sequentially add paths satisfying the conditions of Proposition 7 and Equations 4.1 and 4.2.

The first implementation, denoted AltMIP, runs the entire alt heuristic, undoes the changes, and sequentially attempts to add every path $p_{s^*}^{nc}$ to $\tilde{P}^c$. The second implementation, denoted ReflowMIP, runs the entire Reflow heuristic, undoes the changes, and sequentially attempts to add every path generated by the heuristic to $\tilde{P}^c$. These implementations will take strictly more time than using the underlying heuristics; however, the hope is that larger cost savings will accrue by allowing the optimization problem to encompass joint allocation of flow across multiple paths instead of greedily adding flow a single path at a time. They also have the benefit that they are guaranteed to produce a FIFO and sort feasible solution at each iteration. For Alt and Reflow, we must check for violations at each iteration and run FIFOFix or reject the solution if there are violations.

4.5 Computational Experiments

In order to test the effectiveness of the proposed solution approaches, we use several industry instances outline in Table 4.1. These instances contain actual demand from a large parcel express company for three different weeks. To create the instances, all packages originating and due within a pre-specified region are included. We consider two regions: South and East. South contains all states roughly bounded by Alabama, North Carolina, and Florida. East includes all states roughly bounded by Louisiana, Kentucky, Michigan, Maine, and Florida. The number of commodities, flow classes, and buildings contained in each instance is shown in the table. The time horizon is induced by the earliest time a package originates until the latest time at which a package is due. Each instance contains one week of package originations.

Table 4.1: Instance characteristics

Week	Scope	$\|K\|$	$\|C\|$	$\|B\|$	Horizon Start	Horizon End	Primary Flow Rules
6/12	South	366,525	1,060	527	6/6	6/21	85,740
6/12	East	2,490,498	2,919	1,373	6/6	7/23	281,917
6/19	South	366,303	1,063	527	6/13	6/28	85,740
6/19	East	2,495,337	2,914	1,373	6/13	7/30	281,917
6/26	South	365,707	1,058	527	6/20	7/6	85,740
6/26	East	2,499,377	2,913	1,373	6/20	8/6	281,917

For these instances, we construct a baseline solution by sending each commodity on a minimum marginal cost path following the primary sort path of its flow class. Each improvement heuristic begins with this common plan as initial solution and is given one hour to improve it.

Because the Reflow heuristic contains a minimum-splitable-quantity parameter, we first present results of varying this parameter. We then fix this parameter and explore varying the distribution of the flow classes chosen to re-optimize in Reflow and ReflowMIP. We compare these plans with the plans produced by Alt and AltMIP.

4.5.1 Varying Minimum-Splitable-Quantity

We first vary the minimum-splitable-quantity parameter of Reflow among $\{200, 500, 1000, 2500\}$. A minimum-splitable-quantity of 1000 means that Reflow will only split flow across two or more flow paths if each flow path contains at least 1000 cubic inches of flow.

Table 4.2 shows the cost, trailer miles, and sortation savings after an hour of improvement with each parameter setting on each instance. We first note that the value of the parameter has a negligible impact of the savings. Table 4.3 shows the value of the parameter that produced the cheapest plan on each instance. We postpone a general discussion of the results to the next section and fix the parameter value to 1000 in all future experiments.

Table 4.2: Varying minimum splitable quantity

Instance week	Instance Scope	MinSplitQuant	Num. Iter.	Cost Savings	Trailer Miles Savings	Volume Sorted Savings
6/12	East	200	385	0.069%	0.132%	0.003%
6/12	East	500	424	0.101%	0.208%	-0.009%
6/12	East	1000	398	0.102%	0.200%	0.002%
6/12	East	2500	382	0.094%	0.180%	0.002%
6/12	South	200	2732	1.517%	3.069%	-0.007%
6/12	South	500	2826	1.737%	3.558%	-0.044%
6/12	South	1000	2760	1.625%	3.292%	-0.004%
6/12	South	2500	2793	1.673%	3.398%	-0.014%
6/19	East	200	403	0.059%	0.116%	-0.002%
6/19	East	500	396	0.083%	0.159%	0.000%
6/19	East	1000	410	0.098%	0.187%	0.002%
6/19	East	2500	401	0.082%	0.158%	0.001%
6/19	South	200	2810	1.488%	2.978%	-0.003%
6/19	South	500	2821	1.636%	3.252%	0.017%
6/19	South	1000	2779	1.557%	3.109%	0.007%
6/19	South	2500	2726	1.562%	3.136%	0.009%
6/26	East	200	367	0.076%	0.149%	0.001%
6/26	East	500	394	0.096%	0.191%	-0.001%
6/26	East	1000	382	0.053%	0.098%	0.003%
6/26	East	2500	397	0.086%	0.168%	0.002%
6/26	South	200	2753	1.615%	3.333%	-0.027%
6/26	South	500	2800	1.502%	3.065%	-0.008%
6/26	South	1000	2808	1.775%	3.641%	-0.030%
6/26	South	2500	2806	1.552%	3.161%	0.022%

Table 4.3: Best splitable quantity

Instance week	Instance Scope	Best Splitable Quantity
6/12	East	1000
6/12	South	500
6/19	East	1000
6/19	South	500
6/26	East	500
6/26	South	1000

4.5.2 Varying Distribution of Classes to Reflow

At each iteration, Reflow and ReflowMIP select a sort, $\tilde{s}$, and a subset of the flow classes present at $\tilde{s}$, $\tilde{C}$. The heuristics re-optimize the flow of all classes belong to $\tilde{C}$ downstream from $\tilde{s}$.

We generate $\tilde{C}$ as follows. Let $C_{\tilde{s}}$ be the set of flow classes present at $\tilde{s}$. We specify a parameter N^{max}, draw a random integer N uniformly from $\{1, 2, ..., \min\{N^{max}, |C_{\tilde{s}}|\}\}$ and select $\tilde{C}$ to be N random elements of $C_{\tilde{s}}$ chosen with equal probability. We vary the value of N^{max} among $\{50, 150, 500\}$ and compare against the plans produced by Alt and AltMIP.

Table 4.4 shows the number of iterations performed, as well as the cost, trailer-miles, and sortation savings for the plans produced by each method on each instance. We first note that as N^{max} is increased, the number of iterations completed during an hour reduces. This indicates that the time to complete each improvement iteration increases with N^{max}. Despite this, there does not appear to be a clear relationship between N^{max} and the overall cost savings. There is some benefit to using larger N^{max} for Reflow on the South region instances, but this does not hold for the East region.

Table 4.4: Varying distribution of classes to reflow

Instance Week	Instance Scope	Method	N^{max}	Num. Iter.	Cost Savings	Trailer Miles Savings	Volume Sorted Savings
6/12	East	Reflow	50	297	0.122%	0.238%	-0.001%
6/12	East	ReflowMIP	50	412	0.047%	0.080%	0.005%
6/12	East	Reflow	150	131	0.081%	0.152%	0.005%
6/12	East	ReflowMIP	150	157	0.074%	0.141%	0.006%
6/12	East	Reflow	500	55	0.081%	0.160%	0.001%
6/12	East	ReflowMIP	500	66	0.082%	0.156%	0.006%
6/12	East	Alt	NA	742	0.263%	0.601%	-0.064%
6/12	East	Alt MIP	NA	5092	0.355%	0.619%	0.089%
6/12	South	Reflow	50	1989	1.708%	3.487%	-0.027%
6/12	South	ReflowMIP	50	2819	0.927%	1.763%	0.096%
6/12	South	Reflow	150	859	1.966%	4.236%	-0.211%
6/12	South	ReflowMIP	150	996	1.071%	2.060%	0.094%
6/12	South	Reflow	500	357	2.601%	5.960%	-0.580%
6/12	South	ReflowMIP	500	404	0.566%	1.097%	0.066%
6/12	South	Alt	NA	5406	3.218%	7.978%	-1.225%
6/12	South	Alt MIP	NA	18317	0.934%	1.528%	0.368%
6/19	East	Reflow	50	303	0.099%	0.193%	-0.003%
6/19	East	ReflowMIP	50	392	0.034%	0.057%	0.006%
6/19	East	Reflow	150	147	0.110%	0.214%	0.000%
6/19	East	ReflowMIP	150	189	0.048%	0.084%	0.007%
6/19	East	Reflow	500	60	0.088%	0.171%	-0.002%
6/19	East	ReflowMIP	500	56	0.042%	0.079%	0.004%
6/19	East	Alt	NA	755	0.286%	0.635%	-0.064%
6/19	East	Alt MIP	NA	5323	0.246%	0.432%	0.055%
6/19	South	Reflow	50	1983	1.858%	3.755%	-0.014%
6/19	South	ReflowMIP	50	2794	1.060%	2.009%	0.091%
6/19	South	Reflow	150	911	2.428%	5.112%	-0.204%
6/19	South	ReflowMIP	150	1050	0.841%	1.597%	0.074%
6/19	South	Reflow	500	368	2.364%	5.285%	-0.436%
6/19	South	ReflowMIP	500	407	0.782%	1.525%	0.067%
6/19	South	Alt	NA	5379	3.445%	8.588%	-1.445%
6/19	South	Alt MIP	NA	18539	1.013%	1.696%	0.356%
6/26	East	Reflow	50	235	0.081%	0.158%	0.000%
6/26	East	ReflowMIP	50	415	0.057%	0.103%	0.006%
6/26	East	Reflow	150	133	0.105%	0.209%	0.000%
6/26	East	ReflowMIP	150	159	0.056%	0.105%	0.006%
6/26	East	Reflow	500	61	0.087%	0.174%	0.001%
6/26	East	ReflowMIP	500	64	0.017%	0.026%	0.007%
6/26	East	Alt	NA	735	0.236%	0.545%	-0.065%
6/26	East	Alt MIP	NA	5217	0.255%	0.438%	0.073%
6/26	South	Reflow	50	2008	1.878%	3.869%	-0.040%
6/26	South	ReflowMIP	50	2851	0.854%	1.638%	0.082%
6/26	South	Reflow	150	884	2.126%	4.572%	-0.212%
6/26	South	ReflowMIP	150	1024	0.646%	1.234%	0.079%
6/26	South	Reflow	500	348	2.668%	6.136%	-0.575%
6/26	South	ReflowMIP	500	393	0.901%	1.819%	0.047%
6/26	South	Alt	NA	5372	3.321%	8.310%	-1.298%
6/26	South	Alt MIP	NA	18696	0.831%	1.411%	0.296%

One result at first glance appears to be counter intuitive: the MIP-based methods are performing more iterations in under one hour than the non-MIP methods. This is due to the fact that the MIP-based methods can not produce a FIFO or sort infeasible improvement. In the implementation of the non-MIP methods, there is an expensive check after each iteration to verify that the changes are feasible. If the changes are infeasible, another step reverts to the solution at the beginning of the iteration.

Next, we consider the performance of each method in general. The best performing heuristics are Alt and AltMIP. There could be various causes of this. The first is that these methods free up all flow classes moving between a pair of sorts, potentially increasing the likelihood that a trailer can be removed from the solution; Reflow and ReflowMIP can choose $\tilde{C}$ such that each trailer contains some element in $C \setminus \tilde{C}$. A second potential factor is that Alt and AltMIP generate paths following the primary sort path of each flow class downstream from a chosen next sort. It may be that this restriction can guide the heuristic to better plans given that primary sort paths were designed manually for this purpose over many years.

In general, we see the savings are proportionally better on the smaller instances. This disappointing result is likely due to the fact that the improvement is local, leading to some constant rate of savings over time. Despite this, Alt can produce 3.218%-3.445% cost savings and 7.987%-8.588% trailer miles savings on the South instances in under one hour. This is very good performance considering the scale of the instances being solved.

We now consider the performance gap between AltMIP and ReflowMIP. As we have already stated, the performance gap could be attributed to the choice of $\tilde{C}$ or the restriction of the flows generated. However, there are other differences among these methods. Figure 4.5 shows the distribution of the number of new paths added to the restricted path set and this number plotted against $|\tilde{C}|$ for ReflowMIP with $N^{max} = 500$ on the 6/19 South instance. Figure 4.6 shows the same for AltMIP. We see that at each iteration, ReflowMIP is adding significantly more paths to the restricted path set compared to AltMIP. This, combined with the fact that Alt is faster than Reflow, allows AltMIP to complete significantly more improvement iterations in an hour than ReflowMIP.

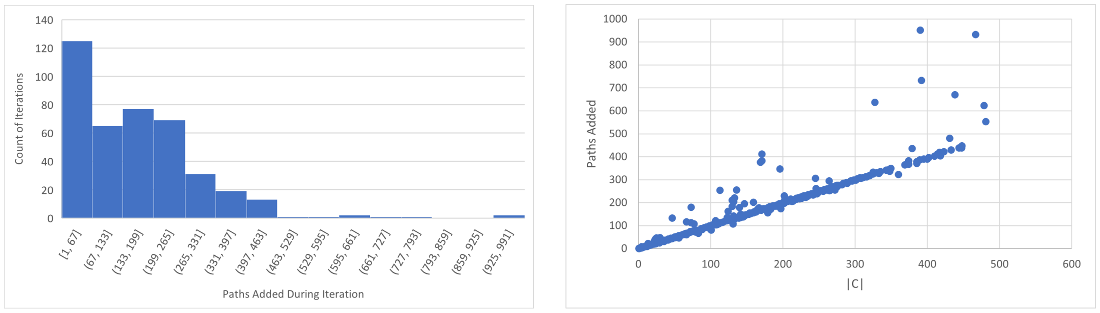


Figure 4.5: Paths added to flow basis: ReflowMIP

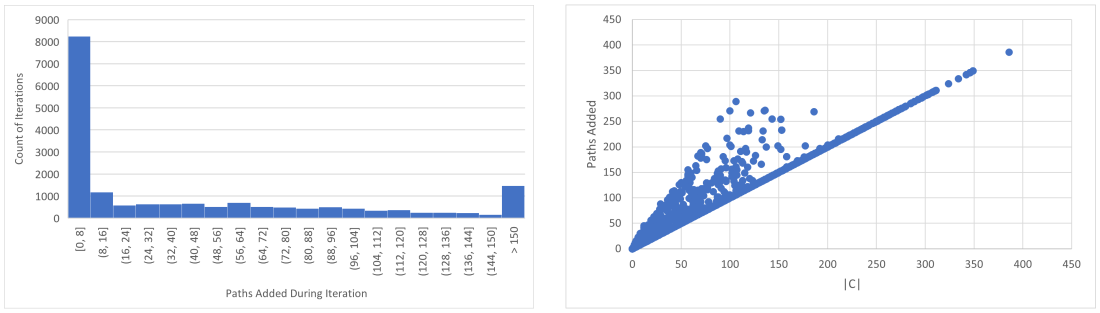


Figure 4.6: Paths added to flow basis: AltMIP

4.6 Conclusion

There is a great need for fast automated planning and decision support tools in the parcel express and E-retail industries. Plans need to be adjusted quickly in the scenarios of weather shutdowns or demand shocks. However, many solution approaches for flow and loadplanning ignore practical constrains like sort capacity and FIFO loading, or assume sufficient automation exists in every building to direct each commodity independently. In industry instances containing thousands of buildings, millions of commodities, and tens of millions of packages, many buildings are not equipped with such automation, and thus plans output by existing tools are not directly implementable.

In this work, we have presented a model for generating flow and load plans where commodities are aggregated into flow classes that may be independently directed at every building. Our model produces plans that guarantee each package will be delivered on time despite the aggregated level of control. This model is too unwieldy for realistically sized instances, and no known heuristics can be adapted to this environment. Therefore, we propose and compare four improvement heuristics capable of preserving the feasibility and desired practicality of the plans. All four heuristics rely on a key property we point out that allows flow to be added one path at a time to a solution in such a way that flow-rule-feasibility is preserved. One of these heuristics, Alt, was shown to produce cost savings of up to 3.4% and savings in trailer-miles of up to 8.5% on large scale industry instances in under one hour.

Appendices

APPENDIX A

MARGINAL COST PATHING APPENDIX

A.1 Capacity Table Dynamic Program

Algorithm 24 Capacity Table Construction

```
 1: function BuildCapacityTable
 2:     for $a \in A$ do
 3:         $y_a \leftarrow 0 \in \mathbb{Z}^{|V_a|}$
 4:         $Y_a \leftarrow \{\}$
 5:         Recurse$(Q_a^{max}, y_a, Y_a)$
 6:     return $\{Y_a\}_{a \in A}$
 7: procedure Recurse$(Q_a^{max}, y, Y)$
 8:     $Q \leftarrow \sum_{v \in V_a} Q_v y_v$
 9:     $c \leftarrow \sum_{v \in V_a} C_{av} y_v$
10:     for $v \in V_a$ do
11:         $c' \leftarrow C + C_{av}$
12:         $Q' \leftarrow Q + Q_v$
13:         $e_{vi} \leftarrow \begin{cases} 1 & \text{if } v = i \\ 0 & \text{otherwise} \end{cases} \quad \forall i \in V_a$
14:         $y' \leftarrow y + e_v$
15:         if $Q' > \max\{\tilde{Q} : (\tilde{Q}, \tilde{y}) \in Y\}$ then
16:             RemoveDominated$((Q', y'), Y')$
17:             $Y$.insert$((Q', y'))$
18:             if $Q' < Q_a^{max}$ then
19:                 Recurse$(Q_a^{max}, y', Y')$
20:         else
21:             $(\tilde{Q}, \tilde{y}) \leftarrow \arg\min_{(\tilde{Q}, \tilde{y}) \in Y} \{\tilde{Q} : \tilde{Q} \geq Q'\}$
22:             $\tilde{C} \leftarrow \sum_{v \in V_a} C_{av} \tilde{y}_v$
23:             if $C' < \tilde{C}$ then
24:                 RemoveDominated$((Q', y'), Y')$
25:                 $Y$.insert$((Q', y'))$
26:                 if $Q' < Q_a^{max}$ then
27:                     Recurse$(Q_a^{max}, y', Y')$
```

```
28: procedure RemoveDominated($(Q, y), Y$)
29:     $C \leftarrow \sum_{v \in V_a} C_{av} y_v$
30:     $(Q', y') \leftarrow \arg\max_{(\tilde{Q}, \tilde{y}) \in Y} \{\tilde{Q} : \tilde{Q} \leq Q\}$
31:     if $(Q', y') = \emptyset$ then
32:         return
33:     $C' \leftarrow \sum_{v \in V_a} C_{av} y'_v$
34:     while $C' > C$ do
35:         $Y$.remove$(Q', y')$
36:         $(Q', y') \leftarrow \arg\max_{(\tilde{Q}, \tilde{y}) \in Y} \{\tilde{Q} : \tilde{Q} \leq Q\}$
37:         if $(Q', y') = \emptyset$ then
38:             return
39:         $C' \leftarrow \sum_{v \in V_a} C_{av} y'_v$
40:     return
```

A.2 Detailed Time Refinement

Algorithm 25 Detailed Time Refinement

```
1: procedure TIMEREFINEMENT
2:     while $InfeasiblePaths \neq \emptyset$ do
3:         $R \leftarrow \emptyset$    ▷ Initialize the set of commodities to repath
4:         $(k, P) \leftarrow InfeasiblePaths$.top()
5:         $InfeasiblePaths \leftarrow InfeasiblePaths \setminus \{(k, P)\}$
6:         $I \leftarrow$ FindVerticesToInsert($k, P$)    ▷ Find vertices to insert to remove mapping error
7:         for $a \in P$ do
8:             $K_a \leftarrow K_a \setminus \{k\}$    ▷ Remove the commodity from the infeasible path
9:         for $(l, t) \in I$ do
10:            $R' \leftarrow$InsertNewNode($l, t$)    ▷ Record which paths are ‘broken’ during vertex insertion
11:            $R \leftarrow R \cup R'$
12:        for $(k', P') \in R$ do    ▷ Remove paths from the network which were ‘broken’ during insertion
13:            for $a \in P'$ do
14:                $K_a \leftarrow K_a \setminus \{k'\}$
15:        $K' \leftarrow \{k' : (k', P') \in R\} \cup \{k\}$    ▷ $K'$ contains commodities whose path was removed
16:        PathCommodities($K'$)
17: function FINDVERTICESTOINSERT(Commodity $k$, Path $P$)
18:     $a^- = ((l_1^-, t_1^-), (l_2^-, t_2^-)) \leftarrow P[0]$
19:     if $t_1^- < e_k$ then    ▷ If commodity leaves origin before its release time
20:         return $\{(l_1^-, e_k)\}$    ▷ Insert the exact release time of the commodity
21:     for $i \in 1, ...|P| - 2$ do    ▷ Iterate over consecutive arcs $a^-, a$ in $P$
22:         $a = ((l_1, t_1), (l_2, t_2)) \leftarrow P[i]$
23:         if $t_1 < t_1^- + \tau_{(l_1^-, l_2^-)}$ then    ▷ If commodity departs on $a$ before $a^-$ arrives in real time
24:             return $\{(l_2^-, t_1^- + \tau_{(l_1^-, l_2^-)})\}$    ▷ Insert the actual arrival time of $a^-$
            $a^- \leftarrow a$
25:     if $t_1^- + \tau_{(l_1^-, l_2^-)} > l_k$ then    ▷ If commodity arrives to destination after its due time
26:         return $\{(l_2^-, t_1^- + \tau_{(l_1^-, l_2^-)}), (l_2^-, l_k)\}$    ▷ Insert the actual arrive of the last arc, and the due time of the commodity
```

```
27: function InsertNewNode(Location $l$, double $t$)
28:     $R \leftarrow \emptyset$     ▷ Initialize set of paths which are ‘broken’ during insertion
29:     $T' \leftarrow T' \cup \{t\}$
30:     $L_{T'} \leftarrow L_{T'} \cup \{(l,t)\}$     ▷ Add the new point as a vertex
31:     $n_1 \leftarrow (l, \max\{t' : (l,t') \in L_{T'}, t' < t\})$     ▷ Find vertex at $l$ immediately preceding $(l,t)$ in time
32:     for $(l,l') \in \delta^+_F(l)$ do     ▷ Iterate over forward star of $l$ in the flat network
33:         $n' \leftarrow (l', \max\{t' : (l',t') \in L_{T'}, t' < t + \tau_{ll'}\}$     ▷ Map arrival of dispatch to a vertex in $L_{T'}$
34:         $A_{T'} \leftarrow A_{T'} \cup \{((l,t),n')\}$     ▷ Add new timed arc to the network
35:     for $a = ((l',t'),n_1) \in \delta^-_{N_{T'}}(n_1)$ do     ▷ Iterate over reverse star of $n_1$ in the time space network
36:         if $t' + \tau_{l'l} \geq t$ then     ▷ If this arc can be optimistically mapped to the new node
37:             $a_{\text{new}} \leftarrow ((l',t'),(l,t))$     ▷ Replace this arc with a new one mapped to the new node
38:             $A_{T'} \leftarrow (A_{T'} \setminus \{a\}) \cup \{a_{\text{new}}\}$
39:             $R \leftarrow R\cup$ ReplaceArc($a, a_{\text{new}}$)
40:     return $R$
```

A.3 Arc Improvement Evaluation Details

Algorithm 26 EvaluateArcImprovement

```
 1: function EvaluateArcImprovement(Arc $a$)
 2:     $R \leftarrow \emptyset$
 3:     for $k \in K_a$ do
 4:         $R \leftarrow R \cup \{k\}$
 5:         $P_k \leftarrow CurrentPathOf(k)$
 6:         for $a' \in P_k$ do
 7:             if $C_{\text{prev}}[a'] = \emptyset$ then
 8:                 $C_{\text{prev}}[a'] \leftarrow a'$.Cost()          ▷ Record cost of arc before improvement
 9:                 $A' \leftarrow A' \cup \{a'\}$
10:             $K_{a'} \leftarrow K_{a'} \setminus \{k\}$
11:             $C_{\text{new}}[a'] \leftarrow a'$.Cost()               ▷ Record cost of arc after removal
12:     for $k \in R$ do
13:         $o \leftarrow (o_k, \max\{t' : (o_k, t') \in L_{T'}, t' \leq e_k\})$
14:         $d \leftarrow (d_k, \min\{t' : (d_k, t') \in L_{T'}, t' \geq l\})$
15:         $P_k^{new} \leftarrow$ShortestPath$(o, d, q_k)$      ▷ Solve shortest $o, d$ path problem on $N$ with costs $\bar{C}_a(q_k)$
16:         for $a' \in P_k^{new}$ do
17:             if $C_{\text{prev}}[a'] = \emptyset$ then
18:                 $C_{\text{prev}}[a'] \leftarrow a'$.Cost()          ▷ Record cost of arc before improvement
19:                 $A' \leftarrow A' \cup \{a'\}$
20:             $K_{a'} \leftarrow K_{a'} \cup \{k\}$
21:             $C_{\text{new}}[a'] \leftarrow a'$.Cost()               ▷ Record cost of arc after improvement
22:     for $k \in R$ do                                          ▷ Restore original state
23:         for $a' \in P_k^{new}$ do $K_{a'} \leftarrow K_{a'} \setminus \{k\}$
24:         for $a' \in P_k$ do $K_{a'} \leftarrow K_{a'} \cup \{k\}$
25:     $\Delta C \leftarrow 0$                                   ▷ Calculate cost change between states
26:     for $a' \in A'$ do
27:         $\Delta C \leftarrow \Delta C - C_{\text{prev}}[a'] + C_{\text{new}}[a']$
28:     return $\Delta C$
```

REFERENCES

[1] Bureau of Transportation Statistics, "Transportation statistics annual report 2021," United States Department of Transportation, Washington, DC, Dec. 31, 2021.

[2] T. G. Crainic, "Service network design in freight transportation," *European journal of operational research*, vol. 122, no. 2, pp. 272–288, 2000.

[3] N. Boland, M. Hewitt, L. Marshall, and M. Savelsbergh, "The continuous-time service network design problem," *Operations Research*, vol. 65, no. 5, pp. 1303–1321, 2017.

[4] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[5] I. Bakir, A. Erera, and M. Savelsbergh, "Motor carrier service network design," in *Network Design with Applications to Transportation and Logistics*, Springer, 2021, pp. 427–467.

[6] M. Chouman, T. G. Crainic, and B. Gendron, "Commodity representations and cut-set-based inequalities for multicommodity capacitated fixed-charge network design," *Transportation Science*, vol. 51, no. 2, pp. 650–667, 2017.

[7] A. Atamturk and O. Gunluk, "Multi-commodity multi-facility network design," *arXiv preprint arXiv:1707.03810*, 2017.

[8] A. P. Armacost, C. Barnhart, and K. A. Ware, "Composite variable formulations for express shipment service network design," *Transportation science*, vol. 36, no. 1, pp. 1–20, 2002.

[9] D. Kim, C. Barnhart, K. Ware, and G. Reinhardt, "Multimodal express package delivery: A service network design application," *Transportation Science*, vol. 33, no. 4, pp. 391–407, 1999.

[10] C. Barnhart, N. Krishnan, D. Kim, and K. Ware, "Network design for express shipment delivery," *Computational Optimization and Applications*, vol. 21, no. 3, pp. 239–262, 2002.

[11] C. Barnhart and R. R. Schneur, "Air network design for express shipment service," *Operations Research*, vol. 44, no. 6, pp. 852–863, 1996.

[12] D. Kim and P. M. Pardalos, "A solution approach to the fixed charge network flow problem using a dynamic slope scaling procedure," *Operations Research Letters*, vol. 24, no. 4, pp. 195–203, 1999.

[13] T. G. Crainic, B. Gendron, and G. Hernu, “A slope scaling/lagrangean perturbation heuristic with long-term memory for multicommodity capacitated fixed-charge network design,” *Journal of Heuristics*, vol. 10, no. 5, pp. 525–545, 2004.

[14] A. I. Jarrah, E. Johnson, and L. C. Neubert, “Large-scale, less-than-truckload service network design,” *Operations Research*, vol. 57, no. 3, pp. 609–625, 2009.

[15] A. Erera, M. Hewitt, M. Savelsbergh, and Y. Zhang, “Improved load plan design through integer programming based local search,” *Transportation Science*, vol. 47, no. 3, pp. 412–427, 2013.

[16] K. Lindsey, A. Erera, and M. Savelsbergh, “Improved integer programming-based neighborhood search for less-than-truckload load plan design,” *Transportation Science*, vol. 50, no. 4, pp. 1360–1379, 2016.

[17] A. L. Erera, M. Hewitt, M. W. Savelsbergh, and Y. Zhang, “Creating schedules and computing operating costs for ltl load plans,” *Computers & Operations Research*, vol. 40, no. 3, pp. 691–702, 2013.

[18] L. Marshall, N. Boland, M. Savelsbergh, and M. Hewitt, “Interval-based dynamic discretization discovery for solving the continuous-time service network design problem,” *Transportation Science*, vol. 55, no. 1, pp. 29–51, 2021.

[19] N. Wieberneit, “Service network design for freight transportation: A review,” *OR spectrum*, vol. 30, no. 1, pp. 77–112, 2008.

[20] W. B. Powell, “Dynamic models of transportation operations,” *Handbooks in Operations Research and management science*, vol. 11, pp. 677–756, 2003.

[21] D. Klabjan, E. L. Johnson, G. L. Nemhauser, E. Gelman, and S. Ramaswamy, “Airline crew scheduling with time windows and plane-count constraints,” *Transportation science*, vol. 36, no. 3, pp. 337–348, 2002.

[22] W. B. Powell, “A local improvement heuristic for the design of less-than-truckload motor carrier networks,” *Transportation Science*, vol. 20, no. 4, pp. 246–257, 1986.

[23] J. M. Farvolden and W. B. Powell, “Subgradient methods for the service network design problem,” *Transportation Science*, vol. 28, no. 3, pp. 256–272, 1994.

[24] N. Katayama, “A combined fast greedy heuristic for the capacitated multicommodity network design problem,” *Journal of the Operational Research Society*, vol. 70, no. 11, pp. 1983–1996, 2019.

[25] I. Ghamlouche, T. G. Crainic, and M. Gendreau, “Cycle-based neighbourhoods for fixed-charge capacitated multicommodity network design,” *Operations research*, vol. 51, no. 4, pp. 655–667, 2003.

[26] ——, “Path relinking, cycle-based neighbourhoods and capacitated multicommodity network design,” *Annals of Operations research*, vol. 131, no. 1, pp. 109–133, 2004.

[27] D. C. Paraskevopoulos, T. Bektaş, T. G. Crainic, and C. N. Potts, “A cycle-based evolutionary algorithm for the fixed-charge capacitated multi-commodity network design problem,” *European Journal of Operational Research*, vol. 253, no. 2, pp. 265–279, 2016.

[28] M. Hewitt, G. L. Nemhauser, and M. W. Savelsbergh, “Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem,” *INFORMS Journal on Computing*, vol. 22, no. 2, pp. 314–325, 2010.

[29] M. Hewitt, “Enhanced dynamic discretization discovery for the continuous time load plan design problem,” *Transportation science*, vol. 53, no. 6, pp. 1731–1750, 2019.

[30] I. Arsik, “Investigations into effectively moving people and goods,” Ph.D. dissertation, Georgia Institute of Technology, 2020.

[31] A. Cohn, S. Root, A. Wang, and D. Mohr, “Integration of the load-matching and routing problem with equipment balancing for small package carriers,” *Transportation Science*, vol. 41, no. 2, pp. 238–252, 2007.

[32] R. M. Karp, “A characterization of the minimum cycle mean in a digraph,” *Discrete mathematics*, vol. 23, no. 3, pp. 309–311, 1978.

[33] A. Schrijver *et al.*, *Combinatorial optimization: polyhedra and efficiency*. Springer, 2003, vol. 24.

[34] T. L. Magnanti and R. T. Wong, “Network design and transportation planning: Models and algorithms,” *Transportation science*, vol. 18, no. 1, pp. 1–55, 1984.

[35] T. G. Crainic, M. Gendreau, and J. M. Farvolden, “A simplex-based tabu search method for capacitated network design,” *INFORMS journal on Computing*, vol. 12, no. 3, pp. 223–236, 2000.

[36] T. Grünert and H.-J. Sebastian, “Planning models for long-haul operations of postal and express shipment companies,” *European Journal of Operational Research*, vol. 122, no. 2, pp. 289–309, 2000.

[37] W. B. Powell and Y. Sheffi, “The load planning problem of motor carriers: Problem description and a proposed solution approach,” *Transportation Research Part A: General*, vol. 17, no. 6, pp. 471–480, 1983.

[38] J. Medina, M. Hewitt, F. Lehuédé, and O. Péton, “Integrating long-haul and local transportation planning: The service network design and routing problem,” *EURO Journal on Transportation and Logistics*, vol. 8, no. 2, pp. 119–145, 2019.

[39] I. Herszterg, Y. Ridouane, N. Boland, A. Erera, and M. Savelsbergh, “Near real-time loadplan adjustments for less-than-truckload carriers,” *European Journal of Operational Research*, vol. 301, no. 3, pp. 1021–1034, 2022.

[40] A. Baubaid, N. Boland, and M. Savelsbergh, “The value of limited flexibility in service network designs,” *Transportation Science*, vol. 55, no. 1, pp. 52–74, 2021.