The Maritime Pickup and Delivery Problem with Cost and Time Window Constraints: System Modeling and A* Based Solution

CHRISTOPHER DAMBAKK

SUPERVISOR
Associate Professor Lei Jiao

University of Agder, 2019
Faculty of Engineering and Science
Department of ICT

Abstract

In the ship chartering business, more and more shipment orders are based on pickup and delivery in an on-demand manner rather than conventional scheduled routines. In this situation, it is necessary to estimate and compare the cost of shipments in order to determine the cheapest one for a certain order. For now, these calculations are based on static, empirical estimates and simplifications, and do not reflect the complexity of the real world. In this thesis, we study the Maritime Pickup and Delivery Problem with Cost and Time Window Constraints. We first formulate the problem mathematically, which is conjectured NP-hard. Thereafter, we propose an A* based prototype which finds the optimal solution with complexity $O(b^{\epsilon d})$. We compare the prototype with a dynamic programming approach and simulation results show that both algorithms find global optimal and that A* finds the solution more efficiently, traversing fewer nodes and edges.

Preface

This thesis concludes the master's education in Communication and Information Technology (ICT), at the University of Agder, Norway.

Several people have supported and contributed to the completion of this project. I want to thank in particular my supervisor, Associate Professor Lei Jiao. He has provided excellent guidance and refreshing perspectives when the tasks ahead were challenging. I would also like to thank Jayson Mackie, co-worker and friend, for proofreading my report.

Grimstad, 24th of May 2019.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Ship chartering companies are regularly faced with the task of determining the price of a shipment which they can offer to a potential customer. These costs are manually calculated based on static estimates that may not reflect the complexity of the real world.

In this thesis, we study the Maritime Pickup and Delivery Problem with Cost and Time Window Constraints (MPDPCTWC). We propose a novel way of deciding a ship and a path from pickup to delivery which minimizes the cost of a shipment. The cost estimation includes operation costs, the price of buying raw materials, and external factors like canal prices, sea conditions, and weather.

To solve the MPDPCTWC problem, we define a model and implement an A* (pronounced A-star) based prototype to combine pathfinding and cost estimation in order to find the optimal combination of ships and routes able to carry out the shipment. The prototype shows promising results and we propose improvements to include more specific domain knowledge.

1.1 Problem Statement

In the world of shipping, dry bulk carriers, for instance *Sabrina 1* in Figure 1.1, are ships that transport dry cargoes such as ores, grain or coal. These vessels are characterized by their hatches on the deck that cover the cargo holds which contain the raw materials. The cargo is transported from a wide spread of locations to ports that hold factories which need the materials.

Figure 1.1: *Sabrina 1*, a typical, so-called *handymax* bulk carrier [1].

Dry bulk shipping is usually done *on-request* by a contractor, similar to how the taxi industry works, in contrast to liner shipping[1]. For example, a factory that needs raw materials can place an order[2] with a shipping company. Then the shipping company has to determine where to buy the materials and with which ship to transport the cargo. The ship must travel from its initial position to the pickup port to load the materials and then deliver it at the destination port.

Klaveness[3] is a company that owns and controls dry bulk carriers. One of their many challenges is to plan which vessel to assign to which shipment in order to minimize costs. This is a complex task, even for experienced planners. Each task includes several unknowns, and the cost estimation can quickly become imprecise. Additionally, this work is mostly done manually, making it time-consuming and prone to errors.

As with other real-world applications, the domain is subject to a set of complex constraints. Some constraints are *firm*, meaning that they

[1] Typically container vessels transporting containers from one port to another, often according to pre-defined schedules, comparable to bus routes.

[2] A purchase agreement for a certain quantity of materials to be delivered. We assume that the shipping company may freely decide where to buy cargo.

[3] `https://klaveness.com/`

can not be violated, and others are *soft*, meaning that they can be broken — but at a cost. The soft constraints are weighted differently, making some of them more important than others. Examples of firm and soft constraints include, but is not limited to:

- **The size of the ship:** A large ship can not travel in the Suez Canal or the Panama Canal. Larger ships are, however, able to carry more load.
- **Time:** There are several time constraints: A ship must be at the delivery port during certain service time windows. If a ship fails to meet the time window, it might not be served. Also, the ship needs to be at the destination within another time window.
- **Contracts:** Some ships only travel between certain ports, countries or continents according to constraints defined in a contract.
- **Pricing:** The price of raw materials may vary between different ports and this must be taken into consideration when selecting where to buy cargo.
- **Operation costs:** The price of operating a ship, including staff, fuel, insurance, etc. A loaded ship is more expensive to operate as the fuel costs are higher.
- **Sea conditions:** Some paths at sea are more favorable than others due to external factors such as underwater streams or icy seas.
- **Weather:** Weather conditions such as wind and storms affect the route a ship selects and therefore the cost.
- **Ports and canals:** Certain ports, canals, and other areas charge an additional cost.

The result of planning should be a route that transports materials from a loading port to a destination with the lowest possible costs, obeying all the firm constraints and finding the best compromise of

the soft constraints. By providing a software platform capable of such planning, ship chartering companies can make better decisions to lower costs.

The research on pathfinding in the context of shipping is in its early stages. Previous work such as the Petrobras Challenge [2], [3], and [4], and other planning challenges [5] mostly focus on optimizing routing and planning of liner shipping systems and are comparable to other well-researched vehicle routing problems. The doctoral dissertation by Dolinskaya [6] proposes an algorithm capable of finding a ship's fastest route based on real-time sensor data, but only on a local scale. Beeker [7] presents an algorithm able to find the shortest path avoiding the limitations of the arcs in a network by using Delaunay triangulation. However, this work also focuses on finding the shortest path and does not include costs and other constraints. Our thesis is taking a well-known algorithm for solving problems similar to the Traveling Salesman Problem and applies it to the open field of global ship pickup and delivery optimization.

1.1.1 Thesis Goals

To solve the Maritime Pickup and Delivery Problem With Cost and Time Window Constraints, we outline the following goals in our study:

Goal 1: Formulate the problem mathematically and evaluate the complexity.

Goal 2: Design and implement an algorithm capable of finding optimal routes from a starting point, through a pickup port, and to its final destination for the given constraints.

1.2 Contributions

This thesis introduces a solution to the MPDPCTWC problem by applying a known technique for pathfinding to bulk carrier shipping.

The contributions of this thesis include:

1. An implementation of the A* search algorithm with cost and time window constraints applied to the context of dry bulk pickup and delivery. The algorithm can find the optimal ship and route for the current set of constraints.
2. A framework for route planning that can be extended to include a vast variety of costs and constraints to account for real world conditions.

1.3 Thesis Structure

This thesis is structured as follows. Chapter 2 presents the current research on pathfinding and route optimization, both in general and in the maritime domain, before formulating and modeling the pickup and delivery problem in Chapter 3. The implementation of the algorithm to solve the problem is covered in Chapter 4 and the results are presented in Chapter 5. Finally, we conclude the thesis and propose future work in Chapter 6.

Chapter 2

Background

This chapter reviews the current research on pathfinding before looking into the research on maritime use-cases similar to our problem. We cover theory regarding different pathfinding techniques and discuss their features, including their strengths and weaknesses. Then, we give a technical description of the concepts and algorithms discussed in this thesis.

2.1 Literature Review

Pathfinding is a thoroughly studied subject in computer science. In essence, pathfinding is similar to the *shortest path problem* defined in graph theory, which aims to find the best path (shortest, cheapest, or some other criteria) between two nodes in a graph. The use-cases for pathfinding include, but are not limited to, maze solving [8, 9], game development [10, 11], robotics [12, 13, 14], and logistics [2, 3, 4, 5].

In the survey of path planning algorithms conducted by Souissi et al. in 2013 [15], they present a wide range of classic pathfinding algorithms and variations developed in recent decades and suitable applications and scenarios. Amongst these methods are Dijkstra and A*, Rapidly-exploring Random Trees, and Potential Fields, which will

be explained throughout this section.

The A* pathfinding algorithm origins back to 1968 when Hart, Nilsson, and Raphael [16] extended Dijkstra's algorithm [17] to include a heuristic to reduce the number of nodes to be visited. This improvement sped up the search process while still guaranteeing the most optimal solution. Today, the original implementation is still highly popular and regarded as the state-of-the-art within pathfinding. However, some improvements have been proposed to deal with specific use-cases. One such improvement is Lifelong Planning A* (LPA*) [18, 19] which caches the intermediate results from previous searches to speed up future searches starting from the same point in the graph. The cached values must, however, be recalculated each time the starting point changes. Another variant is Hierarchical Pathfinding A* (HPA*), proposed by Botea et al. [20]. This approach abstracts the graph into a more coarse-grained, clustered graph to reduce the complexity and search effort of pathfinding between nodes that are far apart. Botea et al. compare HPA* as traveling on a motorway between cities before entering the city roads. This abstraction does, however, mean that the algorithm produces sub-optimal solutions. They reported that their implementation of HPA* was up to 10 times faster than conventional A* and within 1% of the optimal path.

LaValle introduced Rapidly-exploring Search Trees (RRTs) [21] with the aim to quickly explore unsearched regions. Amongst several use-cases, RRTs have shown to be applicable to pathfinding. Inspired by other randomized path planning techniques such as Randomized Potential Field [22] and Probabilistic Roadmaps [23], RTTs are characterized as a tree expansion search which is biased towards unexplored areas. Čáp et al. [21, 24] applied RRT for a co-operative pathfinding problem to co-ordinate multiple agents (such as autonomous aircrafts or cars) to find the shortest path for all agents while at the same time avoiding collisions. They concluded that their implementation of RRT, MR-RRT*, was more efficient than other similar approaches on large environments, but produces sub-optimal routes.

A third family of pathfinding algorithms are Potential Fields, introduced by Khatib [25]. He presented an approach for robot obstacle

avoidance in real-time by coupling environment sensing with low-level robot control. The concept of Potential Fields can be described as the sum of attractiveness to the global goal and repulsiveness from obstacles at any position. Potential Fields are prone to local minima, as described by Mabrouk and McInnes [26] who also suggests possible countermeasures.

A recent paper by Mirjalili et al. demonstrates how Ant Colony Optimization (ACO) can be used for pathfinding of an Autonomous Underwater Vehicle (AUV) and other domains similar to the Traveling Salesman Problem [14]. ACO is a well-known technique in the field of swarm intelligence to explore the environment and find the shortest path — similar to how real-world ants behave to find food sources. Their ACO implementation was tested in case studies with up to 200 way-points and the ACO algorithm quickly converged and found the shortest path covering all points. However, as with other non-deterministic swarm algorithms, the convergence time of ACO is uncertain.

In game development, pathfinding is widely used to make NPCs[1] appear intelligent by moving in meaningful patterns and directions. Millington's book, Artificial Intelligence for Games [27], presents Dijkstra and A* as the two major go-to algorithms for performing pathfinding in games. Both algorithms are able to work with models of the game world and also have the performance and memory footprint required for real-time games. Additionally, A* can be extended to take different types of factors into account, such as different capabilities of the NPCs, current state of the game, other NPCs or players nearby, or a combination of these. Redfern improved his pathfinding algorithm in his game based on player-tracked movements [10]. He modeled the player behavior as heatmaps and used a heuristic function to guide the entities in *good* directions based on the heatmap of previous behaviours. This approach is interesting as it shows that the pathfinding algorithm can take a wide variety of factors and constraints into account, not only distance or travel time. Artificial Intelligence for Games [27] also discusses a similar feature using influence maps. Their use-cases include finding the best path for different military units with certain characteristics in a particular

[1] Non-Playable Character, an element controlled by the computer

terrain.

2.1.1 Research on Maritime Pickup and Delivery, Pathfinding, and Optimization

The research in the maratime domain has mostly focused on route optimization and organizing a fleet of ships to efficiently deliver goods across a set of destinations and minimizing repositioning[2]. This kind of optimization goals are common for the traffic pattern known as *Liner shipping*. In this pattern the available ports and ships are limited, the ships travel according to schedules, and the problem scale is local, similar to bus routes in a city. Another shipping pattern, called *Tramp shipping*, let ships transport cargo on-request, much like a taxi service. Finally, we have *Industrial shipping*, where shipping companies control their own fleet of ships [28]. Our pickup and delivery problem falls in between the latter two patterns.

A common constraint in the shipping domain is service time windows. A port is often subject to heavy traffic, and to cope with this, ships are assigned certain time windows in which service must start. This constraint is usually considered as a hard constraint in the literature, but can in fact often be violated for an inconvenience cost. Fagerholt [29] suggested an optimization algorithm that takes the inconvenience cost into account to find the most optimal schedule. He modeled the Traveling Salesman Problem with Capacity, Soft Time Windows, and Precedence Constraints (TSP-CSTWPC) problem as sequences of nodes to represent schedules. Candidate schedules were generated to find the most optimal one, applying different inconvenience cost functions. His approach found the most optimal solution in three out of four experiments.

For the case of Tramp shipping, Fagerholt and collegues [30] proposed a model to allow a fleet of ships to co-operate on a cargo delivery by splitting loads while still obeying the time windows of the involved ports. Their results showed that splitting loads can significantly improve schedule quality as well as increasing a company's

[2]Moving an empty container or ship.

profit from the reduced sailing costs and the opportunity to bring additional cargo that is not part of the original shipment contract.

A well-known problem in the shipping planning domain is the Petrobras challenge [2], [3], [4]. This challenge was proposed at the International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS) [31] to motivate research innovations to be applied to real-world problems. The objective of the Petrobras challenge is to optimize the fuel cost, the number of ships, and the waiting time when distributing cargo across a set of petroleum platforms, without breaking any of the ship, platform, or port capacity constraints. The Petrobras challenge has a lot in common with both planning and scheduling problems, and can be approached in different manners.

The first attempt to solve the Petrobras Challenge was submitted by Toropila et al. [2], who proposed three solutions. Two of their solutions tackled the problem as a planning problem, using the industry standard planning modelling language PDDL [32] to represent the domain and SGPlan [33] to compute the plans. They also implemented a third approach, a Monte Carlo Tree Search (MCTS) version that resulted in, at that time, state-of-the-art results. Then, in 2013, Barták and collegues [3] changed the PDDL model by describing the sequences of actions as a Finite State Automaton. This could in turn be used to guide the planner to only explore "proper" paths and thus improve efficiency. Lastly, in 2014, Barták et al. [4] aimed to improve the efficiency even further by utilizing the tabling feature of the Picat language. They showed that they are able to outperform the MCTS version described earlier.

The contributions regarding maritime pathfinding is limited but covered by the doctoral dissertation by Dolinskaya [6]. Her research aims to develop a system that can, based on real-time measurements and forecasts, control a vessel and find an optimal path to the destination in a direction, location, and time dependent environment, while minimizing travel time, fuel consumption and obeying motion restrictions (such as turn sharpness). Using real-time vessel sensors and surrounding environment data, such as forcasted wave-field, she developed a pathfinding algorithm to determine the most favorable

path. She argues that the pathfinding algorithm must be independent of the speed function as it differs for each vessel and that an analytical function cannot accurately describe the vessel movement.

Although the main goal of finding the optimal path at sea and minimizing cost is the same as ours, Dolinskaya and her collegues focuses on finding the local best path within a radius of 10 minutes of travel time, while our work has a global perspective. Because of the relatively short travel time range, her use-case is real-time navigation, whereas we aim to estimate the cost of future operations and compare possible shipments in the planning phase. She also uses ship sensor data and information about the surrounding environment as input to the pathfinder, unlike our model which is based on the whole world, weather forecasts and global sea conditions such as underwater streams.

In contrast to the traditional graph model required when performing pathfinding, Beeker [7] suggests to use Delaunay triangulation to yield the actual shortest path from any position to any port. He argues that a major weakness of current pathfinding algorithms is their tight dependency to a graph representation of the world. If a ship is not at a node's location when the simulation begins or the destination changes mid travel, the pathfinder requires the ship to find the closest node before starting the search — with the possibility of the generated path deviating considerably from the actual shortest path. Although his work presents a novel and promising approach of finding the shortest path at sea, he does not include cost or time window constraints.

2.2 Technical Background

This section aims to give the reader a basic introduction of how pathfinding algorithms, and A* in particular, works and give an understanding of the terms used when discussing pathfinding.

2.2.1 Graph Data Structure

In computer science, the graph data structure, closely related to mathematical graphs, is a way to model the relation between objects in the real world as vertices (also called nodes) and edges. The applications of graphs ranges from the study of molecules and atoms to database design and resource allocation and scheduling, and is considered a fundamental concept. [34]

Pathfinding algorithms in computer applications often require the world to be modeled as a graph. In essence, the graph is a simplification of the real world where edges between nodes represent possible paths. If the graph is suitably abstracted, it can be used by pathfinding algorithms to find paths matching any constraints and, in turn, be applied to real-world use-cases.

Each edge in a graph used for pathfinding, is assigned a cost. This cost may represent distance, time required to travel the edge, or some other metric. This measurement is often non-negative and we say that the graph is weighted. Also, for the case of pathfinding, the graphs are often directed meaning, that edges can only be traveled in one direction. However, two nodes may have two parallel but opposite directed edges between them allowing non-symmetrical costs.

2.2.2 Pathfinding and A*

The problem of finding the shortest, cheapest or preferred path in an environment is a task that computer science has studied for a long time, and has gained increased importance and focus with the rise of use-cases such as robots, UAVs and more. Pathfinding algorithms, and A* in particular, have long been used in game development to make computer controlled elements behave intelligently. In this section, we will explain the workings of pathfinding algorithms such as A*.

The A* pathfinding algorithm is an extension of the famous Dijkstra algorithm invented by the Dutch computer scientist Edsger W. Di-

jkstra [17]. A* and Dijkstra are graph traversing algorithms to find the shortest path between two nodes in a graph. By modeling the world as a graph and assign cost to the edges, they can also be used for pathfinding. Both algorithms are deterministic and will generate a complete, guaranteed optimal path in finite time. [35]

Both Dijkstra and A* work around two main lists; the *open* list and the *closed* list. The *open* list is a priority queue containing all nodes in the graph to be processed, ordered by their correct, accumulated cost, and that the algorithm considers to be part of the final path. The *closed* list contains all nodes that has been visited and processed by the algorithm. At each iteration, the pathfinder selects the cheapest (most promising path to the goal) node from the *open* list and evaluates the outgoing edges from this node. All the neighboring nodes are added to the *open* list and the current node is moved to the *closed* list. This process continues until the destination node is reached. As both algorithms always selects the most promising node at each step, the first time they reach the goal node will be the optimal path. [27, 35]

The difference between the two algorithms lies in how the A* guides its search towards promising nodes due to the introduction of heuristics, as explained in the following paragraphs.

Heuristics

The main difference between the Dijkstra and A* algorithm is the evaluation of the next nodes when stepping through the graph. A* uses a heuristic function $h(n)$ to assign a value that indicates how likely the node is to be a part of the globally best path. This value should be as close to the final cost as possible if continuing the current path. When selecting the next node to process from the *open* list, the cheapest node is the node with the lowest sum of the cost so far and the heuristic value. Consequently, A* becomes biased towards exploring nodes that are more likely to be part of the final path. This concept is exemplified in Figure 2.1. [35]

It is important that the heuristic does not overestimate the actual

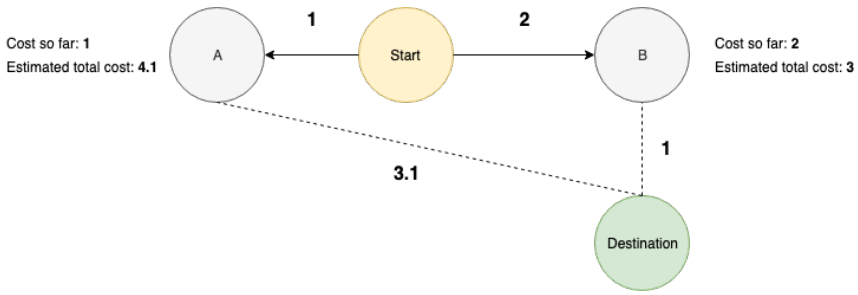


Figure 2.1: Example where a heuristic function is used to optimize the search process. The dotted lines represent the Euclidean distance. When using Dijkstra, node A will be processed before node B because of the lower *cost so far*. On the other hand, A* will process node B first because it has a lower *Estimated total cost*, and thus finds the optimal path to the destination more quickly.

cost of the final path. If so, the pathfinder will generate suboptimal solutions as the selection from the *open* list will be wrong and lead the pathfinder the wrong way. A common heuristic is the Euclidean distance from the current node to the destination. Even if the path to the goal is the same as the Euclidean distance, the heuristic will not be greater than the final cost. We say that the heuristic is *admissible* [36].

An admissible heuristic is required when using A* for tree-search. When searching a graph the requirement is slightly more strict. We say that the heuristic function must be *consistent* [36]. A heuristic $h(n)$ is consistent, and thus also admissible, if the estimated cost from node n to the goal is lower than the sum of the cost from node n to its successor n' and the the estimated cost from node n' to the goal node, as shown by the following inequality:

$$h(n) \leq c(n, n') + h(n').$$

By making the heuristic match the final cost as closely as possible, it can make A* even more efficient by reducing the number of nodes processed, as they remain further down the *open* list and are never explored. If the heuristic value at a node is the same as the actual cost to the goal, as in Figure 2.1, we say that the algorithm is perfectly informed at that node [37].

Chapter 3

Problem Formulation

This chapter elaborates on the problem to be solved and explains the intricacy before defining the mathematical model necessary to describe the problem. We present the graph notations, the objective function, and the constraints.

3.1 Description of the Problem

This project aims to help ship chartering companies assign a vessel to a shipment. Imagine a shipping company receives an order to deliver a certain amount of coal to a factory. If the shipping company accepts the order, they must decide where to buy the coal, with which ship to transport the cargo, and which route the ship should travel while minimizing the total cost.

The destination port is fixed for each shipment order. This is the port where the cargo must be delivered and is out of the shipping company's control. However, the shipping company may decide where to buy cargo. Different ports may offer the requested cargo to various prices due to supply and demand. It might be worth buying more expensive cargo to avoid bad weather conditions or travel a longer route to buy cheaper cargo.

Due to traffic congestion at busy ports, each vessel may be assigned a so-called service time window in which the ship must arrive to be serviced [29, 30]. We assume these time windows are firm, meaning that a ship arriving outside the given time window will not be serviced, and in turn result in the ship not being able to carry out the shipment according to the requirements. Service time windows apply to both pickup ports and the destination port.

The shipping company may also have multiple vessels spread across the world that can carry out the shipment. Some vessels have a more favorable start location than others, with regards to distance to the pickup ports, or sea and weather conditions between the current position and the destination.

Today, shipping companies manually compare potential ships for an order based on well-established list of prices and estimated operation costs. These are static and inaccurate as they include estimated costs of every possible condition (sea, weather, waiting times), which may provide too many, vague or conflicting options.

We aim to make better estimates that can more accurately reflect the final cost of a shipment by combining more up-to-date price forecasts, more accurate travel details and operation costs, including weather and sea conditions, in the evaluation process. This estimation can be applied to every combination of vessel and pickup port to find the one with the lowest overall cost.

To summarize, the goal is to find the most optimal compromise of all the external factors and those that can be decided by the shipping company. These factors include which ship to select for the shipment, which port to buy cargo, which route to travel from the vessel's initial position to the pickup port, and which route to travel from the pickup port to the destination port.

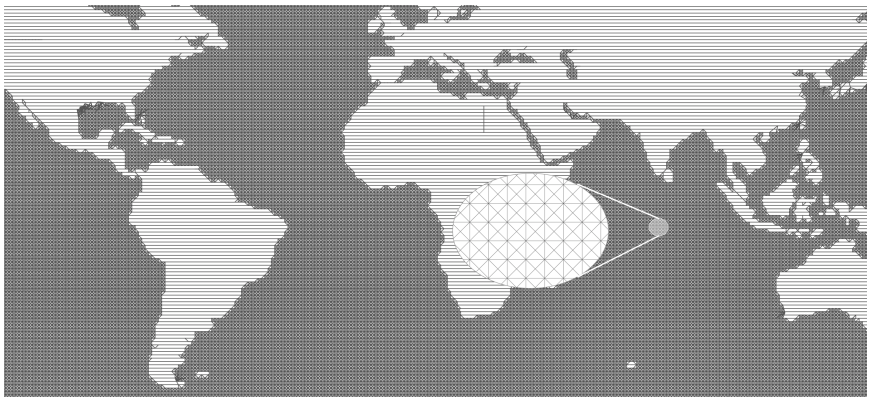


Figure 3.1: An example of a graph to represent the world.

3.2 The Graph Model and Notations

Figure 3.1 shows an example graph used to represent the world. The mathematical formulation for this kind of graph is as follows: given l number of aggregated locations and n number of ports, we have in total $m = l + n$ vertices (also known as nodes). The vertices are connected by edges, which are ordered pairs represented in a $m \times m$ matrix $\mathbf{X}$. The binary element x_{ij} within $\mathbf{X}$ equals to 1 if there is an edge from i to j, and 0 otherwise, as shown in Eq. (3.1).

$$x_{ij} = \begin{cases} 1 & \text{if there is an edge from } i \text{ to } j \\ 0 & \text{otherwise.} \end{cases}, \tag{3.1}$$

With matrix $\mathbf{X}$ we can represent the vertices and the edges as a directed, weighted graph $G(V, E)$, where V is the set of vertices and E is the set of edges. A vertex $v_t \in V$ where $t \in [1, 2, \ldots, m]$ is a geographical location at sea or land (such as a port), defined by latitude and longitude. An edge $e_{a,b} \in E$ where $v_a \in V$, $v_b \in V$, $a \neq b$ is a connection from vertex v_a to vertex v_b and the cost of traveling along that edge is $\omega_{a,b}$.

3.3 Problem Definition and Mathematical Formulation

To describe the problem in a formal manner, we introduce the following notations: let $S = \{s_1, s_2, \ldots, s_o\}$ be the set of ships and s_i be the ith ship belonging to S and o is the total number of ships. All ships have a max load capacity, denoted $s_{i_{dwt}}$ [1] and an initial position denoted as the vertex $v_{s_i,0}$.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of ports and p_j be the jth port in P. Let $P^{\eta,s_i} \subset P$ be the subset of ports where ship s_i may buy and pick up cargo and let $p_{a,s_i} \in P^{\eta,s_i}$ be the ath pickup port for ship s_i. A ship s_i in S may load cargo from any of the ports in P^{η,s_i}. Let $P^\eta = \cup_{\forall s_i} P^{\eta,s_i}$ and $V^\eta = \cup_{\forall s_i} v_{s_i,0}$.

Define $p_d \in P \backslash P^\eta \backslash V^\eta$ as the final destination port where the cargo is to be delivered. p_d is not any of the pickup ports in P^η nor any of the source positions in V^η and is fixed for all ships in S, i.e., there is one destination for all ships in the current shipment. We assume that all pickup ports in P^η have enough raw materials to cover the need of a purchase. The number of tonnes of raw materials to buy is denoted τ and the price of raw materials pr tonnes in port $p_i \in P^\eta$ is γ_{p_i}.

Define $PSA_{s_i,p_{a,s_i}}$ as the set of all paths that ship s_i may travel from $v_{s_i,0}$ to p_{a,s_i}, and one particular path in $PSA_{s_i,p_{a,s_i}}$ is denoted by $psa_{s_i,p_{a,s_i},k}$, where k is the index of the paths. Let $\{e_j\}(psa_{s_i,p_{a,s_i},k})$ be the set of edges in $psa_{s_i,p_{a,s_i},k}$, indexed by j.

Similarly, let $PAD_{s_i,p_{a,s_i}}$ be the set of paths that ship s_i may travel from p_{a,s_i} to p_d, and one particular path in $PAD_{s_i,p_{a,s_i}}$ is denoted by $pad_{s_i,p_{a,s_i},k}$, where k is the index of the paths. Let $\{e_j\}(pad_{s_i,p_{a,s_i},k})$ be the set of edges in $pad_{s_i,p_{a,s_i},k}$, indexed by j. Let $\mu_{s_i,\alpha}$ and $\mu_{s_i,\beta}$ be the factor for operating cost of an empty and a loaded ship s_i, respectively.

[1] Deadweight Tonnage — a metric of how much cargo a ship may carry in terms of weight.

The set of cost functions, C_{s_i}, are applied to each ship s_i, and W_{s_i} is the set of the corresponding weights, both indexed by $k \in \{1, 2, 3, \ldots, |C_{s_i}|\}$ as $c_{s_i,k}$ and $w_{s_i,k}$, where $|C_{s_i}|$ and $|W_{s_i}|$ are the cardinality of C_{s_i} and W_{s_i}, respectively, and clearly $|C_{s_i}| = |W_{s_i}|$ holds.

The arrival time of ship s_i in port $p_j \in P^{\eta,s_i} \cup p_d$ is denoted t_{s_i,p_j}. Also, all ports may have a certain service time window for a certain ship. The service time window for ship s_i in port p_j is $[a_{s_i,p_j}, b_{s_i,p_j}]$ for all $s_i \in S$ and all $p_j \in P^{\eta,s_i} \cup p_d$. This also applies to the destination port p_d. We denote the actual departure time for ship s_i from its initial position as $t_{v_{s_i},0}$. We also assume that ships travel at the same, constant, economic speed [38].

The notations are summarized in Table 3.1.

An objective function to minimize the overall cost for a certain shipment, can be formulated as

$$\begin{aligned} \min_{\{s_i,\ p_{a,s_i},psa_{s_i,p_{a,s_i},k},\ pad_{s_i,p_{a,s_i},k}\}} \Bigg(& \tau\gamma_{p_a} + \sum_{e_q \in \{e_j\}(psa_{s_i,p_{a,s_i},k})} f(e_q, s_i)\ \mu_{s_i,\alpha} \\ & + \sum_{e_q \in \{e_j\}(pad_{s_i,p_{a,s_i},k})} f(e_q, s_i)\ \mu_{s_i,\beta} \Bigg), \\ & \forall s_i \in S,\ p_{a,s_i} \in P^{\eta,s_i}, \\ & psa_{s_i,p_{a,s_i},k} \in PSA_{s_i,p_{a,s_i}},\ pad_{s_i,p_{a,s_i},k} \in PAD_{s_i,p_{a,s_i}}, \end{aligned} \tag{3.2}$$

where

$$f(e_q, s_i) = \sum_{k \in \{1,2,3,\ldots,|C|\}} c_{s_i,k} w_{s_i,k},\ \textit{for edge } e_q \textit{ and ship } s_i. \tag{3.3}$$

subject to

$$|C_{s_i}| = |W_{s_i}|\,, \quad \forall\, s_i \in S, \tag{3.4}$$

$$w_{s_i,j} \in [0,1], \quad \forall\, w_{s_i,j} \in W_{s_i},\ s_i \in S, \tag{3.5}$$

$$a_{s_i,p_j} \leq t_{s_i,p_j} \leq b_{s_i,p_j}, \quad \forall\, s_i \in S,\ p_j \in P, \tag{3.6}$$

$$t_{s_i,0} \leq t_{s_i,p_j} < t_{s_i,p_d}, \quad \forall\, s_i \in S,\ p_j \in P^{\eta,s_i},\ p_d \in P \backslash P^{\eta} \backslash V^{\eta}, \tag{3.7}$$

$$0 < \tau \leq s_{i_{dwt}}, \quad \forall\, s_i \in S, \tag{3.8}$$

$$0 \leq f(e_q, s_i), \quad \forall\, e_q \in psa_{s_i,p_{a,s_i},k} \cup pad_{s_i,p_{a,s_i},k}, \tag{3.9}$$
$$\forall\, s_i \in S,\ \forall\, p_{a,s_i} \in P^{\eta,s_i},$$
$$\forall\, psa_{s_i,p_{a,s_i},k} \in PSA_{s_i,p_{a,s_i}}$$
$$\forall\, pad_{s_i,p_{a,s_i},k} \in PAD_{s_i,p_{a,s_i}},$$

$$\mu_{s_i,\alpha} \leq \mu_{s_i,\beta}, \quad \forall\, s_i \in S. \tag{3.10}$$

The objective function, Eq. (3.2), minimizes the cost of purchasing and transporting cargo. The expression consists of three addends: the price of raw materials, the cost of traveling to the pickup port, and lastly traveling from the pickup port to the final destination. The parameters that can be tuned to minimize the function are the ship to transport cargo, the port to pick up cargo, the path from the vessel's initial position to the pickup port, and the path from the pickup port to the destination port. The cost of traveling an edge in a path is the sum of a set of cost functions and corresponding weights, represented in Eq. (3.3).

The objective function is subject to constraints, which are as follows. Eq. (3.4) and Eq. (3.5) shows that the number of cost functions and weights is the same, and that the value of each weight is in the inclusive range from 0 to 1. Eq. (3.6) defines that a ship s_i may only be served at port p_j during the port's service time window for ship s_i. This constraint applies to both pickup ports and destination ports as P contains all ports.

Eq. (3.7) states that ship s_i must pick up cargo before it can deliver it, making it impossible to pick up cargo in the destination port. However, ship s_i can pick up cargo in its starting position.

Ship s_i can not load more cargo than the ship's capacity, according

#	Description
$G(V, E)$	Graph with verticies V and edges E
v_t	Vertex in V
$e_{a,b}$	An edge from vertex v_a to vertex v_b
S	Set of ships
s_i	Ship belonging to S
$s_{i_{dwt}}$	Ship s_i's Deadweight Tonnage
$v_{s_i,0}$	Initial vertex/position of ship s_i
P	Set of ports
p_j	Port belonging to P
P^{η,s_i}	Subset of ports that have available raw materials for ship s_i
p_{a,s_i}	Port in P^{η,s_i}
V^η	$V^\eta = \cup_{\forall s_i} v_{s_i,0}$. The set of initial positions of ships
P^η	$P^\eta = \cup_{\forall s_i} P^{\eta,s_i}$. Subset of ports that have available raw materials for ship s_i.
p_d	Destination port, in $P \backslash P^\eta \backslash V^\eta$
τ	Amount (in tonnes) of raw materials to buy
γ_{p_i}	Price of raw materials in port p_i
$PSA_{s_i,p_{a,s_i}}$	Set of paths that a ship s_i may travel from $v_{s_i,0}$ to p_{a,s_i}
$PAD_{s_i,p_{a,s_i}}$	Set of paths that a ship s_i may travel from p_{a,s_i} to p_d
$psa_{s_i,p_{a,s_i},k}$	Path in $PSA_{s_i,p_{a,s_i}}$
$pad_{s_i,p_{a,s_i},k}$	Path in $PAD_{s_i,p_{a,s_i}}$
$\mu_{s_i,\alpha}$	Operating cost of empty ship s_i
$\mu_{s_i,\beta}$	Operating cost of loaded ship s_i
C_{s_i}	Set of cost functions applicable to ship s_i
W_{s_i}	Set of weights corresponding to the cost functions in C_{s_i}
$c_{s_i,k}$	Cost function in C_{s_i}
$w_{s_i,k}$	Weight in W_{s_i}
$\lvert C_{s_i} \rvert$	Cardinality of C_{s_i}
$\lvert W_{s_i} \rvert$	Cardinality of W_{s_i}
t_{s_i,p_j}	Arrival time of ship s_i in port p_j
$[a_{s_i,p_j}, b_{s_i,p_j}]$	Service time window for ship s_i in port p_j
$t_{v_{s_i,0}}$	Initial departure time of s_i from vertex $v_{s_i,0}$

Table 3.1: Summary of the mathematical notations

to Eq. (3.8). Finally, Eq. (3.9) states that the cost for ship s_i of traveling edge e_k has a non-negative cost and Eq. (3.10) states that it is more expensive to operate a loaded ship than an empty ship.

3.4 Complexity

The Pickup and Delivery Problem is in general NP-hard [39, 40, 41]. For our Maritime Pickup and Delivery Problem with Cost and Time Window Constraints, we conjuncture that the hardness of the problem is also NP-hard when comparing to the hardness of similar problems [5, 42, 43].

Chapter 4

Proposed Solution

This chapter describes the implementation of the prototype in detail, starting with the graph and continuing with the pathfinding algorithm. We cover the data structures used and present pseudo code for both the proposed algorithm and the dynamic programming (DP) approach used for verification. We also evaluate the complexity of both algorithms. The results from the prototype will be presented in Chapter 5.

4.1 Graph Implementation

As discussed in Section 3.2, the world is modeled as a graph data structure consisting of vertices and edges. In the prototype, we implement a graph with a vertex on every combination of latitude and longitude. The vertices on land are removed, and the remaining vertices are connected to surrounding vertices by edges. Then the ports are added as verticies and connected to nearby vertices by edges. A part of the graph is visualized in Figure 4.1. All vertices are connected by two parallel but opposite directed edges making it possible to assign different cost between a pair of vertices based on direction. Figure 4.2 shows how a vertex is connected to nearby vertices. The cost of an edge is calculated at runtime as it depends on the current

state of the pathfinder, the constraints, and the current simulated time.

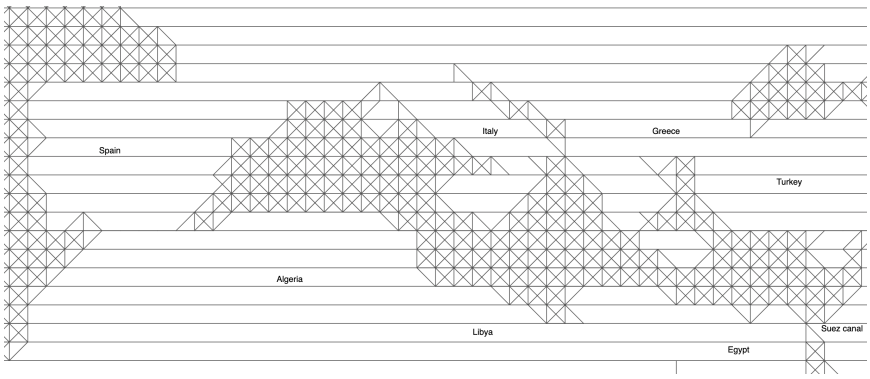


Figure 4.1: A visualization of the graph representing the seas, here a view of the Mediterranean Sea. The edge south of Spain connects the Atlantic and the Mediterranean Sea. Also, notice the Suez Canal in the bottom right corner of the figure.

Each edge is assigned a distance value which is the Great-circle distance[1] between two locations on the earth's surface. Because the earth is a globe, the vertices at longitute -180 are connected with their counterpart vertex at longitute $+180$ and vice versa by edges with a distance of 0 km. In Figure 4.1, these edges are shown as the horizontal lines crossing land. Connecting the vertices in this manner makes the graph an infinite, directed, weighted graph.

The latitudes and longitudes ranges from -90 to 90 and -180 to 180, respectively, and results in a total of $180 \cdot 360 = 64800$ vertices. Further, when removing the vertices on land, including the ports, and making the edges directed, we end up with 31641 vertices connected by a total of 244078 edges. The vertices have an average branching factor of $b = \frac{244078}{31641} \approx 7.7$

[1]The shortest distance between two locations on a sphere [44].

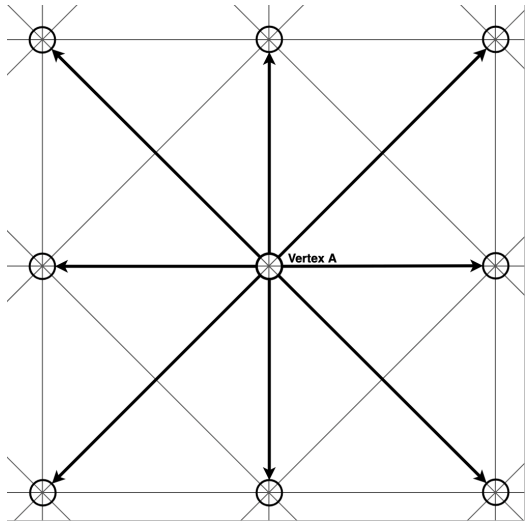


Figure 4.2: Vertices that *Vertex A* is connected to. The circles represent vertices in the graph and the arrows represent directed edges.

4.2 GeoJSON

To represent and describe geospatial data the prototype uses the GeoJSON format, defined in RFC 7946 [45]. This JSON based format allows us to visualize paths, areas, and points of interest and can be used to input information about specific areas to the pathfinder. The results from the pathfinding algorithm can be exported and viewed in a GeoJSON enabled tool. Note that the visualization of such GeoJSON models is dependent on the selected tool and is out of scope for this project. Other formats to represent geospatial data are also available.

In our prototype, GeoJSON is used to assign cost functions to certain areas; for example, setting the cost of traveling through the Suez Canal. As both the graph and the cost functions are based on geospatial data, the pathfinding algorithm can reason about the cost on each step. Figure 4.3 shows a visual representation of the area around the Suez Canal, which is loaded into the prototype and assigned a cost. Every time the pathfinder travels through this area it adds the additional cost to the current path.

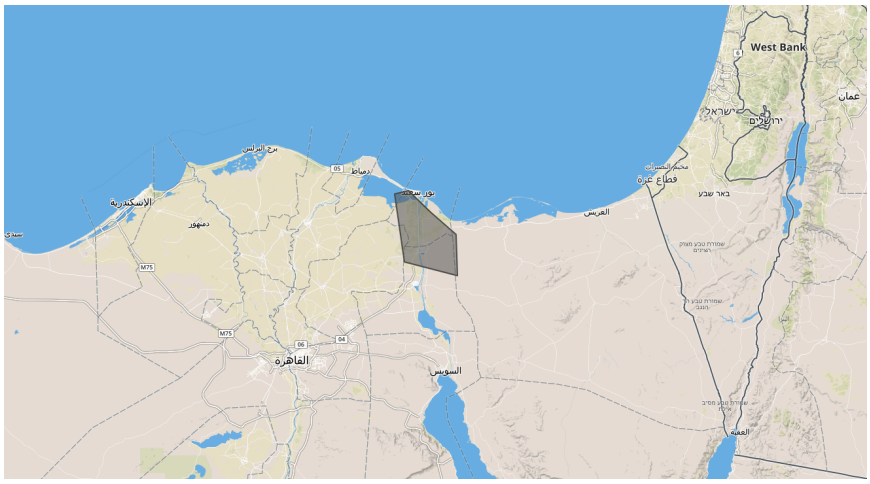


Figure 4.3: One use of GeoJSON to define areas with a certain cost, for example the Suez Canal.

4.3 Prototype Implementation

Here, we describe the implementation and provide pseudo code for the algorithms. The source code for both approaches and for generating the graph is available on GitHub[2]. Although the presented pseudo code is sequential, the actual implementation uses parallelization to speed up the search.

4.3.1 A* Implementation

As explained in Section 3.3, the task of finding the optimal shipment is twofold. First, we need to find the optimal path from the starting position to the pickup port, and then from the pickup port to the destination port. We perform pathfinding using A* on both these intermediate routes and sum the costs. We do the same for all combinations of ships and pickup ports and select the combination with the lowest cost. The search for an optimal combination of ships and pickup ports is described as pseudo code in Algorithm 4.1.

[2] `https://github.com/Dambakk/ShippingRouter`

In Algorithm 4.1, line 2 initializes an empty list to contain the results. Line 3 and 4 iterates the available ships and pickup ports, respectively. The actual pathfinding is done on line 5 and 6, first from the starting position to the pickup port and then from the pickup port to the destination. The result is stored as a pair of path and cost in the list from line 2. The function returns the path with the lowest cost from line 10.

Searching every possible combination of ship and pickup port, as described in Algorithm 4.1, is affordable as the number of ships and ports are often limited to at maximum ten each, and the search is run in advance of the shipment without any firm time restrictions. Also, the reward of finding the global optimal compared to a sub-optimal solution in the shipping industry can be significant in terms of money.

Pseudo code for A* is presented in Algorithm 4.2. We select the Great-circle distance as the heuristic to guide the pathfinder. As discussed in Section 2.2.2, it is crucial that we choose a consistent (and thus also admissible) heuristic, as a consistent heuristic guarantees A* to find the optimal solution. The Great-circle distance is the shortest possible path between two locations at the earth's surface and, because all costs are non-negative, will always be lower than the final cost and thus ensure that A* finds the optimal solution if it exists.

Our implementation of A* is similar to the one described in Section 2.2.2, but modified to fit the maritime domain. Similar to traditional A*, we process the most promising node in the *open* list until the *open* list is empty or we reach the goal. We also calculate the cost and the heuristic at each step and store the values in that node. The cost is the sum of all cost functions multiplied with their corresponding weight. If the calculated cost is better than any previously calculated cost for that node, we store a reference to that connection to be able to traverse back to the start when we reach the goal. Different from traditional A*, we check whether all time windows are obeyed at each step and stop exploring that particular path if not.

In Algorithm 4.2, line 2 and 3 declares the *open* and *closed* lists. The *open* list contains the starting point p_a and the *closed* list is empty. The *while*-loop on line 4 runs as long as the *open* list is not empty. The *open* list gets populated while the pathfinder is running. If the *open* list gets empty and the goal is not reached, there exists no path to the goal. On line 5, we get the cheapest node from the *open* list, meaning the node with the lowest combined cost and heuristic value. If this node is the goal node, we will break from the loop, as achieved with the *if* statement on line 6 and 7.

If we have not reached the goal yet, we look at the outgoing edges from the current node, as shown on line 9 and 10. On lines 11-13, we make sure that we are meeting all time windows. If we do not, we will not explore the current path any further. Then, on line 14, we calculate the cost of the current path as the cost up to the previous node and the cost of the traveled edge.

In the following *if* statement on lines 15-30, we consider three scenarios. The first is if the current node is already in the *closed* list. This means we are processing a node that has already been processed and we check to see if we found a better path on line 16. If that is the case, it might be a globally better path containing the current node. We remove the node from the *closed* list as it is no longer considered fully processed.

The second clause, on lines 22-27, is run if the current node is already in the *open* list. This means we have processed a nearby node and that the current node is queued for processing. We check if the current path is better than the previous path to the same node, and if so update the costs for that node, as shown on lines 23-25. If not, it means there is a cheaper path to same node, and line 26 will stop exploring the current path any further.

If neither of the two previous cases trigger, it means the current node is an un-visited node. In every case, we calculate the heuristic value for the current node as the Great-circle distance to the goal, as explained above.

Finally, on lines 31-36, we store the calculated values and a pointer

to the edge we just traveled. The values will be used in the next iteration of the while loop and the pointer will be used to traverse from the goal node back to the starting node when the algorithm finishes. If the *open* list does not already contain the current node, we add it. When we reach the end of the for loop processing all the edges going out of the current node, on lines 38-39, we move the node from the *open* list to the *closed* list.

After the while loop ending on line 41, we assume the goal is reached. Then, we can find the path from the goal to the starting node by following the pointers to the previous best edge all the way until we reach the start node, as represented by lines 44-47. By reversing the path, as done on line 48, we get the optimal path from the starting node to the goal node. The path and the corresponding cost is returned from the function.

Algorithm 4.1 Applying A* to all possible combinations

Input Graph G with vertices V and edges E, Ships S, All ports P, Ports with cargo available P^{η}, Destination port $p_d \in P \backslash P^{\eta} \backslash V^{\eta}$, Amount of cargo to buy τ
Output Pair(path, cost) the path with the minimum cost

1: **function** PERFORM SEARCH($G(V,E), S, P, P^{\eta}, p_d, \tau$)
2: pathsWithCost ← [ ]
3: **for all** $s_i \leftarrow s_0, \ldots, s_o \in S$ **do** ▷ Loop all ships
4: **for all** $p_{a,s_i} \leftarrow p_{0,s_i}, \ldots, p_{n,s_i} \in P^{\eta,s_i}$ **do** ▷ Loop all available loading ports
5: path1, cost1 ← PERFORM A*($G(V,E), Ps_i, v_{s_i,0}, p_{a,s_i}$) ▷ From initial position to loading port
6: path2, cost2 ← PERFORM A*($G(V,E), Ps_i, p_{a,s_i}, p_d$) ▷ From loading port to destination port
7: pathsWithCost[p_{a,s_i}] ← (path1 + path2, cost1 + cost2 + $\tau \cdot \gamma_{p_{a,s_i}}$)
8: **end for**
9: **end for**
10: **return** min(pathsWithCost)
11: **end function**

4.3.2 Dynamic Programming Approach

In order to validate the optimality of the prototype, we implement a search using dynamic programming (DP) to explore all possible paths to determine the optimal one. The DP approach does not use any heuristic and will process all nodes and edges.

The dynamic programming approach takes all outgoing edges from

the starting position and calculates and saves the cost of traveling those edges together with a pointer to the previous node with the cheapest cost. Then it takes the end node of the processed edges and runs the search on all the outgoing edges recursively, terminating when all edges have been processed. When finished, it is possible to follow the pointers to previous cheapest node from the goal node and back to the starting node, resulting in the optimal path.

To make the search more efficient, we stop exploring paths that lead to a node we have already visited if it has an already lower cost. If that is the case it means we have found a more expensive path to the same node, which we are not interested in and can therefore stop exploring that path. Similarly, if a path does not meet the required time windows we will also stop exploring that path. Pseudo code for the DP algorithm is provided in Algorithm 4.3.

The DP approach in Algorithm 4.3 starts off by defining three global key-value stores in the first block. These will hold pointers to the vertex's previous best vertex, the previous vertex's best cost, and the previous vertex's best time for each of the nodes processed by the algorithm.

The second block consist of two parts. The first part (lines 2-4) finds the optimal path from the start position to the pickup port. The second part (lines 5-7) finds the optimal path from the pickup port to the destination port. Both parts start by getting all the outgoing edges from each part's starting position and pass them to the recursive function *ExploreEdges*, which will process all the edges in a Breadth-First [36] approach. When done, the optimal path (and the corresponding cost) can be found by following the pointers from the goal node and back to the starting node in the same way as with A*.

The *ExploreEdges* function in the last block is a tail recursive function to iterate all the edges in the graph. It first checks the stop condition for the recursion (any more edges to process) on line 2 and returns if it is met. Then it initializes an empty list that will contain all the edges to be processed in the next iteration on line 5. We loop the edges from the input on line 6. On line 7 we get all the outgoing

edges from the end node of the current edge and append them to the list defined on line 5. We check the time window constraints on lines 8-10 and calculate the cost for the current path so far at line 11. If the cost is lower than any previous cost calculated for the same vertex, we update the pointers defined in the first block, as shown in lines 12-15. If the cost is higher, we stop exploring that path as it will never be better than the already cheaper path, as shown in the else clause on lines 16-18.

The if statement on lines 19-21 stops exploring the current path if we have reached the goal. When reaching the end of the for loop on line 22, we have processed all edges in the function input. We take the now populated list of next edges to explore and call the same function on line 23. The recursion stops when all edges have been processed.

4.3.3 Cost Functions

To dynamically apply cost to an edge based on the current state, we define a set of cost functions. Both algorithms support the following costs:

- **Operation Cost:** The cost of operating an empty or loaded ship. This cost includes fuel costs, staff, insurance, etc.
- **Polygon Cost:** Cost of traveling through a certain area, for example, the Suez Canal or the Panama Canal.
- **Polygon Gradient Cost:** Cost for a certain area that increases the longer a ship is in the area, the further it travels in a certain direction, or by some other measurement. For example, the cost of traveling in the Antarctic Sea increases as the latitude increase.
- **Polygon Direction Dependent Cost:** Cost for traveling in a certain area and with a certain direction or within a specified heading range. The cost decreases with a factor of the difference between the specified target heading and the actual

heading. This is useful for modeling weather and sea conditions.

- **Time Window Constraint:** A specified time window in which a ship must enter the port to be serviced. Failing to meet the time window will result in an invalid path.
- **Cargo Cost:** Cost per tonne of buying cargo at a specific port.

4.4 Complexity

A* Algorithm

The complexity of A* is similar to the Breadth-First Search [36] algorithm in that it expands all nodes at each step. Assuming the worst-case scenario without heuristics, we will at first step process all outgoing edges b from the starting node, and in the next iteration process every outgoing edge for all the edges in the previous step, b^2, and so on. For a solution with depth d, the algorithm will process the following number of nodes:

$$b + b^2 + b^3 + \ldots + b^{d-1} + b^d,$$

and hence the complexity of $O(b^d)$.

However, A*'s complexity also depends on the quality of the heuristic. To measure the quality we introduce the heuristic's *relative error*, that is, how far off the heuristic is from optimal. According to Russell and Norvig [36], the relative error is defined as

$$\epsilon \equiv \frac{h^* - h}{h^*},$$

where h^* is the actual cost of traveling from the starting node to the destination node and h is the value of our heuristic function. As the

heuristic h is admissible (always underestimates the final cost), then

$$0 \leq \epsilon \leq 1,$$

and

$$b^{\epsilon d} \leq b^d.$$

With the relative error ϵ, we can more precisely define the complexity of A* as $O(b^{\epsilon d})$, where b is the branching factor of the nodes and d is the depth of the solution. The complexity increases as the length of the solution increase because the algorithm must process exponentially more nodes. However, if the heuristic is suitable, the relative error ϵ will be low and thus reduce the complexity.

Dynamic Programming Approach

The dynamic programming approach always processes all nodes and edges resulting in a complexity of $O(|V| \cdot |E|)$ for a graph $G(V, E)$ where V is the set of vertices and E is the set of edges [46, 47]. The actual implementation does, as previously discussed, include optimizations to reduce the number of edges to process, but in the worst case will process all vertices and edges.

Algorithm 4.2 A* pseudo code

Input Graph G with vertices V and edges E, Ports P, Ship $s_i \in S$, Starting position $p_a \in P$, Goal position $p_b \in P$

Output Pair of path and cost to the path with the lowest cost.

```
1: function PERFORM A*(G(V, E), P, s_i, p_a, p_b)
2:     openList ← [p_a]                                          ▷ List containing p_a
3:     closedList ← [ ]                                          ▷ Empty list
4:     while openList ≠ ∅ do
5:         v_a ← GETCHEAPESTELEMENT(openList)
6:         if v_a = p_b then                                     ▷ Reached goal
7:             break
8:         else
9:             E' ← GETOUTGOINGCONNECTIONS(v_a)
10:            for all e_{a,b} ∈ E' do        ▷ Recall that e_{a,b} is an edge from v_a to v_b
11:                if v_b = p_j and not a_{s_i,p_j} ≤ t_{s_i,p_j} ≤ b_{s_i,p_j}   ∀p_j ∈ P then
12:                    continue  ▷ Did not reach time window. Stop exploring this path
13:                end if
14:                nextCost ← v_a.costSoFar + Σ_{k=0}^{|C|} c_{s_i,k} · w_{s_i,k}
15:                if v_b ∈ closedList then
16:                    if v_b.costSoFar ≥ nextCost then          ▷ Found better path
17:                        closedList − = v_b
18:                        heuristicValue ← CALCULATEHEURISTIC(v_b, p_b)
19:                    else
20:                        continue        ▷ Not a better path. Stop exploring this path.
21:                    end if
22:                else if v_b ∈ openList then
23:                    if v_b.costSoFar ≥ nextCost then          ▷ Found better path
24:                        heuristicValue ← CALCULATEHEURISTIC(v_b, p_b)
25:                    else
26:                        continue        ▷ Not a better path. Stop exploring this path.
27:                    end if
28:                else                                          ▷ v_b is a new un-visited node
29:                    heuristicValue ← CALCULATEHEURISTIC(v_b, p_b)
30:                end if
31:                v_b.costSoFar ← nextCost
32:                v_b.estimatedTotalCost ← nextCost + heuristicValue
33:                v_b.prevBestEdge ← e_{a,b}
34:                if v_b ∉ openList then
35:                    openList + = v_b
36:                end if
37:            end for
38:            openList − = v_b
39:            closedList + = v_b
40:        end if
41:    end while                                                 ▷ No more nodes to process
42:    finalCost ← v_a.costSoFar
43:    path ← [ ]
44:    while v_a ≠ p_a do                                        ▷ Traverse back to the start
45:        path + = v_a.prevBestEdge
46:        v_a ← v_a.prevBestEdge_a
47:    end while
48:    return (REVERSE(path), finalCost)
49: end function
```

Algorithm 4.3 Dynamic programming approach pseudo code

Input Graph G with vertices V and edges E, Ship $s_i \in S$, Port with available cargo $p_a \in P^{\eta,s_i}$, Destination port $p_d \in P \backslash P^{\eta} \backslash V^{\eta}$, Amount of cargo to buy τ

```
1: global variables
2:     prevBestVertex ← {}
3:     prevBestCost ← {}
4:     prevBestTime ← {}
5: end global variables

1: function PERFORM DP SEARCH(G, s_i, p_a, p_d, τ)
2:     E' ← G.GETOUTGOINGEDGES(v_{s_i,0})
3:     EXPLOREEDGES(E', p_a, G)                    ▷ From start position to loading port
4:     (path1, cost1) ← EXTRACTBESTPATHFROMPOINTERS(p_a)

5:     E' ← G.GETOUTGOINGEDGES(p_a)
6:     EXPLOREEDGES(E', p_d, G)                    ▷ From loading port to destination port
7:     (path2, cost2) ← EXTRACTBESTPATHFROMPOINTERS(p_d)
8:     return (path1 + path2, cost1 + cost2 + τ · γ_{p_a,s_i}
9: end function

1: function EXPLOREEDGES(E', p_j, G)
2:     if E' = ∅ then
3:         return
4:     end if

5:     E'' ← [ ]
6:     for all e_{a,b} ∈ E' do                     ▷ Recall that e_{a,b} is an edge from v_a to v_b
7:         E'' ← E'' + G.GETOUTGOINGEDGES(v_b)
8:         if v_b = p_j and not a_{s_i,p_j} ≤ t_{s_i,p_j} ≤ b_{s_i,p_j}   ∀p_j ∈ P then
9:             continue                            ▷ Did not reach time window. Stop exploring this path
10:        end if

11:        cost ← prevBestCost[v_a] + s_i.CALCULATECOST(e_{a,b})
12:        if cost < prevBestCost[v_b] then
13:            prevBestCost[v_b] ← cost                    ▷ Found a cheaper path to v_b
14:            prevBestVertex[v_b] ← v_a
15:            prevBestTime[v_b] ← currentTime
16:        else if cost >= prevBestCost[v_b] then
17:            break    ▷ Found a more expensive path to v_b. Stop exploring this path.
18:        end if

19:        if v_b == p_j then
20:            break                ▷ Reached goal. Do not continue exploring this path
21:        end if
22:    end for

23:    EXPLOREEDGES(E'', p_j, G)                   ▷ Recursive call until all edges are evaluated
24: end function
```

Chapter 5

Results

In this chapter, we present the results of a large scale simulation on 990 different shipment orders. We evaluate the performance of the prototype and compare it to the dynamic programming approach. The results are examined and matched to the complexity analysis in the previous chapter. We show examples of how costs and constraints affect the generated routes.

5.1 Simulation Environment

The prototype is a *Java Virtual Machine* based software written in the Kotlin programming language, running Java 10.0.2 and Kotlin 1.3.30. All simulations shown in this chapter use the configuration listed in Table 5.1. We refer the reader to the GitHub-repository[1] for source code, a full list of dependencies, examples, and further instructions on how to use the platform. To visualize the results, we use the GeoJSON format, and we suggest using *GeoJSON.io*[2] for viewing such files. All ports are identified by its UN/LOCODE[3].

[1] `https://github.com/Dambakk/ShippingRouter`

[2] `http://geojson.io`

[3] United Nations Code for Trade and Transport Locations, `https://www.unece.org/cefact/locode/service/location`.

Cost type	Info	Value/Cost/Time window	Weight
Ship Constraint	Max capacity (DWT)	1 000 tonnes	
Operation Cost	Empty ship	25	
Operation Cost	Loaded ship	250	
Polygon Cost	Suez Canal	10 000	1.0
Polygon Cost	Panama Canal	10 000	1.0
Polygon Gradient Cost	Antarctica	Function of distance	1.0
Polygon Direction Dependent Cost	Taiwan Strait	100 000 (heading=255, angle=90)	1.0
Polygon Direction Dependent Cost	Gulf Stream	100 (heading=45, angle=90)	1.0
Time Window Constraint	ARRGA	0 - 10 000	
Time Window Constraint	AUBUY	0 - 100 000	
Time Window Constraint	CNXGA	0 - 10 000	
Time Window Constraint	JPETA	0 - 5 000	
Time Window Constraint	PHMNL	0 - 5 000	
Time Window Constraint	QAMES	0 - 4 500	
Time Window Constraint	USCRP	0 - 7 000	
Cargo Cost (pr tonne)	ARRGA	1 000	
Cargo Cost (pr tonne)	AUBUY	5 000	
Cargo Cost (pr tonne)	BMFPT	100	
Cargo Cost (pr tonne)	CNTAX	400	
Cargo Cost (pr tonne)	CNTXG	1 200	
Cargo Cost (pr tonne)	JPETA	1 500	
Cargo Cost (pr tonne)	KRYOS	8 000	
Cargo Cost (pr tonne)	PHMNL	300	
Cargo Cost (pr tonne)	QAMES	600	
Cargo Cost (pr tonne)	USFPO	1 900	
Cargo Cost (pr tonne)	USWWO	6 500	

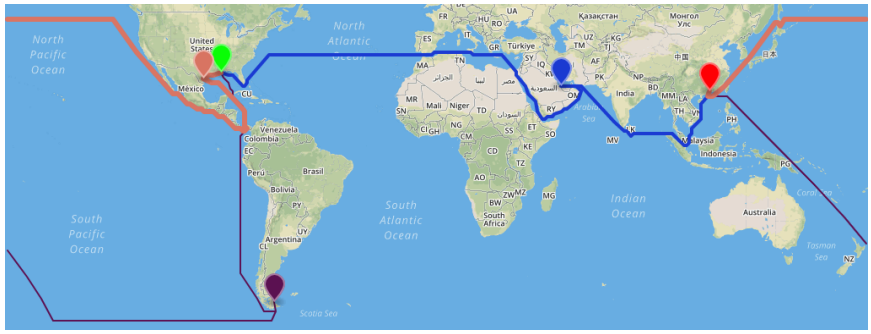
Table 5.1: Simulation configuration

5.2 Simulation Results for Concrete Examples

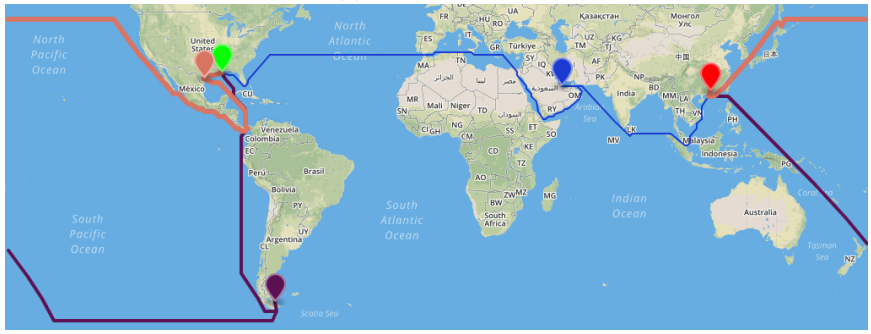
To illustrate how the prototype works, we will first demonstrate how adding costs and constraints effects the generated routes. Then, we will show how the prototype performs a simulation by example.

Each shipment order has a fixed destination port. The factors that can be decided by the shipping company to affect the cost is which ship to assign to the shipment and where to buy the cargo. For the following example, we assume that we only have one available ship with a starting location in the port CNXGA (red marker) in China, a shipment to be delivered in port USWWO (green marker) in the USA, and three possible ports to buy cargo in: QAMES (blue) in Qatar, ARRGA (purple) in Argentina, and USCRP (orange) in the USA. Figure 5.1 shows the possible routes when only performing

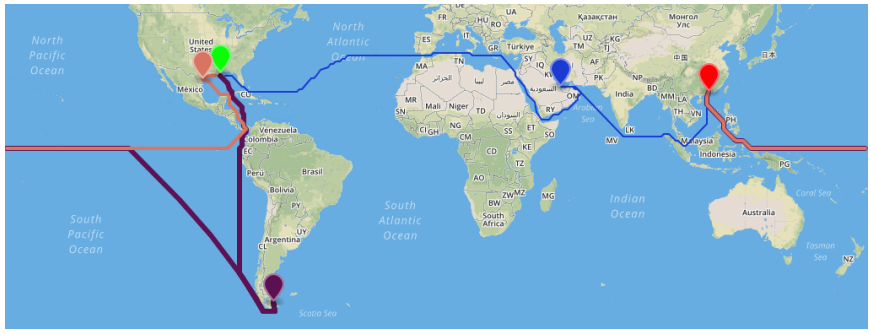
pathfinding based on distance (named as naïve pathfinding), when adding operation and cargo cost, and when adding all costs listed in Table 5.1. A thicker line means a cheaper route compared to the other routes in the current simulation.



(a) Naïve algorithm.



(b) Solution with cargo cost and operation cost.



(c) Solution with all costs and constraints.

Figure 5.1: Examples of routes generated and compared by the prototype.

In Figure 5.1(a), the prototype performs naïve pathfinding, using no additional costs other than the Great-Circle distance. As we can see, the pickup port in the USA, close to the destination, has the thickest line as it is the cheapest route out of these three alternatives. Notice how the paths tend to stay north as long as possible when above the equator, and south as long as possible when below the equator. This is expected behavior and is due to the curvature of the earth and Great-circle geometry making it the shortest path on a sphere.

When adding operation costs and cargo prices according to Table 5.1, the prototype generate the routes in Figure 5.1(b). Although the algorithm suggests the same routes as in Figure 5.1(a), it now states that the route with pickup port in Argentina is cheaper than the route with a pickup port in Qatar, while still recommending the route with a pickup port in the USA, close to the destination, as the cheapest.

Figure 5.1(c) shows the generated routes when all cost functions and constraints listed in Table 5.1 are applied. In contrast to the two previous figures, the path from China to Argentina now tends to stay close to the equator for as long as possible before heading south to the pickup port. This is because of the additional cost of traveling in the Antarctic Sea. Also, the path crossing the North Atlantic Ocean now avoids the north-west stream and wind of the Gulf Stream. Even though the pickup port in the USA is closer to the destination, in this example, all the combined costs make buying cargo in Argentina the overall cheapest route.

Until now, the examples have assumed only one available ship to select for each shipment. When using the prototype on a real-world shipment order, there are typically several available ships spread across the world. To demonstrate, we let all ships have the same attributes (size, loading capacity, etc.), but different starting location. Then we can search every combination of available ship and pickup port for the one resulting in the cheapest shipment. As an example, Table 5.2 shows the result after simulating transporting cargo to the port USWWO. It considers all available ships and pickup ports, and finds that selecting the ship in the CNTAX port and buying cargo in the BMFPT port results in the cheapest shipment, as highlighted

Variable factors		*Fixed factors*	
Ship's position	**Loading port**	**Destination port**	**Cost**
CNTAX	ARRGA	USWWO	4916605
CNTAX	AUBUY		11101790
CNTAX	BMFPT		1208412
CNTAX	KRYOS		13352830
...	...		...
CNTAX	USFPO		2784368
...	...	USWWO	...
BMFPT	ARRGA	USWWO	4540113
BMFPT	CNTAX		6644714
BMFPT	JPETA		7342133
...	...		...
BMFPT	USFPO		2144020

Table 5.2: Finding the cheapest combination of ship and loading port for transporting cargo to USWWO.

in green. The same procedure is applied to every shipment order simulation. Note that the route for each combination of ports can be vizualized in the same manner as in Figure 5.1.

Recall that searching all combinations, as in Table 5.2, is affordable as the available ships and pickup ports are often limited, there are no firm time restrictions, and the reward of finding the global optimal is significant in terms of money.

5.3 Simulation Results in Statistics

In order to evaluate the overall performance of the prototype, we run a large scale simulation. The shipment orders in this simulation are based on all combinations of three out of 11 ports and all permutations of these three ports, replicating in total 990 distinct shipments (all with different combinations of starting position, pickup port and destination port). As in the previous examples, we let all ships have the same attributes, but different starting locations, as represented by the *Ship pos* column. For each ship, we compare the available pickup ports, denoted by the *pickup port* column. The *Dest port* column represent the destination port for the current shipment. In

#				Solution depth		Cost	Num Nodes		Num Edges	
	Ship pos	Pickup port	Dest port	A*	DP		A*	DP	A*	DP
1	CNTAX	ARRGA	USFPO	268	268	4960025	46131	62767	358716	720243
2	CNTAX	CNTXG	ARRGA	188	188	6738294	26065	62767	201649	868401
3	JPETA	USWWO	ARRGA	264	264	10191607	34704	62767	269186	805081
4	CNTXG	JPETA	PHMNL	31	31	2309379	1124	62767	8370	857448
5	USFPO	BMFPT	ARRGA	106	106	3106289	12638	62767	98143	764100
...										
989	USWWO	QAMES	BMFPT	408	408	6335884	54402	62767	419998	763638
990	USWWO	USFPO	KRYOS	179	179	7273761	27606	62767	213846	768729
Average				**229.43**	**229.43**	**6394765.19**	**32898.92**	**62767.00**	**254590.64**	**816619.94**

Table 5.3: Simulation results

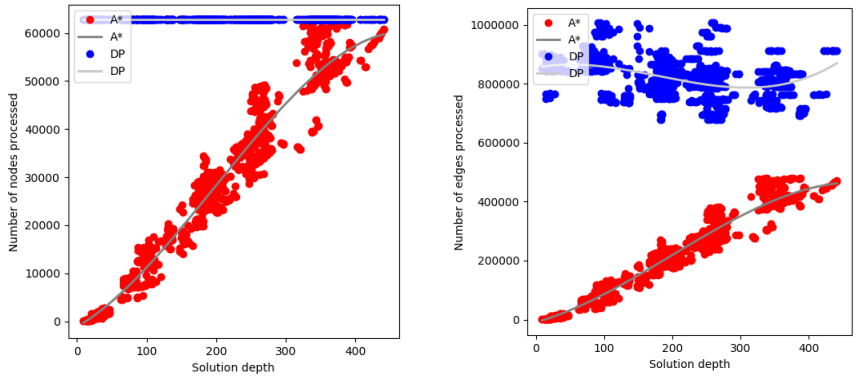
the remaining columns, we compare the length (number of edges) and the estimated cost of the generated paths, as well as the number of nodes and edges processed by each algorithm to generate the solution.

As expected, the A* algorithm always find the same, optimal path as the dynamic programming approach, confirming that A*'s heuristic is admissible[4]. We also see that A* processes fewer nodes than the DP algorithm which always processes all nodes. This is expected as A*'s search is limited by the heuristic. Likewise, A* processes fewer edges than DP in order to obtain the same result. Figure 5.2 visualizes the relation between the number of nodes and the solution depth, as well as the number of edges and the solution depth.

Figure 5.2(a) shows the number of nodes each algorithm processes to find the optimal path. As we can see, the number of nodes required for the A* algorithm increases in an exponential manner as the solution depth increase until it approaches the maximum number of nodes in the graph. This correlates to the complexity $O(b^{\epsilon d})$, where the complexity is exponential to the depth d of the solution. The DP approach always handles all nodes, hence the line, and correspond to the first factor of DPs complexity $O(|V| \cdot |E|)$.

Figure 5.2(b) shows the number of edges that are being processed by each algorithm. Similar to Figure 5.2(a), A* has an exponential tendency until it reaches the maximum number of nodes in the graph. Even though DP's complexity suggests that it should process the same number of edges regardless of the solution depth, optimizations

[4]The heuristic always underestimates. See Section 2.2.2.



(a) Relation between solution depth and number of nodes processed to find optimal path.

(b) Relation between solution depth and number of edges processed to find optimal path.

Figure 5.2: Comparing the number of nodes and vertices processed by the two approaches.

to the implementation that excludes invalid and expensive paths decrease the number of edges to process, hence the variations as shown in the figure. In the worst-case scenario, DP will also handle all edges.

Although A*'s complexity, $O(b^{\epsilon d})$, is exponential in terms of solution depth and DP has complexity $O(|V| \cdot |E|)$, the error factor, ϵ, representing the heuristic, will make sure the complexity of A* is still lower than DP's, as confirmed by the figures.

Chapter 6

Conclusion and Future Work

This chapter summarizes and concludes the work and the software presented in this thesis. We also suggest improvements to the prototype and areas that require further research.

6.1 Conclusions

In this thesis, we have defined and modeled the Maritime Pickup and Delivery Problem with Cost and Time Window Constraints to help ship chartering companies estimate and compare the cost of shipments. The model gives a precise definition of the underlying graph, the objective function, and the constraints that apply in the shipping domain. Together, they make a formal description of an optimization problem, which is conjectured as NP-hard, and we solved it using a custom-made, A* based algorithm.

The algorithm is part of a software platform capable of generating and comparing solutions in order to find the optimal one across a given set of ships and ports. We demonstrated that the A* based algorithm is able to generate optimal paths and find the best compro-

mise of the given costs and constraints with a complexity of $O(b^{\epsilon d})$. An implementation using dynamic programming verified the results.

6.2 Future Work

The current state of the prototype provides a framework to define real-world costs and constraints, and the focus of future work should be to use this framework to precisely replicate the domain. Specifically, we suggest the following improvements for achieving a more accurate model:

1. A real-time weather modeling tool
2. Predicting cargo prices
3. Real-time ship data for replanning
4. Support for repositioning, split loads, and ship co-operation

To begin with, we suggest putting effort into converting weather data to suitable cost functions. The weather affect ship routes, both in the short-term and long-term and modeling it precisely can give better cost estimations. People with domain knowledge must define the cost of each factor and decide its weighting compared to other costs.

Secondly, being able to predict cargo prices is crucial to know whether to buy or wait and is already a hot research topic for shipping companies. Extending the prototype using neural networks to predict cargo prices can enable the prototype to compare shipments in terms of time in the future and decide whether to buy or wait.

Thirdly, by implementing real-time input from ships at sea, this prototype can extend its use from planning to cost estimation of replanning routes for ships at sea. Replanning may occur when new information becomes available or factors that affect the route, like weather, changes.

Finally, research on other, more mature Vehicle Routing Problems focuses on features like co-operation of multiple vehicles, splitting loads to optimize vehicle capacity, and minimize repositioning. All these concepts apply to our problem and should be studied further.

Moreover, in terms of implementation, we suggest putting more effort into the graph abstraction. This includes both creating a more detailed graph when close to ports or in canals, as well as inserting well-known long-distance routes. To programmatically create a more detailed graph when needed, we suggest looking into applying Adaptive Mesh Refinement, a technique to increase grid resolution in areas of interest.

Although the prototype finds solutions within a reasonable time, the running time can be improved. The current implementation uses parallelization to run up to four simulations in parallel. Refactoring could enable the program to run even more simulations at the same time and thus shorten the running time even further. Also, A* relies on heavy list manipulations. Carefully selecting and implementing data structures may also help reduce running time.

References

[1] W. Commons, "Sabrina I," 2006.

[2] D. Toropila, F. Dvorak, O. Trunda, M. Hanes, and R. Barták, "Three approaches to solve the petrobras challenge: Exploiting planning techniques for solving real-life logistics problems," pp. 191–198, 11 2012.

[3] R. Barták and N.-F. Zhou, "On modeling planning problems: Experience from the petrobras challenge," in *Advances in Soft Computing and Its Applications* (F. Castro, A. Gelbukh, and M. González, eds.), (Berlin, Heidelberg), pp. 466–477, Springer Berlin Heidelberg, 2013.

[4] R. Barták and n.-f. Zhou, "Using tabled logic programming to solve the petrobras planning problem," *Theory and Practice of Logic Programming*, vol. 14, 05 2014.

[5] A. Charisis, N. Mitrovic, and E. Kaisar, "Container shipping route and schedule design with port time windows and coordinated arrivals," 11 2018.

[6] I. S. Dolinskaya, "Optimal path finding in direction, location, and time dependent environments," *Naval Research Logistics (NRL)*, vol. 59, no. 5, pp. 325–339, 2012.

[7] E. Beeker, "An algorithm for finding the shortest sailing distance from any maritime navigable point to a designated port," *Mathematical and Computer Modelling*, vol. 39, no. 6, pp. 641 – 647, 2004. Defense transportation: Algorithms, models, and applications for the 21st century.

[8] N. Hazim, S. S. M. Al-Dabbagh, and M. A. S. Naser, "Pathfinding in strategy games and maze solving using A* search algorithm," *Journal of Computer and Communications*, vol. 04, pp. 15–25, 01 2016.

[9] and, "A comparative study of a-star algorithms for search and rescue in perfect maze," in *2011 International Conference on Electric Information and Control Engineering*, pp. 24–27, April 2011.

[10] S. Redfer, "Player-traced empirical cost-surfaces for A* pathfinding," 2011.

[11] B. Stout, "Smart moves: Intelligent pathfinding," 1997.

[12] J. Bruce and M. Veloso, "Real-time randomized path planning for robot navigation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, pp. 2383–2388 vol.3, Sep. 2002.

[13] Z. A. Algfoor, M. S. Sunar, and H. Kolivand, "A comprehensive study on pathfinding techniques for robotics and video games," *Int. J. Comput. Games Technol.*, vol. 2015, pp. 7:7–7:7, Jan. 2015.

[14] S. Mirjalili, J. Song Dong, and A. Lewis, *Ant Colony Optimizer: Theory, Literature Review, and Application in AUV Path Planning*, pp. 7–21. Cham: Springer International Publishing, 2020.

[15] O. Souissi, R. Benatitallah, D. Duvivier, A. Artiba, N. Belanger, and P. Feyzeau, "Path planning: A 2013 survey," in *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)*, pp. 1–8, Oct 2013.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, July 1968.

[17] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, Dec 1959.

[18] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth National Conference on Artificial Intelligence*, (Menlo Park, CA, USA), pp. 476–483, American Association for Artificial Intelligence, 2002.

[19] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artif. Intell.*, vol. 155, pp. 93–146, May 2004.

[20] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, vol. 1, pp. 7–28, 2004.

[21] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," tech. rep., 1998.

[22] L. Kavraki and J. . Latombe, "Randomized preprocessing of configuration for fast path planning," in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pp. 2138–2145 vol.3, May 1994.

[23] L. E. Kavraki, P. Svestka, J. . Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, pp. 566–580, Aug 1996.

[24] M. Čáp, P. Novák, J. Vokrínek, and M. Pěchouček, "Multi-agent rrt: Sampling-based cooperative pathfinding," in *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, (Richland, SC), pp. 1263–1264, International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[25] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, vol. 2, pp. 500–505, March 1985.

[26] M. Mabrouk and C. McInnes, "Solving the potential field local minimum problem using internal agent states," *Robotics and Autonomous Systems*, vol. 56, no. 12, pp. 1050 – 1060, 2008. Towards Autonomous Robotic Systems 2008: Mobile Robotics in the UK.

[27] I. Millington and J. Funge, *Artificial Intelligence for Games, Second Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd ed., 2009.

[28] S. A. Lawrence, *International sea transport: the years ahead [by] S. A. Lawrence*. Lexington Books Lexington, Mass, 1972.

[29] K. Fagerholt, "Ship scheduling with soft time windows: An optimisation based approach," *European Journal of Operational Research*, vol. 131, pp. 559–571, 02 2001.

[30] H. Andersson, M. Christiansen, and K. Fagerholt, "The maritime pickup and delivery problem with time windows and split loads," *INFOR: Information Systems and Operational Research*, vol. 49, no. 2, pp. 79–91, 2011.

[31] ICKEPS, "Planning ship operations on petroleum platforms and ports." online, 2012.

[32] A. Gerevini and D. Long, "BNF description of PDDL3.0." online, 2005.

[33] C.-W. Hsu and B. Wah, "The SGPlan planning system in IPC-6," 08 2008.

[34] S. , S. , and N. Dr, "Applications of graph theory in computer science an overview," *International Journal of Engineering Science and Technology*, vol. 2, 09 2010.

[35] D. Ferguson, M. Likhachev, and A. Stentz, "A guide to heuristic-based path planning," 01 2005.

[36] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[37] R. Dechter and J. Pearl, *The Optimality of A**, pp. 166–199. New York, NY: Springer New York, 1988.

[38] D. Sekar, "Tanker world scale index."

[39] P. Toth and D. Vigo, *The Vehicle Routing Problem*. Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, 2002.

[40] Y. Gong, J. Zhang, O. Liu, R. Huang, H. S. Chung, and Y. Shi, "Optimizing the vehicle routing problem with time windows: A discrete particle swarm optimization approach," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, pp. 254–267, March 2012.

[41] M. W. P. Savelsbergh, "Local search in routing problems with time windows," *Annals of Operations Research*, vol. 4, pp. 285–305, Dec 1985.

[42] M. M. Solomon, "Algorithms for the vehicle routing and scheduling problems with time window constraints," *Operations Research*, vol. 35, no. 2, pp. 254–265, 1987.

[43] H.-F. Wang and Y.-Y. Chen, "A genetic algorithm for the simultaneous delivery and pickup problems with time window," *Computers & Industrial Engineering*, vol. 62, no. 1, pp. 84 – 95, 2012.

[44] "The great circle distance," Apr 2018.

[45] Butler *et al.*, "The GeoJSON Format," RFC 7946, RFC Editor, August 2016.

[46] R. Bellman, *On a Routing Problem*. P (Rand Corporation), Rand Corporation, 1956.

[47] A. V. Goldberg and T. Radzik, "A heuristic improvement of the bellman-ford algorithm," *Applied Mathematics Letters*, vol. 6, no. 3, pp. 3 – 6, 1993.