Norwegian University of
Science and Technology

# Ship Path Planning using Navigational Charts with Time-Dependent Weather Constraints

**Sverre Velten Rothmund**

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Tor Arne Johansen, ITK
Co-supervisor: Tor Eirik Østrem, Marine Technologies, LLC

Norwegian University of Science and Technology
Department of Engineering Cybernetics

# Problem Description

Automatically planning paths is an essential first step towards a more autonomous shipping industry. Currently, paths are planned using human operators who insert every waypoint that constitutes the path. Ships are outfitted with autopilots that are able to follow the designed routes, and have a simple collision alert systems that bring the captains attention towards approaching vehicles and obstacles. Safety is the largest concern when automatically making a path that a ship can follow. The path must avoid shallow waters where there is a risk of grounding, and avoid areas with dangerous weather conditions which can damage or in the worst case capsize the ship. The path should follow the dictated flow of traffic in traffic separation schemes to minimize the risk of collision.[1]

## Assignment

The project consists of programmatically generating a path that is safe for a ship to follow. The project should extract data from navigational charts using the NAVTOR SDK. The path must be safe with respect to the following:

- Avoiding grounding and collisions with rocks, ship-wrecks, bridges, cables, and other features present on the navigational charts.
- Avoiding dangerous weather conditions.
- Avoiding traversing traffic separation schemes in the wrong direction.
- The path should be kinetically feasible for the ship to follow.

The project can be split into the following parts:

- Survey the literature for different path planning strategies that can be used to solve the problem at hand
- Develop a path planning algorithm that can plan a safe path between two coordinates without human intervention.
- Test the path planner on official navigational charts.

[1] The problem description is developed in collaboration with Tor Eirik Østrem at Marine Technologies LLC. The state of the art is taken from his experience in the industry.

# Preface

This project marks the end of my 5-year master program in Engineering Cybernetics at NTNU. This master thesis is done in the 10th semester and constitutes a full semester of work, giving 30 credits. This work in its entirety is done between the period of 8th of January 2018 and 31st of May 2018, and is not a continuation of earlier work.

This project is done in a collaboration with Marine Technologies LLC. They supported the project with guidance and access to the following:

- NAVTOR API: an application program interface used to access navigational chart.
- Navigational charts, which amongst other covered the coast of Norway north of Haugesund and south of Molde.
- NAVTOR G-ECDIS: A software used in ship navigation to display the ship's position and planned path on top of the loaded navigational charts.

No literature or theory was provided. Finding these sources was a part of the work of this project.

I would like to thank Tor Eirik Østrem which was my contact person at Marine Technologies for having almost weekly guidance sessions with me. He helped me with acquiring the required material and knowledge from Marine Technologies, discussions on where the project should go, and discussions around the different problems in this project.

Furthermore, I would like to thank Nickolay Sergeev at Navtor for the extensive help he contributed with on figuring out how to use the Navtor API and for the help on debugging errors caused by the API. The limited documentation of the API would have slowed the project down even further had it not been for his help.

Lastly, I would like to thank Tor Arne Johansen which was my main supervisor at NTNU for a discussion around my proposed solution of the project, and for giving me some feedback on this thesis.

Trondheim, 01.06.2018
Sverre Velten Rothmund

## Abstract

Automatically planning safe paths for ships to follow will be an important step towards a more autonomous shipping industry. This thesis develops an algorithm that generates safe paths that avoid grounding, sailing in dangerous weather conditions, and sailing against the dictated flow of traffic. The algorithm uses the NAVTOR API to extract data from navigational charts, and uses this chart data to find a path that avoids grounding and violating traffic separation schemes. A continuous state bidirectional A* algorithm that takes the kinematic constraints of the ship into account is implemented and tested in multiple situations on official navigational charts. The search algorithm is developed further to handle bidirectional search with time-dependent constraints. This is used to generate a path that avoids sailing in dangerous weather which changes throughout the voyage. The algorithm will either wait out the weather at the initial port, or find a route that circumvents the dangerous area. The resulting algorithm generates safe paths with respect to grounding and weather constraints, and manages to follow the traffic routing schemes when they are implemented in a consistent manner. Further work is required to handle cases where the traffic routing schemes cannot be interpreted literally. Some changes to the way navigational charts are designed are outlined which would make this process easier.

# Sammendrag

Automatisk planlegging av sikre skipsruter vil være et viktig steg mot en autonom skipsfartsindustri. Denne masteroppgaven utvikler en algoritme som genererer trygge skipsruter som unngår grunnstøt, unngår seiling i farlige værforhold, og unngår å seile imot definerte kjøreretninger. Algoritmen bruker NAVTOR API for å hente ut data fra sjøkart, og bruker dette til å unngå grunnstøt og brudd på trafikkseparasjonssystemer. En bidireksjonell A* algoritme med kontinuerlige tilstander som tar hensyn til skipets kinematiske begrensinger er blitt implementert og testet i forskjellige situasjoner på offisielle havkart. Algoritmen er videre utviklet til å håndtere bidireksjonelt søk med tidsvarierende begrensninger. Dette brukes for å unngå farlige værforhold som endrer seg i løpet av reisen. Skipet vil prøve å seile rundt de farlige værforholdene, eller vente i avreisehavna til værforholdene forbedrer seg. Den resulterende algoritmen klarer å lage en trygg rute som unngår grunnstøt og ferdsel i farlige værforhold, samt å følge trafikkseperasjonssystemer når de er implementert på en konsekvent måte. Videre arbeid trengs for å håndtere tilfeller der trafikkseperasjonssytemene ikke kan tolkes bokstavelig. Mot slutten av oppgaven presenteres noen forslag til endringer i måten havkart utformes som vil gjøre dem mer egnet for autonom baneplanlegging.

# Contents

Problem description . . . . . i
Preface . . . . . ii
Abstract . . . . . iii
Sammendrag . . . . . iv

**1 Introduction** **1**
1.1 Background and motivation . . . . . 1
1.2 Outline . . . . . 3
1.3 Problem Formulation . . . . . 4
1.3.1 Constraints . . . . . 5
1.4 Contributions . . . . . 6

**2 Theory** **7**
2.1 Coordinate transformation . . . . . 7
2.1.1 Great-circle distance . . . . . 7
2.1.2 Translation . . . . . 8
2.1.3 Initial heading . . . . . 8
2.2 Nonholonomic system . . . . . 9
2.3 Line Of Sight guidance . . . . . 9
2.4 Problem solving by graph search . . . . . 12
2.4.1 Dijkstra's algorithm . . . . . 14
2.4.2 A* . . . . . 15
2.4.3 Bidirectional search algorithm . . . . . 15
2.4.4 Time dependent bidirectional search . . . . . 16

**3 Related work** 19
3.1 Combinatorial search . . . 19
3.1.1 Cell decomposition . . . 19
3.1.2 Probabilistic roadmap . . . 21
3.1.3 Voronoi diagram . . . 23
3.1.4 Visibility graphs . . . 24
3.1.5 Rapidly-exploring random trees . . . 25
3.1.6 Hybrid A* . . . 26
3.2 Optimization based path planners . . . 27
3.3 Applicability to this thesis . . . 28
3.4 Local collision avoidance . . . 29
3.5 Commercial solutions . . . 30

**4 Algorithm** 35
4.1 Ship model . . . 35
4.2 Complying with charted objects . . . 37
4.3 Formulation as a graph search problem . . . 44
4.3.1 Grid-map . . . 48
4.4 Bidirectional search . . . 49
4.5 Bidirectional search with weather . . . 51
4.5.1 Weather data . . . 51
4.5.2 Bidirectional search with time dependent constraints . . . 53
4.6 Post-processing . . . 55

**5 Results and discussion** 57
5.1 Long distance routes . . . 58
5.2 Traffic separation schemes . . . 61
5.3 Monodirectional and bidirectional comparison . . . 64
5.4 Weather constraints . . . 67
5.5 Weather bidirectional comparison . . . 72
5.6 Simulation results . . . 79

**6 Limitations and improvements** 83
6.1 Computational time . . . 83
6.2 Variable time-step . . . 84
6.3 Placement inside channels . . . 84

6.4 Variable speed . . . . . 85
6.5 Tide . . . . . 86
6.6 Traffic routing objects . . . . . 86
6.7 Improving navigational maps for automation . . . . . 89

**7 Concluding remarks** **91**
7.1 Summary . . . . . 91
7.2 Conclusion . . . . . 92
7.3 Applications . . . . . 93

**A Feedback from Marine Technologies** **a**

**Bibliography** **c**

# Chapter 1

# Introduction

## 1.1 Background and motivation

A step towards the development of fully autonomous maritime shipping is to automatically plan routes without human intervention. An automatic path planner will use the available data before the ship sets sail to plan a safe and efficient route that will bring the ship from the current position to a given goal position. The goal position has to either be chosen by a human operator or some sort of higher level planning algorithm. The path produced will be produced off-line, and is therefore unable to avoid unmapped obstacles such as other ships. A local planner must be implemented that will find a way to avoid these obstacles. The ship should follow the COLREGS regulations for avoiding collisions at sea, which amongst others define on which side the ships should pass each other. An example of a COLREGS compliant local collision avoidance strategy is Johansen et al. (2016), this strategy is discussed in more detail in section 3.4. To be able to avoid obstacles, they must first be detected. Most larger ships can easily be detected as they are transmitting their current position using an automatic identification system (AIS) (Johansen et al., 2016). Ships without AIS can be detected by using radar, LIDAR, or cameras. An autonomous ship will need a path following controller which calculates a heading reference which will make the ship follow the current waypoints. An example of a path following controller is the Line Of Sight controller presented in chapter 2.3. In addition, the ship needs an autopilot that controls the actuators of the ship to achieved the reference heading given by

the path following controller. The path following controller and the autopilot will need a navigation scheme to figure out the ships position and heading. This can be achieved through the use of Inertial Navigation Systems (INS) combined with Global Navigation Satellite System (GNSS). This thesis will focus on the offline planner, and assume that the rest of the system is in place.

Automatically generating paths opens up the opportunity to make optimal paths with respect to for example fuel consumption. Automatically generated paths can incorporate more data than a human operator is able to utilize. Such data can be wind, wave, and current data. The planner can take this data into consideration and combine them with a ship model to make a path that minimizes energy consumption by sailing with the current and wind. An autonomous path planner can thereby be a step towards a more efficient and greener maritime industry. One example of this is Kobayashi et al. (2015) which is discussed in more detail in chapter 3.2.

This thesis will be an early step towards a fully automatic ship path planner, and will therefore focus on making a safe, rather than optimal, path. The planner will use real navigational charts that are currently used in the industry. The charts will be read using the NAVTOR SDK. The planner will find a path that avoids grounding with objects that are marked as dangerous and will avoid restricted areas. The navigational chart contains one-way lanes, where there is a required traffic direction. These will be followed to avoid the risk of collision with other ships. One limitation that will be prevalent thought this thesis is the long computational time that is needed to access the navigational charts. This problem will affect the choice of solution strategy, and how the algorithm is designed.

Another essential step for safe autonomous path planning is to avoid dangerous weather conditions. The fact that the weather conditions move over time makes avoiding them more challenging. An estimate for the weather up to 5 days into the future with a 3-hour resolution is available from a few different providers such as "European Centre for Medium Range Weather Forecasts" and the American based "Global Forecast System". Over large distances such as the Atlantic oceans, bad weather conditions can be avoided by planning the path around the storm. When this is not possible, then the ship should delay its departure until the weather is safe to sail.

## 1.2 Outline

This chapter will be ended by presenting a precise problem formulation, followed by a presentation of the contributions made by this thesis.

Chapter 2 presents theory that is needed in the following sections. Thereafter chapter 3 will present different strategies for path planning from the literature. How these strategies fit the problem at hand is discussed and the most promising strategy is chosen. A comparison of commercial path planners is done towards the end of the chapter.

Chapter 4 will present the full algorithm this thesis has produced. This chapter will gradually build up the algorithm. First, the simple ship model the algorithm will use is presented. Then the problem will be described as a graph search problem which makes it solvable with the chosen strategy. Thereafter the algorithm is developed to handle bidirectional search to speed up the planning process. Then the algorithm is developed further to support time dependant weather constraint. The chapter ends by presenting a post-processing scheme that makes a smoother low-resolution path that is easier for the path following controller to follow.

Chapter 5 will present the resulting paths that are generated with this algorithm. The chapter will present results that show how the algorithm works, and highlights advantages and disadvantages of different parts of the algorithm.

The next chapter 6 presents limitations of the algorithm and how it can be improved further. This section will also present problems with using the current navigational charts for automation, and outline changes that can be made to make the navigational charts more suitable for automation.

The final chapter, chapter 7, will first sum up how the algorithm works and what is achieved. Thereafter the thesis is concluded, and some idea on how the results in this thesis can be used is presented.

## 1.3 Problem Formulation

This project will develop an off-line path-planner for ships. The path-planner will use navigational charts to generate a safe path that can avoid time dependant weather constraints. Collision avoidance with dynamic and uncharted objects such as other ships, debris, drift-ice, or large mammals will be handled by a separate collision avoidance system that will not be handled in this thesis. Navigational charts will be read using the NAVTOR SDK.

The path planner should generate a set of waypoints which will be followed by a line of sight based path following controller. The line of sight principle is presented in chapter 2.3. This controller prefers a low-resolution path, where the distance between each waypoint is significantly longer than the ship length.

#### Objective

- The route must start and end at the given start and end coordinates.
- The route must ensure a safe journey that avoids collisions and follows the requirements set forth by the international maritime organization (IMO).
- The route should avoid sailing in dangerous weather conditions.

#### Elements used in traffic routing systems

IMO presents a set of elements that are used in traffic routing. The path should take these elements into consideration. All of the following elements are present in the NAVTOR SDK. The following list is taken directly from IMO (2018).

- Traffic separation scheme: a routing measure aimed at the separation of opposing streams of traffic by appropriate means and by the establishment of traffic lanes.
- Traffic lane: an area within defined limits in which one-way traffic is established. Natural obstacles, including those forming separation zones, may constitute a boundary.
- Separation zone or line: a zone or line separating traffic lanes in which ships are proceeding in opposite or nearly opposite directions; or separating a traffic lane

from the adjacent sea area; or separating traffic lanes designated for particular classes of ship proceeding in the same direction.

- Roundabout: a separation point or circular separation zone and a circular traffic lane within defined limits.
- Inshore traffic zone: a designated area between the landward boundary of a traffic separation scheme and the adjacent coast.
- Recommended route: a route of undefined width, for the convenience of ships in transit, which is often marked by centerline buoys.
- Deep-water route: a route within defined limits which has been accurately surveyed for clearance of sea bottom and submerged articles.
- Precautionary area: an area within defined limits where ships must navigate with particular caution and within which the direction of flow of traffic may be recommended.
- Area to be avoided: an area within defined limits in which either navigation is particularly hazardous or it is exceptionally important to avoid casualties and which should be avoided by all ships, or by certain classes of ships.

### 1.3.1 Constraints

- Uncharted obstacles will not be taken into account, and shall be handled by a separate collision avoidance system.
- The start and end points will be a safe distance from land, a separate module will handle docking and departure.
- The ship is expected to hold its cruise speed at all times. This assumption is used to estimate the position fof the ship relative to storms throughout the voyage.

## 1.4 Contributions

This thesis develops a path planning algorithm that is based on concepts from the hybrid A* which is presented in Dolgov et al. (2010). This source uses the hybrid A* algorithm as a basis for a full-fledged car path planner. This thesis will extend some concepts from the hybrid A* algorithm to work in a bidirectional search to speed up the process, and implement it to work with official navigational charts. The planner is developed to handle different objects that are present in navigational charts which are used to control the ship traffic. Furthermore, the bidirectional search is developed to handle time-varying constraints. This allows the path planner to avoid dangerous weather conditions which change throughout the voyage.

# Chapter 2

# Theory

## 2.1 Coordinate transformation

The following coordinate transformations assume a spherical earth, and are all taken from Veness (2017). Assuming a spherical earth can lead to distance calculation errors up to 0.55% (Veness, 2017). This error will be negligible relative to other design choices that lead to suboptimalities, such as discretization of the ship sate. The index $i$ takes the value 1 for the initial point, and value 2 for the end-point.

$\mu_i$ - latitude
$l_i$ - longitude
$\theta_i$ - heading relative to north
$p_i = (\mu_i, l_i)$ - a point
$\Delta\mu = \mu_2 - \mu_1$ - change in latitude
$\Delta l = l_2 - l_1$ - change in longitude
$R = 6,371 km$ - mean earth radius

### 2.1.1 Great-circle distance

$d = Great_Circle_Distance(p1, p2)$

This function accepts two points, $p_1$ and $p_2$, given in latitude/longitude. The function returns the great circle distance, $d$, between the two points. A great circle is the shortest path between two points on the surface of a sphere.

$$a = \sin^2\left(\frac{\Delta\mu}{2}\right) + \cos(\mu_1)\cos(\mu_2)\sin^2\left(\frac{\Delta l}{2}\right) \tag{2.1}$$

$$c = 2\,\text{atan2}(\sqrt{a}, \sqrt{1-a}) \tag{2.2}$$

$$d = Rc \tag{2.3}$$

The atan2$(y, x)$ function is a modification of the arctan$(y/x)$ function which places the solution in the correct quadrant of the four quadrants.

### 2.1.2 Translation

$p_2$ = *Translate($p_1$,$\theta$,$d$)*

This function accepts a point, $p_1$, a heading clockwise from north, $\theta$, and a distance in meter, $d$. It returns the position $p_2$ that is reached by starting out with the heading $\theta$ at $p_1$ and following a great circle for the distance $d$.

$$\Delta s = \frac{d}{R} \tag{2.4}$$

$$\mu_2 = \arcsin(\sin(\mu_1)\cos(\Delta s) + \cos(\mu_1)\sin(\Delta s)\cos(\theta)) \tag{2.5}$$

$$l_2 = l_1 + \arctan 2(\cos(\mu_1)\sin(\Delta s)\sin(\theta), \cos(\Delta s) - \sin(\mu_1)\sin(\mu_2)) \tag{2.6}$$

$$p_2 = (\mu_2, l_2) \tag{2.7}$$

### 2.1.3 Initial heading

$\theta_1$ = *Initial_heading($p_1$,$p_2$)*

This function accepts two points, $p_1$ and $p_2$, and returns the heading, $\theta_1$ which is needed at $p_1$ to reach $p_2$ while following a great circle.

$$\theta_1 = \text{atan2}(sin(\Delta l)cos(\mu_2), cos(\mu_1)sin(\mu_2) - sin(\mu_1)cos(\mu_2)cos(\Delta l)) \tag{2.8}$$

## 2.2 Nonholonomic system

A system where the state depends on the order of actions that were done to reach it is a nonholonomic system. A tank is a simple example of a nonholonomic system. First driving 1 minute with 80% of the max rotation rate on the left track and 100% on the right track, followed up by driving 1 minute with 100% on the left and 80% on the right, leaves the tank in a different state than if the two 1 minute sections were swapped. Both tracks rotate the same amount of times in the two cases, but the order of the rotations define the end position of the tank. A holonomic system on the other hand will end in the same state with the same set of actions, unaffected by the order of the actions. Example of a nonholonomic system is a platform suspended in water that can freely move in the surge and sway direction. Spinning the surge propeller 1 minute at 100% thrust, followed by spinning the sway propeller 1 minute with 100% thrust places the platform in the same state as if the order was switched.

Path planning with nonholonomic systems is generally harder than for holonomic systems. For a nonholonomic system, the path the system followed to reach a point affects the state of the system. Example of such systems can be cars and ships, where the heading at any position is defined by the path the vehicle followed to that position. Finding a path to a specified point for a holonomic system, for example a queen moving freely on a chess board, is substantially easier as only the current and end position of the queen needs to be taken into account.

A more precise definition of a holonomic system can be found in Spong et al. (2005, p. 362)

## 2.3 Line Of Sight guidance

Line Of Sight (LOS) guidance is a concept used for deciding a heading the ship should have to follow a predefined path. The path should consist of waypoints with straight lines between the waypoints. The ship is assumed to have a heading autopilot that steers the ship heading towards the heading reference calculated by the LOS guidance scheme. One way of doing line of sight guidance is to control the ship directly towards the current waypoint. This is in many cases not a good approach as the ship will not attempt to follow the designed straight line path. The ship might then hit obstacles

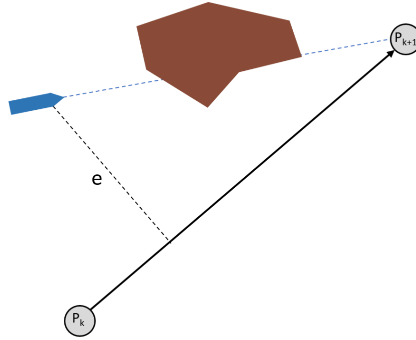
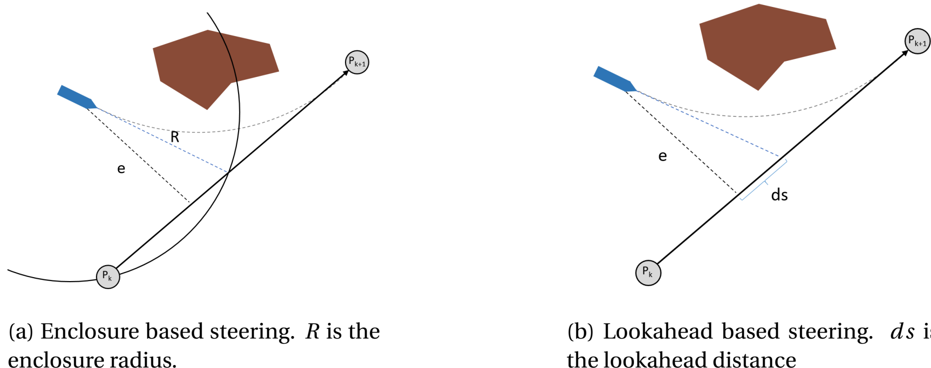



Figure 2.1: LOS guidance directly towards the current waypoint. *e* is the cross track error.

that the path was designed to avoid, this is illustrated in figure 2.1. Two alternative strategies that avoid this problem are as follows:

**Enclosure based steering:** Figure 2.2a illustrates this concept. A circle with a constant radius R is made around the ship. This circle may cross the planned straight lined path at two points. The ship should then head towards the crossing point that is closest to the current waypoint. When the ship has a large cross track error (distance between the ship and the planned straight line path), this strategy will make the ship move towards a close point on the path. When the cross track error is small the ship will move towards a point further up the path. The ship will thereby start to move quickly towards the path, and then gradually change over to following the path when the cross track error gets smaller. Enclosure-based steering only works when the ship is within the radius R of the path.

**Lookahead based steering:** Figure 2.2b illustrates this concept. First the closest point on the straight line path is found, then the point a distance $ds$ up the path is found. The ship will head towards this point. This strategy is similar to enclosure based steering, but the distance $ds$ up the path does not change with the cross track error. When the cross track error is large, the distance towards the path will dominate the distance $ds$ along the path. The ship will then mainly move towards the path. When the cross track error is small, the distance, $ds$, along the path will dominate, and




(a) Enclosure based steering. $R$ is the enclosure radius.

(b) Lookahead based steering. $ds$ is the lookahead distance

Figure 2.2: Different line of sight steering strategies.

the ship will mainly move along the path. This strategy is computationally simpler than enclosure based steering and works for all distances.

Choosing a small $R$ or $ds$ makes the ship move quicker towards the path, choosing larger values can make a smoother path with less overshoot. The parameters should be chosen in such a manner that the ship quickly reaches the path, without overshooting it.

Another parameter that must be decided upon is when to change to the next waypoint. Changing the waypoint when the current waypoint is reached leads to the ship not attempt to turn towards the next waypoint before the current waypoint is hit. The ship will then substantially overshoot the planned path to the next waypoint, which can be dangerous. One alternative strategy of changing the waypoint is to change when the ship is within some radius of the current waypoint. The ship will then start to turn towards the next waypoint earlier, and thereby manage to follow the next path segment closer. This radius should be designed based on the turning radius of the ship, such that the cross track error along the path is minimized. A limitation with this strategy is that if the ship has a cross track error larger than the switching radius of the path, then the waypoint will not be switched until the ship manages to get closer to the path. This may lead to the ship sailing past the current waypoint, and having to circle back towards it to get close enough to change to the next waypoint. A strategy that avoids this problem is to instead consider the distance along the planned path to the waypoint, and change when this distance is small enough. The ship will

then be able to change to the next waypoint no matter how large the cross track error is. Fossen (2011, p. 264) suggest changing the waypoint when the distance measure to the current waypoint is less than two times the ship length.

More information on line of sight guidance can be found in Fossen (2011, ch. 10).

## 2.4 Problem solving by graph search

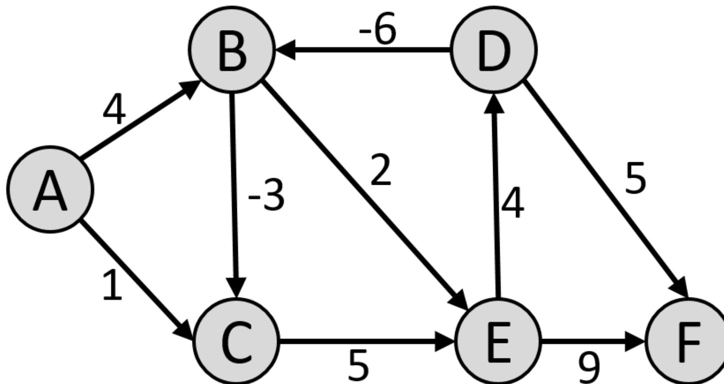



Figure 2.3: Example of a weighted directed graph. The circles are vertices, and the arrows directed edges. The number on each edge is the weight of that edge.

Some planning problems can be solved by describing the problem as a graph, and then searching through the graph for a solution. A graph is a set of vertices, $V$, and edges, $E$, that connect the vertices. A graph is said to be directed if the edges have a direction. Figure 2.3 show an example of a graph. Planning problems, where the goal is to find a set of actions that will bring an agent from a given start state,$s$, to a goal state, $t$, can be reformulated as a graph if the conditions presented in Russell and Norvig (2014, p. 66) are fulfilled. The planning problem has to have a finite set of discrete actions that can be performed at each discrete state. Each action must deterministically move an agent from one state to another. The agent must also be able to know which state it is in (the current state is observable), and which state which will be reached by each action. This general case of problems can be transformed into a graph search problem by setting all states as vertices in the

graph, and the actions that connect two states as the edge between the corresponding vertices. If the task is to find the best set of actions that bring the system from the start to the goal state (for example minimizing the number of actions) then there has to be associated a cost with each action. The sum of cost for the actions should be minimized. This cost will be represented as a weight on each edge in the graph, this will produce a weighted graph. A graph is said to be connected if there is a path between any two vertices. A connected graph without any cycles is called a tree. Figure 2.4 shows an example of a tree.

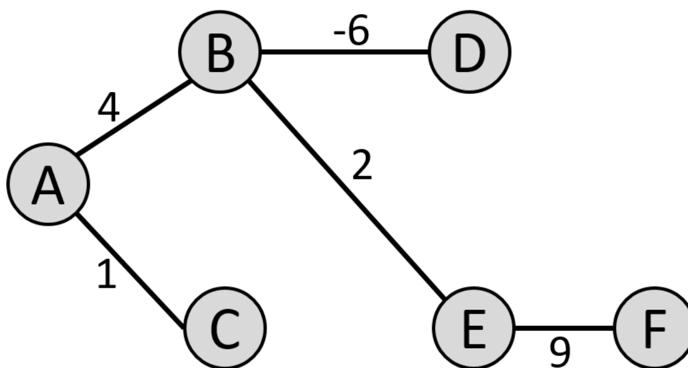



Figure 2.4: Example of a tree, a graph without any cycles.

Even though the constraints on the graph search problems are strict, there are still a lot of problems that can either be solved to optimality or be approximated by using graph search. Some problems will naturally have a discrete set of actions at each state, for example finding the best route to follow to get from one city to another. Here all intersections where the road splits can be seen as a unique state, and deciding what road to take will be a discreet choice. Other problems, such as maneuvering at sea does not have a natural set of discrete states and actions. These problems can not be solved to optimally by graph search problems, as the ship can choose between a continuous set of possible sailing directions and positions. The solution can still be approximated by discretizing the problem. If we limit the direction the ship can sail into a finite set of headings, and then assume that the ship will follow that heading for a given time interval, then this problem will also fulfill the demand of discreet

actions and states. The state will be the position (and possibly heading) of the ship at the end of each time-step, and the actions will be the different possible discreet heading directions. This is just one of many ways of discretizing that particular problem. Observably can be handled by a navigational scheme, for example using GNSS. Making the outcome of each action deterministic will be handled by a lower level controller that ensures that the planned state will be reached.

### 2.4.1 Dijkstra's algorithm

Dijkstra's algorithm can be used to find the shortest path on a weighted directed graph with nonnegative weights. The algorithm was first presented by Edsger Dijkstra in Dijkstra (1959). The version of the algorithm presented here is from Cormen et al. (2009, p. 658)

Dijkstra's algorithm assigns a value g(v) to all vertices v. This value will represent the shortest found path to vertex v. This value is initialized to be infinite at all nodes except for the start vertex, where it will be 0. For each iteration of the algorithm the vertex, $u$, with the lowest g($u$) value will be explored. All vertices, $v$, that can be directly reached from $u$ over the edge $(u, v)$, are found. If the new path going through $u$ over the edge $(u, v)$ is shorter than the previous found shortest path g(v) then a new shorter path to $v$ is found, and g(v) is updated. That is:

$$g(v) \leftarrow min\{g(v), g(u) + w(u, v)\} \tag{2.9}$$

Where $w(u, v)$ is the weight on the edge from u to v. After a vertex is explored it is placed in the "closed" list over all vertices that are handled. Vertices that are already explored cannot be improved upon as all weights are positive, and they were the lowest weighted unexplored vertex when they were explored. All other vertices that are still not in the "closed" list will have a higher $g(v)$ value than any explored vertex. When the end vertex, $t$, is explored then the value $g(t)$ will be the shortest distance from $s$ to $t$ in the graph. To also find the shortest path connecting $s$ to $t$, each vertex will contain a pointer to the vertex that gave them their current g-value. The optimal path can then be found by starting at the end vertex, $t$, and following the pointers back to the start vertex, $s$.

The proof of optimally of the Dijkstra's algorithm is presented in Cormen et al. (2009, p. 659).

### 2.4.2 A*

The A* algorithm is a faster version of the Dijkstra's algorithm that uses domain-specific knowledge in the form of a heuristic function $h(v)$ to speed up the search process. A* was first developed by Peter Hart, Nils Nilsson, and Bertram Raphael, and presented in Hart et al. (1968). The version of the A* algorithm that is presented here is from Russell and Norvig (2014, p. 95)

The heuristic $h(v)$ gives an optimistic estimate of the distance between the vertex $v$ to the goal state. The heuristic has to be admissible and consistent to ensure that the A* algorithm will find the shortest path to the goal state. An admissible heuristic is one that is always shorter or equal to the true shortest path from a given vertex to the end vertex. That is $h(u) \leq d^*(u,t)$, where $d^*(u,t)$ is the true shortest path from $u$ to the end vertex $t$. A consistent heuristic requires that the heuristic estimate is lower or equal to the heuristic estimate of any neighboring vertex plus the actual cost of reaching that vertex. That is: $h(u) \leq h(v) + d^*(u,v)$. For path-planning in Euclidean space, a possible heuristic is the Euclidean distance from the vertex to the goal. The closer the heuristic is to the actual solution the fewer unnecessary nodes does A* need to explore.

The A* algorithm introduces a value $f(v) = g(v) + h(v)$ which tells how promising a vertex is. $f(v)$ is equal to the distance that had to be traveled to reach $v$, $g(v)$, plus the optimistic estimate of the distance that is left to reach the goal state $t$, $h(v)$. The A* algorithm works in the same way as the Dijkstra's algorithm with the change that the most promising unexplored vertex, the one with the lowest $f(v)$ value, is explored first In Dijkstra's algorithm the closest unexplored vertex, the one with the lowest $g(v)$, was explored first. This change lets the A* algorithm search through promising nodes first while still guaranteeing that the shortest path is found.

A proof of optimality of the A* algorithm is presented in Russell and Norvig (2014, p.97).

### 2.4.3 Bidirectional search algorithm

The area explored by Dijkstra's algorithm can be portrayed as a gradually expanding circle around the start vertex. This is since Dijkstra's algorithm always explores the closest, lowest g(v) value, vertex first. The algorithm then finds a solution when the circle grows large enough to encompass the end vertex. Bidirectional search

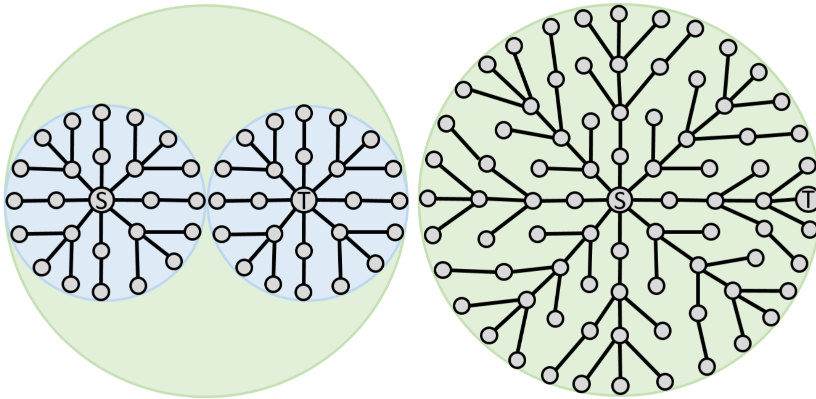



Figure 2.5: Two search trees with half the diameter are much smaller than one with the full diameter. In this small example the bidirectional search has 50 vertices, while the monodirectional search has 81.

algorithms are inspired by the fact that a large circle with diameter $d$ has a much larger area than two smaller circles that cover half the diameter each, as $b^d >> 2b^{d/2}$ (Russell and Norvig, 2014, p. 92). This concept is presented visually in figure 2.5. In the same manner two search trees, one starting from the start vertex, and one from the end vertex, will cover a much smaller area than only having one starting from the start vertex. Bidirectional search can therefore drastically reduce the search space for larger problems. To be able to use a bidirectional search algorithm a method for finding the intersection of the two search trees has to be developed. The first intersection found will not necessarily lead to the shortest path, so some additional search may be required to find the best intersection (Russell and Norvig, 2014, p. 92). A bidirectional search algorithm also requires the ability to search backwards. This can in some instances be as easy as turning the edges on the graph, but will in other instances, such as when the graph is generated during the search, be more problematic. Pijls and Post (2009) successfully demonstrates that bidirectional search can be used to speed up A* as well.

### 2.4.4 Time dependent bidirectional search

Finding the shortest path in a graph with time-dependent edges and weights can be problematic in a bidirectional search. One example of time-dependent graphs are

roadmaps that take traffic jams into account. Another example is the problem this thesis tries to tackle, where the ship should avoid time dependant weather conditions. Here the weather will move as the ship follows the path, so when the ship reaches a specific point matters. The forward searching part of the bidirectional search can easily figure out when it is at a specific location, given that it knows when it started and that it has a good estimate of how long it takes to traverse each edge. The reverse search is a lot more problematic, as it has no way of knowing when the ship will arrive at the end point before the path is found. It is therefore impossible for the backward-search to take the time-dependent effect into account.

Nannicini et al. (2012) found a way of utilizing bidirectional search in the case of a time-dependent roadmap. The time it takes to traverse an edge will depend on the traffic on the road corresponding to that edge, which changes throughout the day. This was done by replacing the time-dependent arc cost on the backward search, with its corresponding lower bound value. The two search directions will grow into each other until all vertexes on the frontier of the backward search have lower f(v)=g(v)+h(v) value than actual (time-dependent) cost of following the shortest path through the first vertex that was explored by both search directions. The forward search will for the rest of the search be limited to only explore nodes that the backward search has already explored. The program terminates when the forward search finds the goal vertex. The backward search is thereby used to limit the search space for the forward search.

# Chapter 3

# Related work

There are many different ways of solving path planning problems, which have different advantages and disadvantages. This chapter will present different strategies that have previously been developed in the literature. The chapter will then discuss how well the different strategies fit the current problem. Thereafter a strategy for local collision avoidance is presented. The chapter is ended by comparing three different commercial ship path planners.

## 3.1 Combinatorial search

### 3.1.1 Cell decomposition

A direct way to use an A* algorithm to find the shortest path on a map is to discretize the map into a grid. Each cell in the grid will be marked as safe if it contains no

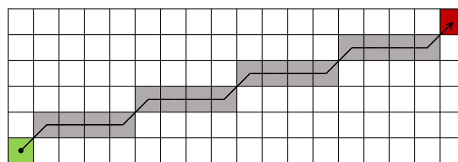


Figure 3.1: A path where only transition to neighbouring cells are accepted.

obstacles, else it will be marked as dangerous. Each safe cell will be a vertex in the graph, and the position of the vertex will be the middle of the cell. The edges that connect the vertices will define which transitions that can be done on the map. If an agent can transition from one cell to any one of the eight cells around it, then the path will be limited to heading changes with a 45° interval. This is shown in figure 3.1. Ueland et al. (2017) present a path planner that accepts more possible heading changes by not only allowing transitions to the 8 neighbor cells that are next to the current cell, but also allowing transitions to cells that are up to 4 cells away. Any cell that is in a direct line from a previous allowed heading is not included, so, for example, are multiple cells to the left not included as the same path will be reached by going one cell to the left multiple times. The paper combines this path planner with a SLAM mapping process and implements it in the case of an oceangoing vessel that will navigate a harbor. The solution is shown to work experimentally with a small holonomic vessel, that can move freely in the surge and sway direction.

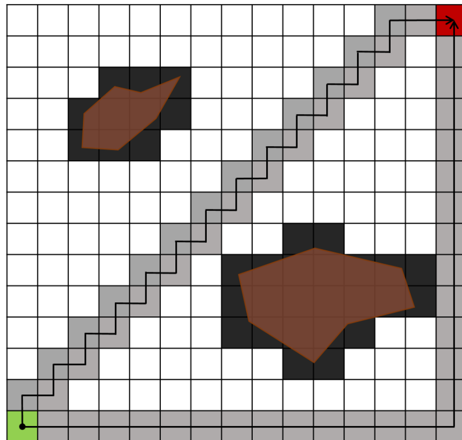


Figure 3.2: The problem of multiple equally long paths with cell decomposition.

A problem with grid-map base path planners is that the search-space is large, which could lead to a long computational time when used over a large area. The search space of the planner presented by Ueland et al. (2017) can be reduced a bit by limiting the number of transitions allowed. This will make the path less optimal as it will have fewer possible heading directions that can be followed. The lowest number

of heading directions is to have 4 new possible states per state, which would limit the path to straight lines going up-down or right-left. Another strategy for reducing the size of the problem is to make the cells larger, such that there will be fewer of them. The size of the cells has to be chosen smaller than half the size of the smallest safe water area that should be sailed. This is to ensure that at least one cell will be without a part of an obstacle in narrow areas.

Another problem that arises with this method is best demonstrated in the case of allowing 4 transitions from each cell: up, down, right, and left. The resulting path will now only consist of straight lines. The problem with this is demonstrated in figure 3.2. The path that goes all the distance to the right, and then up, is equally long as the one going zigzag diagonal up to the right. If the paths were to be smoothed out before they are followed then the diagonal path would be substantially shorter than the path going to the right then up. Using an A* algorithm with a heuristic that uses Euclidean distance, will make the algorithm explore the zigzag path first. But as the two paths are equally long, both can be chosen as the optimal solution by the path planning algorithm. This problem persists when increasing the number of possible transitions, but the maximal error it can lead to decreases.

### 3.1.2 Probabilistic roadmap

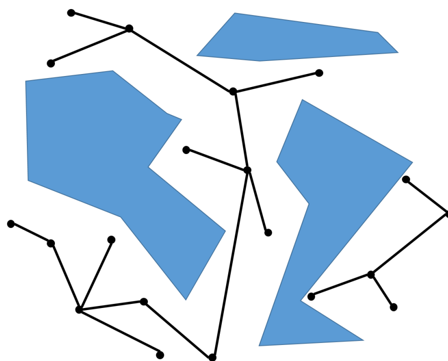


Figure 3.3: Example of a probabilistic roadmap. The map contains two unconnected components.

A way of substantially reducing the search space is to drastically reduce the number of vertices in the graph. The concept of probabilistic roadmap (PRM) does

this by randomly placing vertices around the map, instead of placing one at each grid. This concept was first presented by Kavraki et al. (1996). PRM consists of two stages, a learning stage where a graph of legal transitions is constructed, and a query phase where this path is used to find a legal path between two arbitrary placed points. The learning phase will repeatedly place a random point, denoted as $u$, and then finds the closest $N$ points in the graph that are within a maximum distance $c$, this set is called $N_c$. All the points will be tested, in growing order of distance, to see if there is a legal path between $u$ and the current point in in $N_c$, called $v$,. If there is a legal path, then the edge (u,v) is added to the graph. Edges that would lead to cycles are not added, as this algorithm wants a cycle-free graph. Having a cycle-free graph makes finding the path between two points quick, as there is only one option. Cycle-free graphs may lead to unnecessarily long paths. Kavraki et al. (1996) proposes to solve this problem by smoothing the path after it is generated. Figure 3.3 shows an example of a probabilistic roadmap.

After the graph becomes significantly dens, PRM will go over to the growing phase, where it tries to connect disconnected components of the graph. There might be difficult parts of the map where the planner failed to find a legal path by simply placing random points. PRM will attempt to bridge these gaps by actively placing points in these areas. These difficult areas will be selected based on different metrics. One proposed metric will give high priority to areas with few points, as these areas are mainly covered by obstacles and therefore can contain narrow passageways between two connected components. Another proposed metric is to give areas where two disconnected component are close a higher priority, as these areas probably are close to where they could be connected.

The next stage of PRM is the query phase, where the graph is used to find a path between two points. PRM will first try to connect the start and end point to the previously generated graph. This is done by trying to connect the start and end points to the closest nodes in order of rising distance. If they are unable to connect, then a random walk strategy is used on the start/end point until they are. The start and end point will have to be connected to the same connected component of the graph. If the algorithm is unable to do this then there is no path between the start and end point with the current graph. The graph can then be expanded further in hopes of finding a path.

The main advantage of PRM is its speed. Making the graph, which is the time-consuming part of the algorithm, is done off-line before the path planner is used. The query stage is able to quickly find a path between two given points if one exists. The main drawback of the algorithm is that it does not attempt to find an optimal path, as the graph contains at most only one path between any two points. The algorithm tries to make the path better by smoothing the finished path which is done by cutting corners and removing unnecessary points.

PRM can get closer to the optimal solution by allowing connections that generate cycles (Rantanen, 2014). New vertices will then be connected to all close vertices it can be connected to. Finding a path from the start to the goal state will then require some sort of graph search algorithm, for example Dijkstra's algorithm or A*. Allowing the PRM to generate cycles will make the algorithm slower, but will allow for better solutions.

### 3.1.3 Voronoi diagram

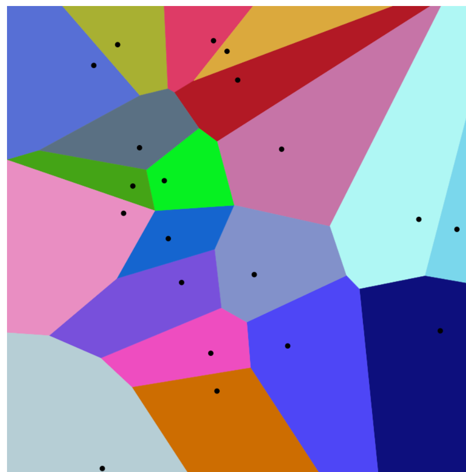


Figure 3.4: A Voronoi diagram. The small black dots are the seeds. The colored regions around the dots is the area closest to that seed.
Credits: Balu Ertl 2015

Another strategy for generating a road map graph is to use the concept of Voronoi diagrams. A Voronoi diagram accepts a set of points called seeds, and partitions

the space around the seeds into sections. Each section contains all the points that are closer to the seed of that section than any other seed. Figure 3.1.3 shows an example of a Voronoi diagram. This concept can be used in path planning by using the obstacles as seeds and letting the borders in-between the sections be edges in a graph. The graph will then contain edges between all obstacles, and the edges will be as far away from the two closest obstacles as possible. Generating a path that is in the middle of obstacles is generally safer than having a path that just misses one of the obstacles.

Candeloro et al. (2017) presents a Voronoi diagram based path planner for underactuated marine vessels. The path planner uses a map section where obstacles and depth contours are represented as polygons. Each edge point of the polygons is used as seeds for the Voronoi diagram. Additional random points along the map border where there is water are added as seeds. The entire sailable area is thereby enclosed by seeds that are used in the Voronoi diagram. A Voronoi diagram is then generated in the sailable area, and a graph consisting of the edges between the sections is generated. A modification of the Dijkstra's algorithm is then used to find the k-shortest paths along the Voronoi diagram edges. The shortest one that fulfills the clearance constraint is then chosen. All waypoints that can safely be removed without breaking a clearance constraint are then removed. Lastly, a smooth path is made between the waypoints using pieces of Fermat's spiral. Candeloro et al. (2017) also presents a replanning strategy to handle uncharted obstacles.

The Voronoi diagram based strategies two main advantages are that it is quick, as the search space in the Voronoi diagram is small, and it places the path in the middle of canals which makes the path safer. This method is guaranteed to find a path if there is one. This strategy is mainly applicable when sailing in areas where there is land on all or most of the sides. If one or more of the sides of the path is completely open water, then the map section would highly influence what path is chosen, as the path will be placed in the middle between the land area and where the map arbitrarily ends.

### 3.1.4 Visibility graphs

Another strategy of drastically reducing the search space is to utilize the fact that the shortest distance between two points is a straight line. This assumption only holds in Euclidean space, which is a good enough assumption in local areas on the earth

for a path planner. The shortest path between two points will then always consists of straight paths between the edges of obstacles. Shah and Gupta (2016) has developed a visibility graph based path planner for a surface vehicle navigating an area with stationary obstacles, such as land obstacles. The algorithm uses the edges of the obstacles as possible waypoints and limits the search by only taking edges that can be seen from the current ship position into account. This paper also develops a better heuristic than the Euclidean distance to the target. This heuristic first finds all obstacles that intersect the straight line to the goal position. Then the furthest edge point in any of the obstacles obstructing this path is found on the right and on the left side of the path. The closest of the left and right distance will be taken into account when calculating the heuristic, as the ship has to at least sail around this obstacle.

Visibility graphs will in contrast to Voronoi diagrams produce an optimal path between two points, but this path will be without safety margins. A safety margin can be introduced by marking areas that are closer to land than the margin as dangerous areas. The visibility graph will not take the ship dynamics into consideration, and can therefore make arbitrary sharp turns.

### 3.1.5 Rapidly-exploring random trees

Lavalle (1998) develops a path planner for nonholonomic systems called the Rapidly-Exploring random trees (RRT). The algorithm constructs a tree with states and the inputs needed for the state transitions. The algorithm consists of 4 steps. First, a random state, $x_r$ within the set of possible states is chosen. Then the already explored state closest to $x_r$ is found, this state is called $x_c$. Thereafter the input, $u$, that brings $x_c$ as close as possible to $x_r$ after being applied in $\Delta t$ time is found. Only states that do not result in collisions can be chosen, this new state will be denoted as $x_n$. $x_n$ is then added as a child vertex to $x_c$, and the corresponding input $u$ is recorded. This results in an algorithm that is biased towards exploring unexplored portions of the state space. Choosing a random input on a random state in the tree, instead of finding the input that brings the tree closest to a random position, will lead to a bias towards already explored areas (Lavalle, 1998).

RRT can be used to find a kinematically feasible path for non-holonomic systems, such as cars, or more importantly ships sailing at cruise speed. The path is not guaranteed to be optimal, but it is guaranteed to be physically feasible for the system to follow the path.

### 3.1.6 Hybrid A*

Dolgov et al. (2010) presents a full-fledged path planner for a vehicle navigating a parking lot that is close to optimal and kinetically feasible for the vehicle to follow. This paper develops an algorithm called Hybrid A* that takes concepts from the rapidly-exploring random tree (RRT) to circumvent the problems with the cell decomposition method. In the same way as in the cell decomposition method, the map is decomposed into a grid-map of cells. Each cell can contain one state, or vertex in the graph, but the state can have any position inside the cell, not just the middle. The vertex can also contain states which are not present on the map, such as heading. The graph starts out by just containing the start vertex, but will gradually grow out as a tree. RRT avoided the problem that simply choosing random actions at each state would bias the search toward the already searched area, by first finding a random position and then finding the action that brought the next state closest to that position. Hybrid A* uses the concepts from A* to bias the search towards the goal and using the cells to limit the number of vertices in the explored area to avoid this problem. Hybrid A* defines a set of legal actions at each state: maximal turn to the left, keeping straight, and maximal turn to the right. The algorithm will choose the most promising vertex, the one with the lowest transition plus heuristic cost. This state is exposed to the three different possible actions for a given time, $\Delta t$, to generate three new successor vertices. The position of the new vertices is calculated and saved in the cell in the grid-map that encompasses the position of the new vertex. If the cell already holds another vertex, then the least promising vertex will be deleted. By doing this the algorithm can use continuous positions, and arbitrary heading changes. This will circumvent the problems that arise when the path can only transition from the middle of one cell to another, which were presented in chapter 3.1.1. This method also incorporates a model of the vehicle, which makes the path kinetically feasible. The paper suggests two heuristics where the maximum value of the two is used. The first is called nonholonomic-without-obstacles, here the nonholonomic kinematic model of the car is used on an environment without any obstacles. A lower bound on the distance from any point to the goal state can then be calculated and used as a heuristic. The other heuristic is called holonomic-with-constraint, this heuristic ignores the nonholonomic property of the vehicle and finds the shortest route to the end location with the obstacles in place.

The Hybrid A* algorithm will return a finished path when it manages to make an analytical solution from one of the vertices in the graph to the goal state. These solutions are made using a Reed-Shepp path. The resulting path will be a winding path consisting of sharp right and left turns, which has to be smoothed out before it can be used. The smoothing is done by optimizing the found path with respect to an objective function which rewards minimizing turning and keeping a safe distance to obstacles.

The paper develops this algorithm further to work in a semi-structured environment. Here the correct driving lanes of the parking lot are decoded as a graph the car tries to follow whenever possible. When the lanes have to be broken, for example due to an obstacle in the way, the method above is used to quickly and safely circumvent the problem and get back to the lane.

## 3.2 Optimization based path planners

A strategy for finding an optimal path is to gradually tweak an existing path in such a manner that a given objective function is minimized. One example of this is presented in Kobayashi et al. (2015) where they find an optimal path over the Pacific ocean with respect to different objectives. The benchmark and initial path for the optimization is the great circle path from San Francisco to Tokyo. The paper uses a ship model that incorporates wind and current forces. This model is used to find two different routes between San Fransisco and Tokyo. One of the routes minimizes the fuel-oil consumption and the other minimizes the energy-efficiency operational indicator. The paper uses Powell's method to find the optimal placement of N control points that define a N-1 order Bézier curve that constitutes the path.

Optimization based path planning strategies find a path that approaches the optimal path with respect to some objective function. They work with continuous states, no discretization of state is required. The downside of optimization based approaches is that they are generally slower than road-map methods (Bitar, 2017). Optimization base methods will often need an initial feasible path which it improves towards a local optima.

## 3.3 Applicability to this thesis

The goal of this thesis is to tackle the problem of finding a safe but efficient route for a ship to follow to reach a given goal state. The ship has to comply with charted traffic routing objects, such as traffic separation zones. These are marked areas on the chart where the traffic is only allowed to go in a given direction. The main limitation of the project is the long computational time needed to access the navigational chart. Having a fast path planner is not essential as the path is being made off-line. On the other hand, using multiple hours to produce a path is unwanted as this will be too cumbersome for a human operator to uses as an assistance tool for path planning. Minimizing the number of times the chart is accessed is therefore important to keep the computational time reasonable. Using a cell decomposition method will therefore probably be too slow, as it explores too many of the cells.

Using PRM would reduce the number of vertices that need to be explored compared to cell decomposition. This method is mostly applicable in cases when the graph can be made once and then utilized many times. This is probably not the case in the setting of this thesis, as each mission will get new mission specific maps, which requires that a new graph is made each time. PRM will also explore a lot of unnecessary vertices, as areas far away from the final path are also checked.

Voronoi diagrams would work well with the Navtor API. The API returns the obstacles in a region as polygons, which can directly be used to make the Voronoi diagram. This strategy would access large portions of the charts at the time, making the number of access low, which is the main problem. The drawback of using a Voronoi diagram is that it only works in areas that are almost completely enclosed by land. This works well in Norwegian fjords but does not work well in larger open water areas. This thesis attempts to develop a path planning strategy that can work globally, the Voronoi diagram strategy is therefore not applicable in this thesis. The Voronoi diagram does not have an obvious extension to make it work with traffic separation schemes.

Visibility graphs would make the path more optimal than in the case of the Voronoi diagram. The algorithm can be developed to use great arch segments instead of straight lines which would make the visibility path find the shortest path over large areas as well. The visibility graph will make paths as close to edges as possible and does not incorporate the ship dynamics. These limitations break with the objectives of the algorithm which is to make a safe path. The ship dynamics could possibly

be taken into account by smoothing the path, and then finding new paths until a kinematically feasible path is found.

Rapidly-exploring Random trees will incorporate the ships dynamics which will be preferable. RRT can also easily implement the support for one-way lanes, by checking if an edge has the correct sailing direction. If this is not the case, then the edge will be deleted. Weather can similarly be implemented as the arrival time of a vertex can be calculated, and the position of the weather at that time point used to see if the edge breaks a weather constraint. As RRT is unguided, it will be way too slow and can end up with unnecessary long paths.

Of the proposed algorithms Hybrid A* seems the most promising, as it keeps the advantages of RRT while being guided which can reduce the number of times the chart is accessed and approximates the optimal path. This strategy will require substantially many more chart access than the Voronoi diagram method and the Visibility graph method, but can easily be extended to work with weather and traffic constraints. This method is also the most promising for further development, as it is possible to extend it to use a complicated ship model which uses weather and currents to find optimal routes.

Optimization based methods can not be directly implemented as they require an initial feasible path. This would require a path from the start state to the end state that does not break any constraints. They can therefor rather be used to smooth out a path found by one of the other methods presented in this chapter. One example of how optimization methods can be used for smoothing is presented in Dolgov et al. (2010). The computational complexity of optimization based methods which require accessing the chart many times would be problematic for the current problem.

## 3.4 Local collision avoidance

The algorithms presented thus fare are mainly for planning a path before the trip starts. These paths use some sort of map to make a safe path with respect to the mapped obstacles. A collision avoidance system is needed to avoid unmapped obstacles, such as other vehicles. For the case of maritime traffic, the COLREGS rules for avoiding collision at sea should be followed. These rules define amongst other which side ships should pass on each other to avoid both of the ships turning to the same side.

Johansen et al. (2016) presents a COLREGS compliant ship collision avoidance system (CAS) that scales well with the number of obstacles. The algorithm assumes a prior path that the ship attempts to follow using a line of sight based control law. The algorithm also assumes that the position of all obstacles is in some way measured. The CAS produces a course angel offset and change in propulsion that will attempt to avoid collisions while complying with the COLREGS regulations. The offset is found by testing a finite set of different control behaviors consisting of different steering angles and propulsion speeds. Following the path without any offset is always chosen if this action is sufficiently safe. If this is not the case, then the safest combination of course angle offset and change in propulsion speed is chosen. The ship will then follow this change for a fixed time interval, until CAS has recalculated a new course angle offset and change in propulsion speed. Reevaluating the offsets at a regular time interval makes the algorithm robust to unpredicted behavior from the other vehicles, such as if the vehicle does not follow COLREGS. The course angle offset is always relative to the course angle given by the line of sight guidance law, not the current course of the ship. This makes the ship automatically return to the original path when the obstacle is avoided. The paper ends with a simulation study that shows the effectiveness of this algorithm in cases with up to 8 obstacles.

## 3.5 Commercial solutions

This section will briefly compare three commercial ship path planners. The planners will be tested to see if they manage to make efficient routes that avoid grounding, if they manage to follow traffic separation schemes, and if they manage to make paths from one fjord in Norway to another. All of the applications had computational times of a few seconds.

**MarineTraffic - voyage planner:** (MarineTraffic, 2018) This planner uses a database of routes from millions of past voyages to find a path between two points. This strategy should make paths that follow traffic separation schemes, as most recorded routes follow these schemes. The finished route manage to use traffic separation schemes, but will sometimes traverse them in the wrong direction. Figure 3.5c and 3.5d shows that the path uses the same separation line of the Dover strait when traversing the straight northeastwards and southwestwards. The paths are also far from optimal, containing complicated detours, as can be seen in figure 3.5a. The

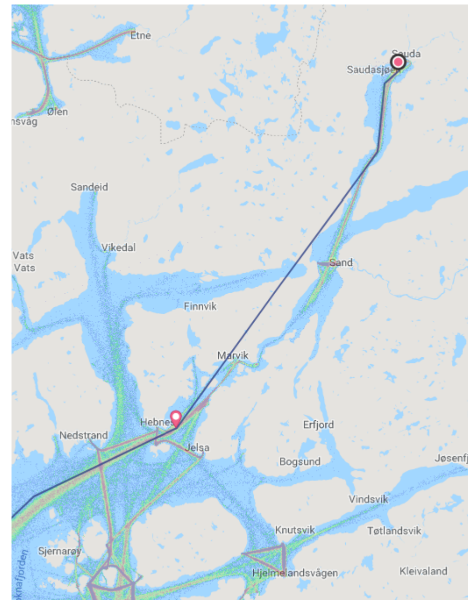
path is not guaranteed to avoid grounding as shown in figure 3.5b. The planner fails to find a path between two Norwegian fjords.

**SeaRoutes:** (Maritime Data Systems, 2018) This planner tends to make better routes than the one from MarineTraffic, one example can be seen in 3.6a, but the routes can contain sharp local detours as shown in figure 3.6b. This planner is only able to find paths between large ports, and therefor unable to find a path between two fjords in Norway. The planner is able to handle traffic separation schemes as shown in figure 3.6c. This planner has substantially fewer collisions, but they still happen as can be seen in figure 3.6d

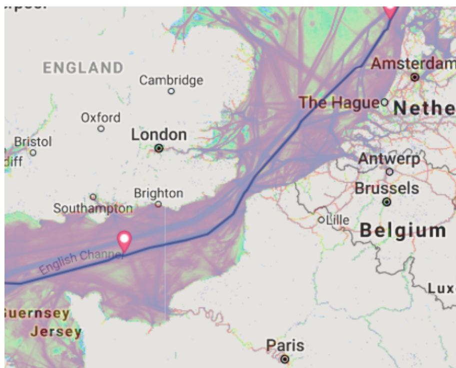
**Netpas:** (SeaFuture Inc., 2018) This path planner is the only one that manages to make a path between the two Norwegian fjords, as can be seen in figure 3.7a. The route consists of very sharp turns that overlap land at some locations. Thees turns lead to suboptimalities as can be seen in figure 3.7b. The planner is able to handle traffic separation schemes.

In conclusion, non of the tested commercial planners are safe to use as a path planner that a ship can follow. They can be used to predict voyage time and be an assistance tool to a human planner.

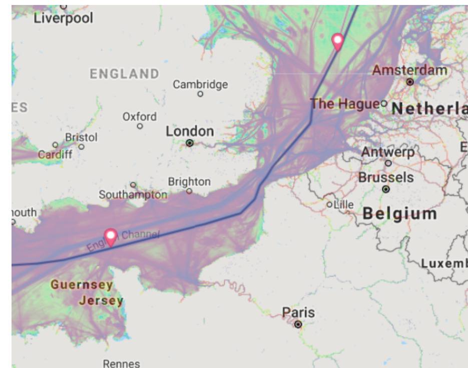
(a) Choosing an unnecessarily long path




(b) The path overlaps land

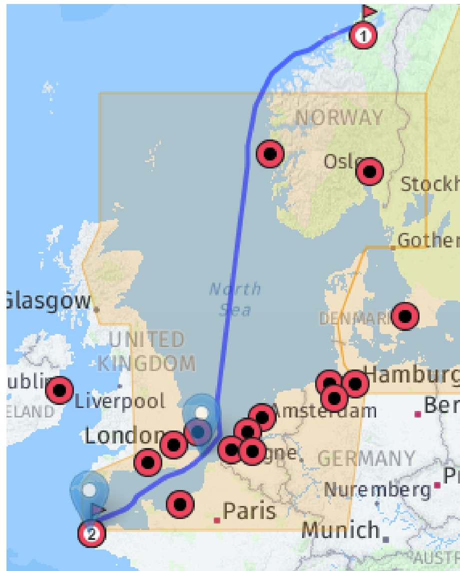



(c) The wrong lane is chosen in the Dover Strait TSS when sailing southwestwards

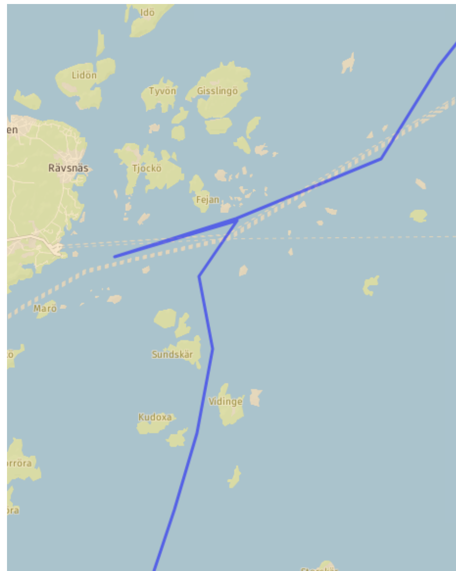



(d) The correct lane is chosen in the Dover Strait TSS when sailing northeastwards.

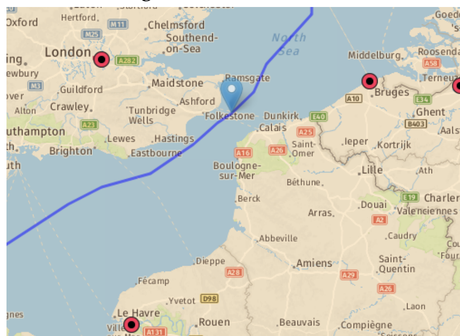
Figure 3.5: Different routes made by MarineTraffic - voyage planner.



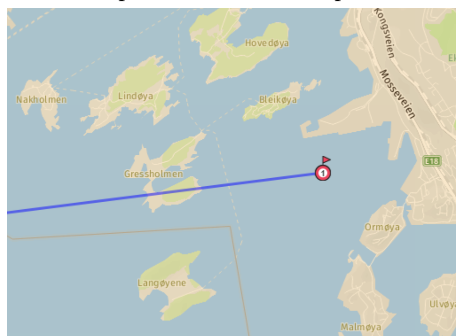

(a) Manages to find an efficient route




(b) The path can contain sharp detours

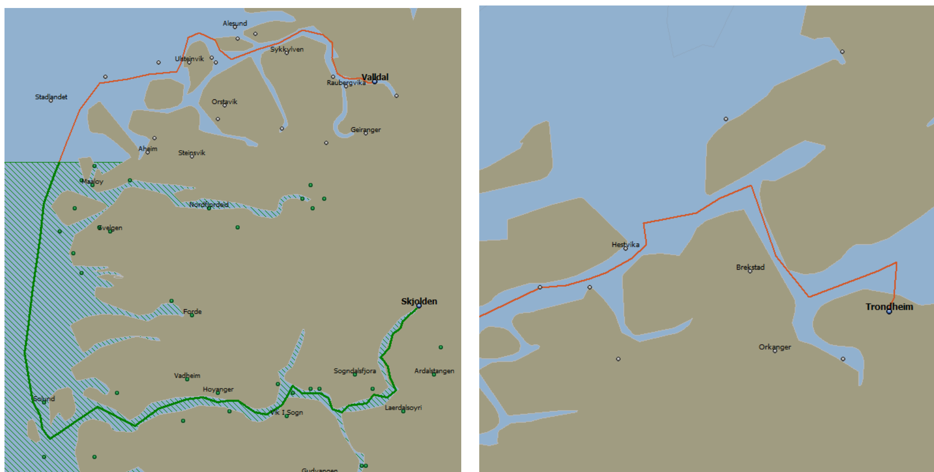



(c) The correct lane is chosen in the Dover Strait TSS when sailing southwestwards




(d) A place where the designed path collides with land

Figure 3.6: Different routes made by SeaRoutes




(a) Route between two norwegian fjords

(b) The path can contain sharp detours

Figure 3.7: Different routes made by Netpas.

# Chapter 4

# Algorithm

The main limitation when developing this algorithm was the long computational time needed to get features from the nautical charts. This is since the NAVTOR API is designed for human users which access the chart at most once a second. The API is designed for extracting a large area of the chart at the time, and displaying this for the human operator. Under the designed conditions, the long computational time needed to get features from the chart does not matter. But for an algorithm that will check thousand of small areas for compliance, this long computational time will be problematic. The algorithm is therefore designed around minimizing how often the chart has to be accessed. In the finished algorithm extracting features from the chart stands for 92% of the CPU-time.

## 4.1 Ship model

The ship will use a path-following controller based on the line-of-sight principle. The controller accepts a low-resolution set of waypoints, where the distance between two waypoints is significantly longer than the ship length.

The algorithm will not make a complicated path where the dynamics of the ship would play a big role. This is to comply with the controller requirements of a low-resolution path, and will lead to the ship sailing in a predictable manner which can reduce the risk of collisions. A very simple ship model consisting of three states is therefore proposed:

$\mu$ - latitudinal position
$l_i$ - longitudinal position
$\theta$ - heading relative to north

The ship can at each time-step choose between a finite set of discreet heading changes, $\Delta h$, that will be held constant for the length of the time-step $\Delta t$. These heading changes must be designed to be physically possible to follow in the given time-step. These could, for example, be chosen to be

$$[-40 \quad -20 \quad 0 \quad 20 \quad 40] \tag{4.1}$$

The heading changes should not be chosen too big, as this will lead to the ship deviainge quite far from the designed path.

The ship is assumed to hold a constant velocity, $u$, equal to the cruise speed of the ship. It will be straightforward to extend the algorithm to use different velocity in different sections. For example, holding lower velocity closer to land. Information regarding currents can also be implemented by adding them to the velocity $u = u_r + u_c$. Here $u_r$ is the ships designed cruise speed relative to the water, and $u_c$ is the speed of the current. If this information is implemented, then the algorithm will automatically utilize it as it tries to minimize the voyage time.

The time-step is set to a constant, $\Delta t$.

$$h[t+1] = h[t] + \Delta h \tag{4.2}$$

$$p[t+1] = translate(p[t], h[t+1], \Delta t u) \tag{4.3}$$

The square brackets indicate the time-step. $h[t]$ is the heading of the ship at time-step $t$. The translate function was introduced in section 2.1.2 and gives the position a ship reaches when following the great circle arc that starts with the course $h[t+1]$ from the point $p[t]$. The arc will be followed for a time-interval $\Delta t$ and the ship will hold the velocity $u$ for the entire interval. This results in a traversed distance of $\Delta t u$.

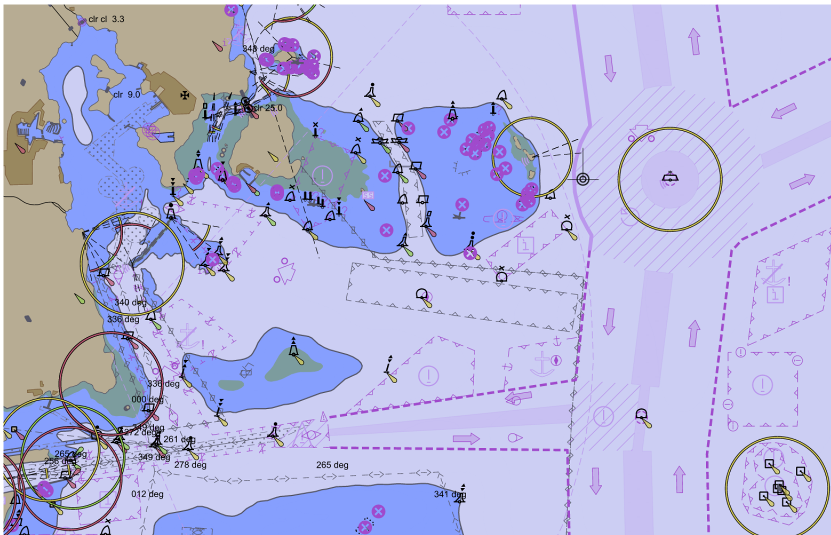



Figure 4.1: Example of navigational chart section displayed using the Navtor G-ECDIS software.

## 4.2 Complying with charted objects

The main goal of the path that is to be generated is to make a safe path with respect to the nautical charts that are supplied. This entails avoiding shallow water areas where there is a risk of grounding. Dangerous obstacles marked in the chart, such as shipwrecks, awash rocks, bridges, and much more, should also be avoided. The path should also comply with traffic routing elements, such as traffic separation zones and restricted areas to minimize the risk of collision with other ships. This information is extracted from nautical charts using the NAVTOR SDK. This SDK has a function that accepts a geometry, parameterized by a set of edge points, and returns a list of all features that are inside this geometry. These features can be different depth zones, traffic separation zones, precautionary areas, oil platforms, and hundreds of other features types. Each feature has an ordered set of points that define the geometry of the feature. For example, will the depth area feature have a geometry that covers the area between two depth contour lines. Each features does also have a set of attributes that are feature specific. The depth area feature will have a couple of different attributes, including the lower and upper bounds of its depth. The traffic separation

scheme lane part feature will instead have a feature that define the required direction of traffic in that lane. All features have an alert priority attribute. This priority has a value of 1 if colliding with the feature could lead to grounding. Land areas and depth areas that are shallower than the specified safety depth are marked with alert priority 1. If the value is 2 then the feature may in some way be relevant when planning a path. Examples of areas with alert priority 2 are traffic separation zones, recommended tracks, deep water routes, restricted areas, buoy, awash rocks, ship-wrecks, bridges, cables and many more.

When the path planner checks a geometry for collision, all the features that are returned are iterated through and tested for compliance. If one of the features break a constraint then this function returns with failure. First, the alert priority of the feature is tested, thereafter features specific tests are done. Some features will simply return a false such as awash rocks, shipwrecks, buoys, bridges and overhead cables lower than the safety margin. Others features such as separation zones have more complicated compliance check. The rest of this section is dedicated to the compliance check of different features.

**Traffic routing features**

There are two main ways of routing the flow of traffic, recommended tracks, and required tracks. The recommended tracks can appear in many different forms such as "recommended route centerlines", "recommended track", and "recommended traffic lane part". All of these except for the last one are lines the ship can follow, the last one is an area where the ship should follow a given heading. The required tracks are always presented as areas marked as "Traffic separation scheme lane parts". These areas split the flow of traffic into to separate lanes, where each lane has a required heading. Ships sailing in a traffic separation schemes must either follow the dictated flow of traffic, or cross the lane with a heading 90° on the traffic flow (IMO, 1996, A, 10.c). For safety reasons, this algorithm will avoid making routes that cross the traffic separation schemes, but will be allowed to cross areas where there is a recommended track. Traffic separation schemes have areas in between the lanes and on the side that are marked as separation zones or lanes. These areas can be seen on figure 4.2. The full arrows are "Traffic separation scheme lane parts". In between the two lanes is a "Traffic separation zone". On the right side of the separation scheme is a "traffic separation scheme boundary", on the left is a "Traffic separation line".

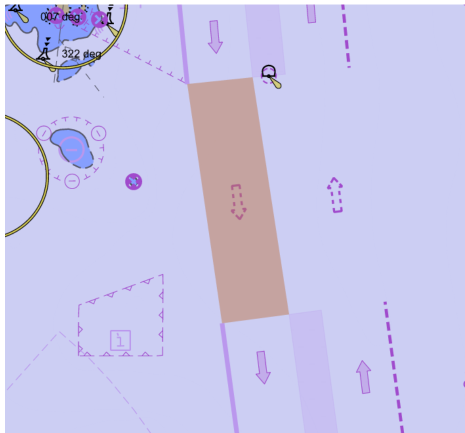
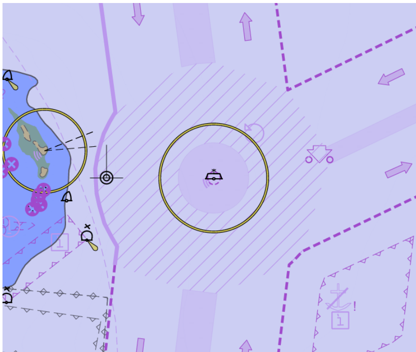


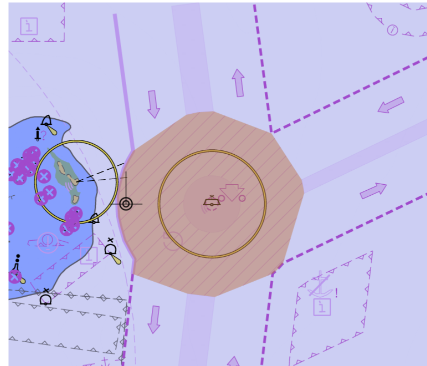

Figure 4.2: Traffic separation zone. The dotted arrow represents a "recommended traffic lane part" and the full arrow a "traffic separation scheme lane parts". One of the recommended lane parts is highlighted to show the area marked with this feature.

All of these features are illegal to cross, and this function will return failure if the tested area contains any part of these features. The dotted arrow on figure 4.2 is a "recommended traffic lane part". The recommended traffic lane part spans the area highlighted in orange, and does not have separation zones in between. For the "Traffic separation scheme lane parts" a small deviation of heading error is accepted. This is currently set to 45°. The "recommended traffic lane part" accepts a higher heading deviation of 110°. This allows the ship to cross and join the traffic routing feature at the "recommended traffic lane parts".

The other types of recommended tracks are ignored. These are often only applicable to some types of ships, for example a recommended track for yachts, and should therefore be implemented based on the current mission. They can be implemented by giving the search algorithm a penalty every time it does not follow a recommended track. The penalty must be scaled by how much of a detour the ship should take to follow the recommended tracks.



(a) Traffic separation scheme roundabout. The roundabout should be traversed counterclockwise.



(b) Traffic separation scheme roundabout where the area marked as a roundabout is highlighted.

Figure 4.3: Traffic separation scheme roundabout

### Roundabout

Another traffic separation scheme object is the roundabout which is shown in figure 4.3a. The roundabout should be navigated in a counterclockwise fashion. The roundabout feature contains only the set of points that outline its geometry, and does not contain any information on where its center-point is. The roundabout in figure 4.3a has a "traffic separation zone" (pink area) with a buoy in the middle. This is generally not the case, and the roundabout feature is not linked to the traffic separation zone in any manner. Using this area is therefor not practical. Which way the ship is traversing the roundabout can be found by checking that the closest point on the roundabout border is on the right side of the ship, relative to the ships heading. This solution is not perfect as it requires that the roundabout is approximately circular and has a high resolution, both which are not necessarily the case as can be seen in figure 4.3b. This strategy will disallow some legal position and heading combinations, but will not allow illegal position and heading combinations. It will also allow some legal routes. It will therefore be a too strict rule that guarantees that the rules are followed. A legal position and heading combination that will be disallowed by this rule is shown in figure 4.4.

The roundabout test is done by first checking the distance to each point on the roundabout border and choosing the closest point. The distance is found using the

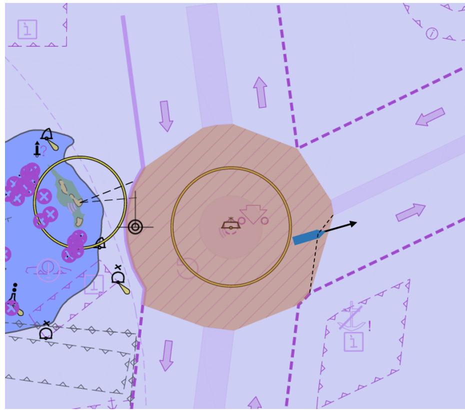


Figure 4.4: A legal ship position and heading where the closest point on the roundabout border is on the left side of the ship.

$Great_Circle_Distance(p1, p2)$ function presented in chapter 2.1.1. To check if this point is on the right side of the ship, the heading, $\theta_2$, needed to reach the point is calculated using the $Initial_Heading(p1, p2)$ function presented in chapter 2.1.3. The difference between this heading, $\theta_2$. and the ships heading, $\theta 1$ is calculated, and normalized to the interval $< -180°, 180°]$. The difference is denoted as $\Delta\theta$. If the difference in heading is greater than 0, then the heading needed to reach the point was a higher heading value than the ships heading. As the heading values increase clockwise, a higher heading value represents that the point was to the right-hand of the ship. When the change in heading would be large enough to place the point on the left side of the ship, then the normalization would have altered the heading change to be negative.

**Algorithm 1** Check roundabout compliance

```
1: p_1 ← ship position
2: θ_1 ← ship heading
3: p_2 ← position of closest point to ship
4: θ_2 ← Initial_Heading(p_1,p_2)
5: Δθ ← Fix_Angle(θ_2 − θ_1 )
6: if Δθ >= 0 then
7:     return TRUE
8: else
9:     return FALSE
```

**Algorithm 2** Fix angle

```
1: function FIX_ANGLE(θ)
2:     while θ <= −180° or θ > 180° do
3:         if θ <= −180° then
4:             θ ← θ + 360°
5:         else if θ > 180° then
6:             θ ← θ − 360°
7:     return θ
```

### Inshore traffic area

The area between a traffic separation scheme and land tends to be marked as an inshore traffic area. Innshore traffic areas should generally be avoided if the ship can safely sail around it, unless the ship is sailing to or from a port inside the area IMO (1996). The area can also be used by small vessels, sailing vessels, and vessels engaged in fishing. The path will therefore be allowed to enter inshore traffic areas, but not leave them. This will make it possible for the path to find goal states within the zone while still avoiding to sail through the zone. One problem with following traffic separation zones is that it is often possible to sail around the zone without hitting any features with constraints. This problem is avoided when there is an inshore traffic area on both sides of the separation zone, such as in the English channel, see figure 4.5. In other cases when the inshore traffic area is only on one side, as in figure 4.6, the problem persists.

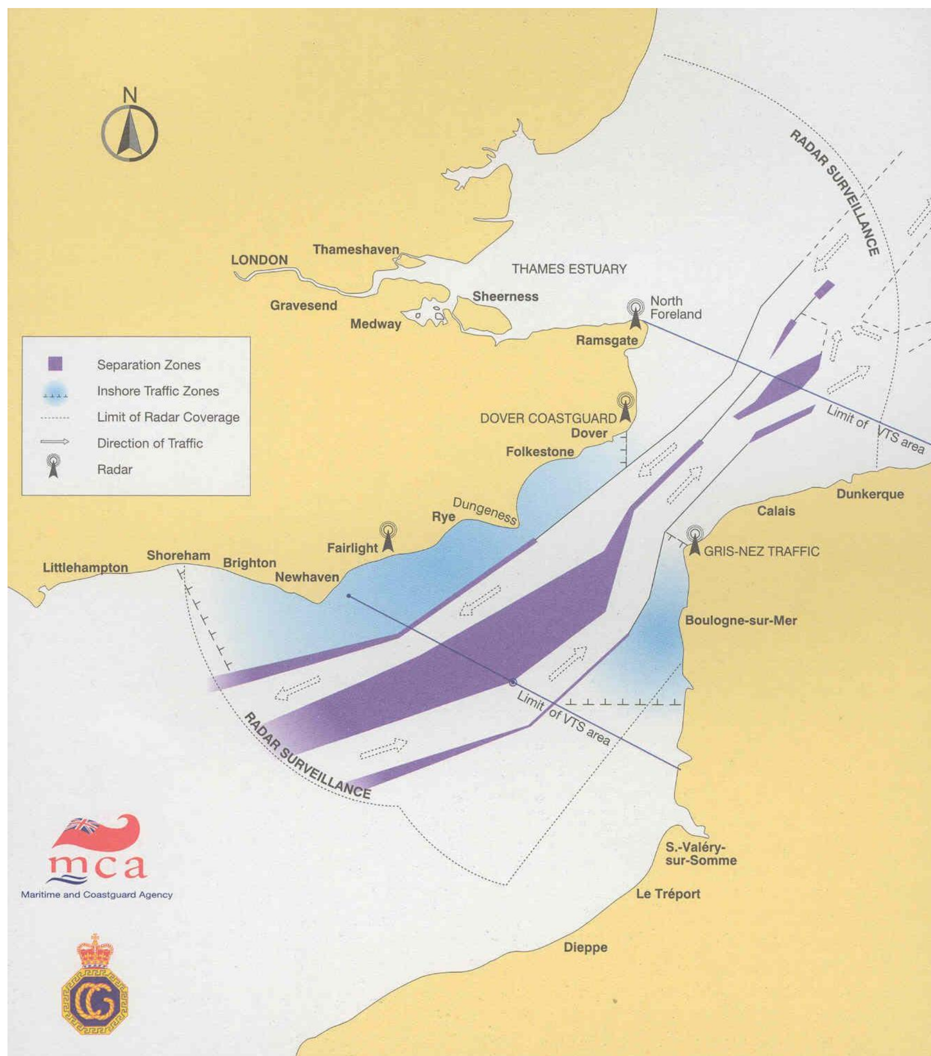



Figure 4.5: Traffic separation zone at the Dover strait. There is an inshore traffic area on both sides of the strait.
© Crown copyright 2010
www.mcga.gov.uk

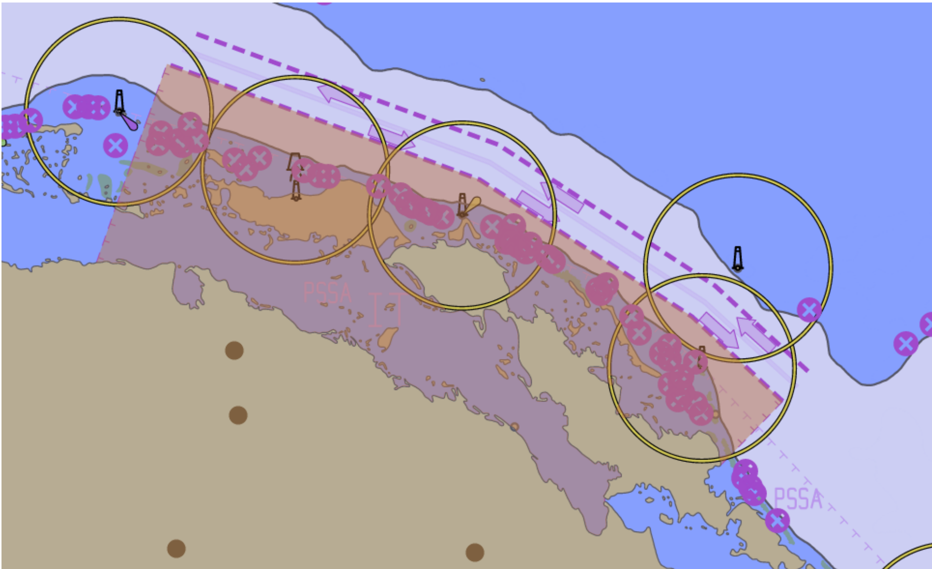



Figure 4.6: The area between a traffic separation zone and land tend to be marked as an inshore traffic zone.

#### Other features

There are hundreds of other charted objects that should in some cases be avoided, or in other cases followed. These often depend on the mission at hand. These features will be ignored for this thesis, and should be implemented in a real mission in a way that matches that specific mission. Such features can be restricted areas where some ships may be disallowed entrance in some cases.

## 4.3 Formulation as a graph search problem

In this section we will use a terminology that all vertices that can be reached from the vertex, u, are the children of u, and u is the parent of these vertices. All the children of u are its successors, and the parent of v is its predecessor.

The concept of using a procedurally generated graph with continuous position coordinates and a discreet set of control inputs are taken from Dolgov et al. (2010). This source used a model of a car with a three different steering commands. Here

more possible heading changes were introduced to avoid smoothing the path after it is generated. Having a precise smooth path is more important when navigating a parking lot than when finding a safe path for a ship to follow. This source also introduced the concept of a grid-map that will be used in section 4.3.1, but it did argue on why this was done.

The algorithm starts with a graph consisting of only the given start state of the ship as a vertex. The graph will gradually grow using the ship model and explore new areas that are kinetically feasible for the ship to follow. In section 4.1 the ship model was introduced with a finite set of possible heading changes. Equation 4.2 calculates the new states that will be reached by exposing the ship to the different heading changes. Each vertex in the graph can generate successor vertices that correspond to these new states. One new vertex for each heading change. With this discretization the requirements for a graph search problem which were presented in chapter 4.3 are fulfilled. Due to the discretization of heading changes, there are a finite set of discrete actions that can be applied to each state. Due to the discretization of the time steps, the states will be discreet. The heading changes are chosen small enough to guarantee that the path following controller is able to move the ship from one state to another. The ship will have some way of estimating its position, using for example GNSS and INS. Knowing the state of the ship enables the path following controller to know which vertex it should follow. This fulfills the requirements of observability of the state. The discretization of state and control input makes the problem solvable by graph search, but hinders the problem to be solved to optimality. The optimal path will probably require other heading changes than the ones presented in the ship model. Having more possible heading changes will make the solution closer to the optimum, but will drastically increase the computational time. A trade-off between computational time and optimality of solution will therefore have to be made.

Each round of the algorithm the vertex with the lowest $f = g + h$ value will be chosen to be explored. This is the principle introduced in the A* algorithm that will lead to the most promising vertices being explored first, this will, as discussed in chapter 2.4.2, in most cases lead to drastically fewer vertices that need to be explored. When a vertex is explored a new vertex is created for all the possible heading changes that are presented in chapter 4.1. The states corresponding to the new vertices are created according to equation 4.2. The next step of the algorithm is to remove all of the new vertices that are unsafe. This is done by making a rectangular geometry

around the parent vertex and the newly generated vertex with a given margin on all four sides. This rectangle will thereby encompass the entire route the ship will follow from the parent to the child vertex, given that the margin is chosen large enough. This rectangle is tested for compliance with charted objects, as described in section 4.2. If this rectangle passes, then there are no obstructions along the path connecting the two points. The margin must be chosen big enough to guarantee that the ship will manage to stay inside the margin an all sailable weather conditions. The heading used at the parent vertex to reach the child vertex is used to check if the ship is traveling in the correct direction in traffic separation zones and recommended tracks. This heading is saved in the child state. The state will be deleted if it fails the test. All states that pass the test will get their $f(v)$ value calculated. The algorithm will then continue to explore the most promising unexplored vertex, the one with the lowest $f(v)$ value, until a termination criteria is reached.

The $f(v)$ value is the sum of the cost it has taken to reach the current state, $g(v)$, plus the heuristic estimate of the distance left to the goal state, $h(v)$. The $h(v)$ value is chosen to be the great circle distance to the goal state divided by the cruise speed of the ship. This will result in the time it takes to follow the great circle at the ships highest used velocity. This heuristic will be admissible as there are no shorter paths between two points on a sphere than the great circle distance, and the ship will not plan to sail with a higher velocity than the cruise speed. The optimal path for the ship will at least take this amount of time to follow. The small errors in distance that appear due to the earth not being a perfect sphere will be insignificant. This small error might move the route slightly further away from the optimum, but will be much smaller than the error due to the discretization of the states. The shortest path from a point, a, to any neighboring point, b, can be lower bounded by the great circle distance. The great circle path from the current point, $a$, to a neighboring point, $b$, then to the end point, $c$, will never be shorter than the direct great circle distance from a to c. The heuristic will therefore be consistent as the real distance from $a$ to $b$ will be at least this long, fulfilling the requirements of: $h(u) \leq h(v) + d^*(u, v)$. Scaling the distances by the cruise speed will not affect this result. The transition cost, $g(v)$, is chosen to be the time the transition took to get to that state plus a penalty for changing heading. This penalty is introduced to prefer straight predictable courses. The penalty is designed to increase with the size of the heading change to encourage small changes. Small

heading changes are safer, easier for the path following controller to follow, and are more predictable for other ships.

The values are calculated as follows for the transition from state $u$ to $v$, with the start state $s$ and goal state $t$. $dt(v)$ is the time-step length for vertex $v$, and $dh(v)$ was the heading change used to reach $v$ from $u$. $dh_{max}$ is the largest allowed heading change between two states. $\alpha$ is a factor that decides how large a penalty the largest heading change can give relative to the time penalty. $v_c$ is the cruise speed.

$$h(v) = \frac{Great_Circle_Distance(v,t)}{v_c} \tag{4.4}$$

$$g(v) = g(u) + dt(v) + (\alpha dt(v))\frac{dh(v)}{dh_{max}} \tag{4.5}$$

$$f(v) = g(v) + h(v) \tag{4.6}$$

To keep track of which vertex should be explored next, references to all generated and unexplored vertices are kept in a sorted list, called the *open_list*. The elements will be sorted by rising $f$ value. When a new vertex is generated, a reference to that vertex is inserted into the *open_list* at the correct sorted position. Only two operations will be done on the list, retrieving the first element, and inserting at a sorted location. Finding where to insert an element can be done in O(log(n)) time with binary search Cormen et al. (2009, p. 800) Binary search requires random access which is supported by *std::vector* in c++. But inserting at a random location in a *std::vector* takes O(n) time, as all elements after the insertion point need to be moved (unless a reallocation happens to allocate more space, then all elements need to be moved) (cplusplus.com, 2017b). It can therefore be more efficient and simpler to use a linked list, such as *std::list*, which supports constant time insertion, but no random access (cplusplus.com, 2017a). As a linked list does not support random access, using binary search is therefore impossible. Finding the correct location to insert into a linked-list would require searching through the list from the start, one element at the time, until the correct location is found, this will be done in O(n) time. The element can then be inserted in constant time at that location. Removing elements at the beginning of a linked list takes constant time. The *open_list* can therefore be efficiently implemented in c++ as a *std::list*. When a vertex is explored, the reference

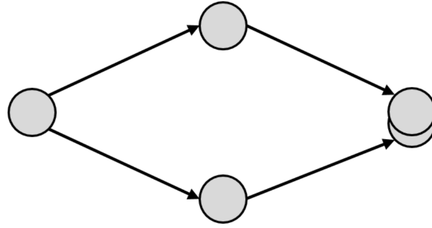


Figure 4.7: The are multiple ways of ending in almost the same state.

stored at the first location of the *open_list* is retrieved, and the first element is popped from the list, making the list one element shorter.

### 4.3.1 Grid-map

A problem that arises with directly using a procedurally generated graph as presented in the algorithm above is that the number of unexplored vertices grows way to fast. In open areas the A* algorithm will avoid checking many of the unnecessary vertices as the most promising vertex actually is the best candidate. In areas where the most promising path dead ends, the problem arises. There will be way too many possible paths that lead into the dead end area in slightly different ways. If the dead end is close to the goal exploring a path that goes in a zigzag or in circles inside the dead end could seem more promising to the heuristic than exploring a path that goes around the obstacle. The algorithm will thereby be stuck checking the different paths inside the dead end for a way too long time. Many of the states that are explored are almost identical to many other states, they are just generated through a different combination of heading changes. For example will a 10° turn left followed by a 10° turn right end up in almost the same state as a 10° right turn followed by a 10° degree left. Now both of these states that have the same location and only a slightly different heading will have to be explored separately. This effect is shown in figure 4.7. The problem of exploring almost the same state multiple times also arises outside dead-ends, but it is especially prevalent in dead ends. In testing, it could go a couple of thousands of iterations while the algorithm continued to search through different states in a dead end area that was close to the goal.

The problem of testing almost identical states multiple times was solved by drastically limiting the search space by introducing a grid-map that is overlaid the navigational chart. This grid-map limits the number of vertices per grid to one. When a new vertex is generated, its corresponding grid is found. If the grid is empty then the vertex will be saved in that grid, if it is full then the vertex will be deleted. The location of vertices are still defined by their latitudinal and longitudinal position state, so the search is still with continuous positions. Having a grid-map makes the search not guaranteed to find the optimal solution, as the algorithm will greedily only let the most promising vertex stay in the grid. The optimal route will not be found if a less promising vertex in the same grid would be a part of the optimal path. The grid should be design in such a manner that all successor vertices of a parent vertex end in different cells. A smaller grid size will make the path more able to do highly dynamic maneuvers, but will substantially increase the computational time.

For simplicity's sake the grid-map was introduced as a constant size two-dimensional array. This strategy allows for constant time access, but uses a lot of unnecessary memory. It does not allow for very small cells over a very large area, as this would use up the program stack. With this strategy, most of the cells will be empty but still take up memory. If the size of the program stack is a limitation then the grid-map should dynamically grow to make space for new cells, while not retaining any empty cells. This could be done by first making a cell when it is needed, and then save the cells on a sorted location in for example a *std::vector*. The cells can be sorted with respect to their y grid-coordinate first, then by the x grid-coordinate. Binary search could be used to quickly find the cell again. This approach would use O(log(n)) time instead of constant time for access, but require substantially less memory.

## 4.4 Bidirectional search

As discussed in the theory section 2.4.3 using bidirectional search can reduce the number of vertices that need to be explored. This will reduce the number of grounding checks that need to be done, which can save a lot of computational time. The bidirectional search is done by running the algorithm presented thus far two times in parallel. One of the searches has the start state, $s$, as the initial vertex, while the goal state, $t$, is the goal vertex. The other one has $t$ as its initial vertex, and $s$ as the goal vertex. The backward search will work in the same way as the forward

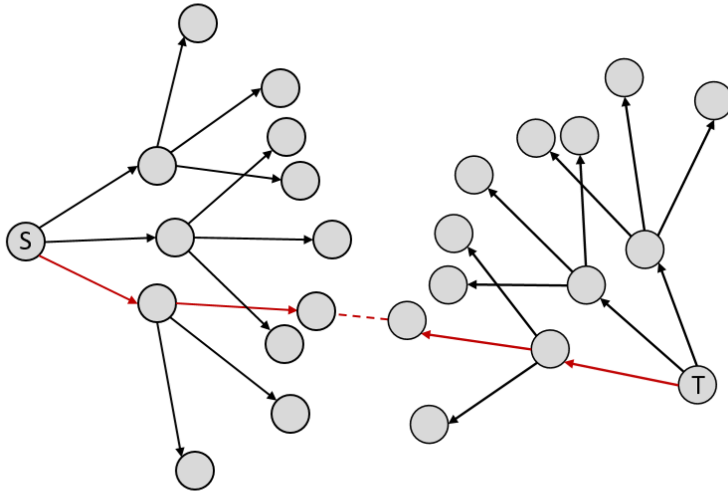



Figure 4.8: Two search trees that have met.

search, with the change that it will generate new states in the opposite direction of the heading variable. The heading variable will therefore show the heading the ship will have when sailing the path in the correct direction.

The main challenge with bidirectional search is how to connect the two search trees. This is solved by using the grid-map introduced in chapter 4.3.1. A maximum allowed connecting distance is chosen. This should be chosen to be larger than half the distance traversed in one time-step to ensure that the two search trees cant grow past each other without hitting. All grids that are within this distance from the explored state are checked if they are connectable. Two states are connectable if they are from different search directions, within a radius of the given maximum distance, are both explored vertices, that they have a change in heading within the given maximum, and finally that the transition between the states complies with the navigational chart. Both have to be explored vertices as these have a better $f(v)$ value than the current vertex, and are therefore good candidates for connecting the two graphs. Unexplored vertices may have a terrible $f(v)$ value. The compliance with the navigational chart is done last to avoid unnecessary chart lookups. A change in headings is acceptable if the change from both of the states to the heading needed to reach the other state are within the given maximum heading change limit. The

transition with the lowest transition cost among all transitions that fulfill these criteria is chosen.

This way of connecting the two graphs does not take the transition cost between the connecting vertices fully into account. If this transition cost is high, then the path might want to continue its search until it finds a cheaper connecting point. This is not a problem as the transition cost is dominated by the traveling time between the vertices. Finding a longer route that avoids a potential high-cost area will therefore not be relevant. Connecting to the first possible connection that is found will be close to optimal.

## 4.5 Bidirectional search with weather

### 4.5.1 Weather data

To ensure a safe journey the planned route has to avoid areas with dangerous weather conditions. The ship has to either sail around the weather or delay its departure from the initial port to wait out the weather. The ship should not wait at any other point in the journey.

The problem with using weather data in a bidirectional search, is that it is time-dependent. The weather at any given vertex in the backwards search is unknown, as the arrival time at the goal state is unknown. Finding when the ships arrives at a vertex in the forward search is straightforward. The time the ships arrives at a vertex is the planned departure time plus the time it took to traverse each edge that placed the ship at the given vertex.

To simplify the problem, the weather can only have two possible states, too dangerous to sail and acceptable. Areas with wind speed stronger than the accepted maximum will be marked as too dangerous to sail. The area will only be marked as dangerous for the time interval the wind is too strong, plus some time before and after as a margin. The margin is introduced since weather data is not perfect, and the exact arrival time of the ship is a bit uncertain.

Global weather data is available through multiple services such as the "European Center for Medium Range Weather Forecast" and the American "Global Forecast System". The weather data can be downloaded through different APIs, such as the "Open Weather Map" API. This API lets you insert a latitude/longitude coordinate and returns weather data, including wind speed, for the given coordinate. This weather

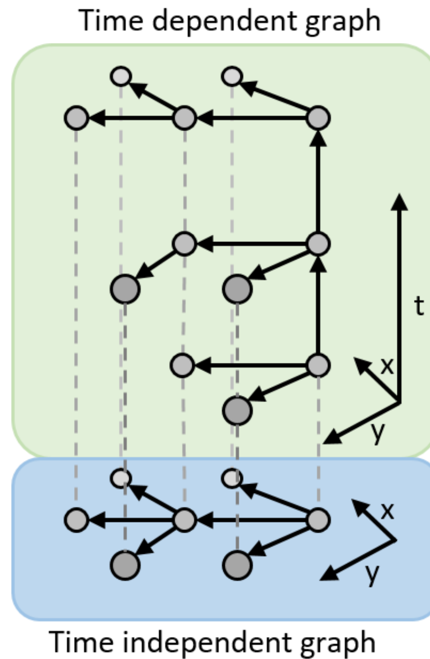



Figure 4.9: Time independent and time dependent graph shown over their respective geographical coordinates. The transitions upwards in time illustrate waiting at port for the weather to move. The figure does not display that time moves as the ship moves along its path.

data has a resolution of 3 hours between each data-point, and has a forecast up to 5 days into the future. It also supports daily forecast for 16 days into the future.

Only simulated weather will be used for this thesis. This is to avoid having to wait for the real weather to behave in a problematic fashion, and to get the same weather in multiple tests. Sailing around weather is generally only a possibility in open waters such as the Atlantic ocean. As I was not able to acquire maps for large open-water areas it was impossible to test the algorithm in areas where sailing around the weather is possible. The simulated weather can on the other hand be designed in such a manner that sailing around it is realistic for smaller areas.

### 4.5.2 Bidirectional search with time dependent constraints

The first problem to solve to implement weather is how to efficiently implement the ability to wait at the departure port. The easiest way to do this is to simply extend the possible actions the start vertex can do to include waiting for a specified time interval. Using this solution directly would waste a lot of computational time as it would lead to the same set of states being tested for chart compliance multiple times at different time-steps. The navigational charts do not change over time, so one state should only be checked once for chart compliance. This is done by introducing two graphs, which are illustrated in 4.9. One, which is time independent, tests for chart compliance when a new vertex is explored. The other one, which is time-dependent, can explore the time-independent graph at different time-steps. The time-dependent graph will have the ability to wait at the start vertex, and may therefore have multiple vertices with the same geographical position. Whenever the time-dependent graph explores a new vertex, it will ask the time-independent graph for its successor states. If this vertex is already explored in the time-independent graph, then its resulting successor states are directly returned without testing them for chart compliance again. If the state is not already explored, then it will be explored by generating successor states and testing them for chart compliance before they are potentially returned. The time-dependent graph will delete all vertices that break with a weather constraint. The time-dependent graph vertices will have a reference to the time-independent graph vertex with the same geographical coordinate. The time-independent graph vertex will have a list with referenced to all time-dependent vertices that share its geographical coordinate. There can be multiple time dependent vertices per time independent vertex. The time-independent graph will span all the possible geographical positions visited by the time-dependent graph at all possible time steps, plus the positions covered by time-dependent vertices that were deleted since they collided with unacceptable weather.

The backwards search has no way of knowing when it will reach its vertices, as it does not know when the goal state is reached. The backwards search will therefore unlike the forward search not check if a vertex complies with the weather constraints, it can also not wait at the goal vertex as this makes no difference. The forward search will check each new state for weather compliance. Forward search states that have the time and place that falls within the illegal weather conditions are deleted. The method for handling bidirectional search with time-dependent constraints presented

in Nannicini et al. (2012), which was presented in the theory section, cannot be used in the case of a procedurally generated graph. The forward search algorithm has no natural way of continuing its search amongst the vertices explored by the backwards search. The backwards search will grow outwards as a tree, there will therefore only be one possible path from a vertex to the goal. The forward tree will therefore have only one way to go from the connecting vertex, and no way to chose a different route if this route defies the weather constraints.

Instead of searching amongst the backwards tree, the forwards tree will gradually reject illegal parts of the backwards tree until a legal route is found. When the two search trees connect it will be possible to know when each of the vertices in the path will be reached. The path can then be checked for weather compliance. If the path traverses unsafe weather then the connecting vertex from the backward search and all of its successors will be marked as rejected. A rejected state will not be explored, but can be connected to from the forward search as it might not break a weather constraint at a later time-point. The backward and forward search will then continue until a connection is found that does not break with the weather constraints.

When a new forward search vertex is generated that falls in the same cell in the grid-map as a rejected vertex, the rejected vertex will be replaced with the new vertex. All successor vertices of the rejected vertex will also be deleted. It is better to delete too many of the backwards search vertices, than letting one of them stop the forward search with a branch that breaks with a weather constraint. When a vertex in the forward search is explored, all backwards search vertices within the connection radius are found. All of them that break the weather constraints are marked as rejected. All of the vertices are marked and tested without checking if the connection is physically possible for the ship to follow, or if the connection would break a chart constraint. This is to ensure that there are no backwards search vertices that should be deleted that are not. This happens a lot as the forward search will gradually delete the backwards search, leaving behind subsections of the backwards tree that can be hard to connect to. The forward search will thereby gradually deletes illegal parts of the backwards search that are in its way.

This strategy will in the worst case have to delete most of the vertices in the backwards search, making it slower than in a monodirectional case. In most cases with weather constraints the backwards search will be helpfully as not all of the

backwards search nodes will be rejected. Dangerous weather conditions are also semi-rare. Having a bidirectional search may therefore still be advantageous.

This solution bases itself on the fact that weather does not change quickly in distance or time. Sailing detours to wait out the weather should be avoided as this will waste fuel and crew time, the ship should rather wait out the weather at port. The weather is assumed to not be dynamic enough that the ship will have to wait in the middle of the voyage to avoid the weather. With these assumptions it is acceptable to prevent vertices in the backwards search from being explored further when the two search trees meet. By stopping the search here, and due to the grid-map preventing a new vertex to take the same position as previously explored vertex, the path planner may lose solutions where the ship sails a detour to wait out the weather. This is as discussed wanted behaviour.

The two search trees should be hindered from growing into and past each other if for some reason some vertex in the trees failed to connect to another. This is done by marking all vertices that are close to a vertex from the other search direction, but unable to connect to it, as rejected. This distance should be chosen small enough that only vertexes inside the other search tree are rejected, and not vertices passing on the side of the other search tree.

## 4.6 Post-processing

As it is preferable for the path planner to have a low-resolution path, all unnecessary waypoints in the path should be deleted. This is simply done by deleting all waypoints where the ship has the same heading on the way to and from the waypoint.

The path produced will not be perfect mainly due to the discretization of heading changes. It can therefore contain unnecessary turns and waypoints that should be removed. An example of this is given in figures 4.10 and 4.11. As the goal is to make a predictable route that is easy to follow, all waypoints that can safely be removed should be removed. A waypoint can safely be removed if the resulting path does not contain any heading changes sharper than the given maximum, and that it does not break with any chart or weather constraints. The algorithm will start by trying to remove a large number of trailing waypoints at the time, for example 10, and then try with gradually fewer trailing waypoints until one and one is tested. Removing only one and one waypoint is not enough as a detour consisting of for example 2 waypoints

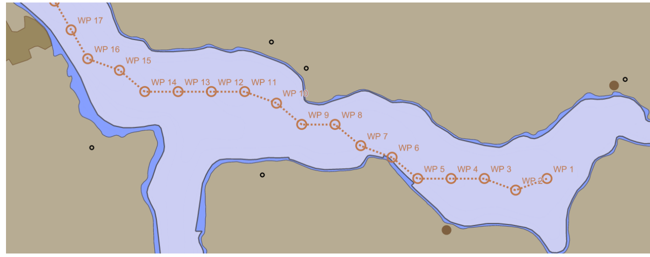



Figure 4.10: Unnecessary turns due to the discretization of the heading change input.

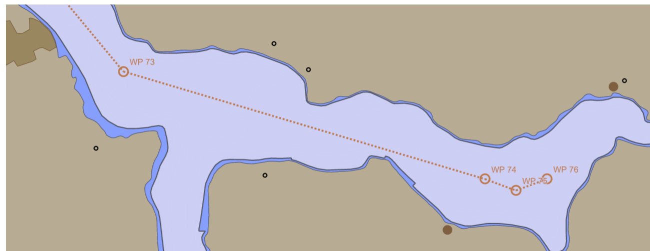



Figure 4.11: The path after post-processing. Unnecessary turns are removed.

might break the heading change constraint by removing a single waypoint, but not break the constraint by removing both at the time. Starting with many waypoints at the time will remove larger unwanted features before the smaller unwanted features that are left are removed. Removing a waypoint might make it preferable to remove an earlier waypoint, which was undesirable to remove when it was last checked. This function will therefore repeat until it finishes one round without changing anything, and then reduce the number of trailing waypoints that are tested.

# Chapter 5

# Results and discussion

This section will display a set of routes generated by the path planner developed in this thesis. Different situations that highlight the properties of the algorithm will be presented. The paths are displayed using the NAVTOR G-ECDIS software. This software uses a series of different resolution charts, where a higher resolution chart is displayed when the user zooms in past a specified level. The NAVTOR SDK used to extract navigational data extracts the highest resolution chart for the area. Lower resolution charts are used in a lot of figures in this section, as this allows the figure to show larger areas at the same time. The lower resolution charts deviate a tiny bit from the high resolution charts, they tend to be more restrictive. The paths in the plots may sometimes touch the land area, this is only due to the error that arises by using a low resolution chart. In the higher resolution chart, which are more correct, this is not a problem.

All the plots are done with the following settings

- Legal heading changes = [40, 20, 0, -20, -40]
- cruise speed = 8m/s
- safety depth = 9m
- max deviation for required traffic direction = 45°

The rest of the parameters change between plots and are specified for each plot.

## 5.1 Long distance routes

This section is done with the following parameters:

- time-step = 120s
- margin to obstacles = 100m
- grid cell size = $0.008° \times 0.008°$

Figure 5.1 shows that the path planner manages to make a route between two fjords in Norway. The path is approximately 460km long and goes from Skjolden in Lustrafjorden to Valldal in Storfjorden. The path manages to navigate narrow areas as shown in figure 5.2b, avoid small local obstacles such as awash rocks as shown in figure 5.2a, and make smooth turns around edges as shown in figure 5.2d. The path that is generated is guaranteed to be at least 100m away from the land. But the algorithm does not try to place the path further away when that is possible. Figure 5.2a shows that the path navigates in between some awash rocks that are within the safety margin away from the path, while it would be safer to sail around them. The margin should be chosen large enough that sailing this close to land should not be dangerous. Sailing further away from land than the margin can still be advantageous, as this introduces a larger safety margin. Figure 5.2c shows that the algorithm sometimes generates slightly sub-optimal paths. This path segment can not be improved by removing waypoints making the post-processing algorithm unable to improve the segment.

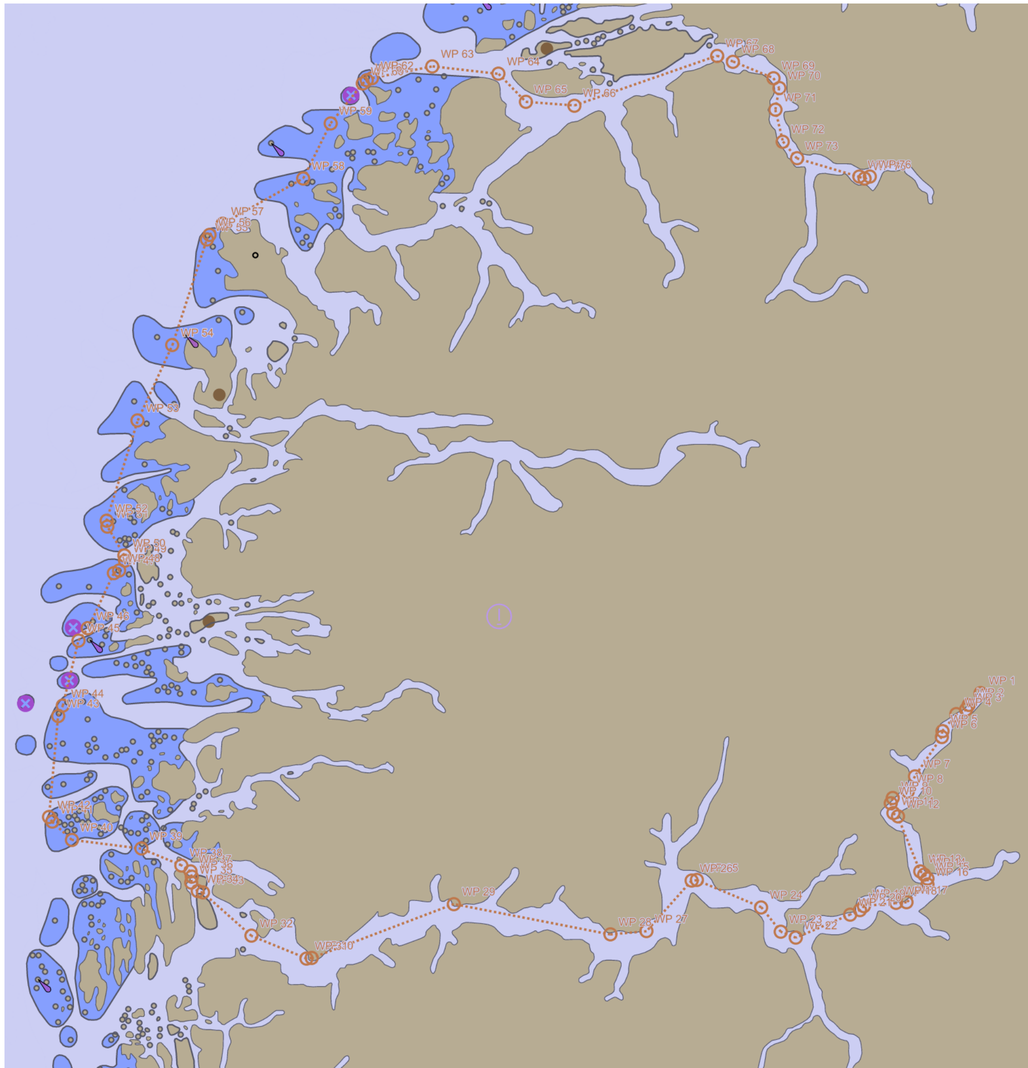
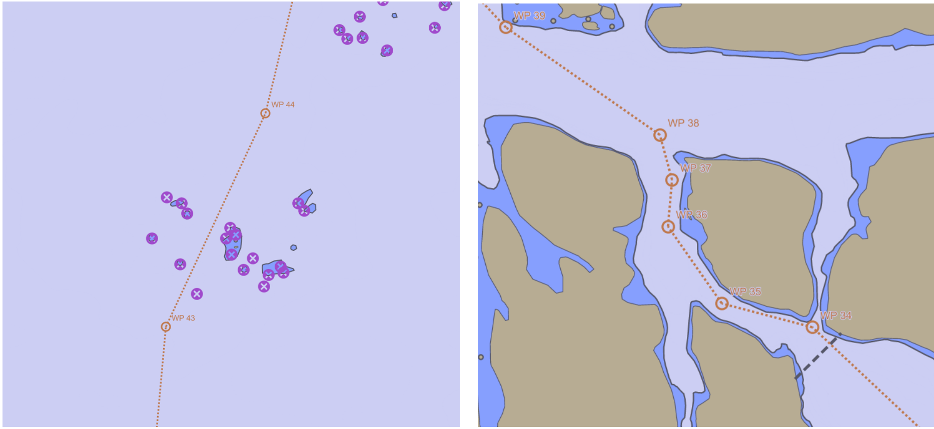
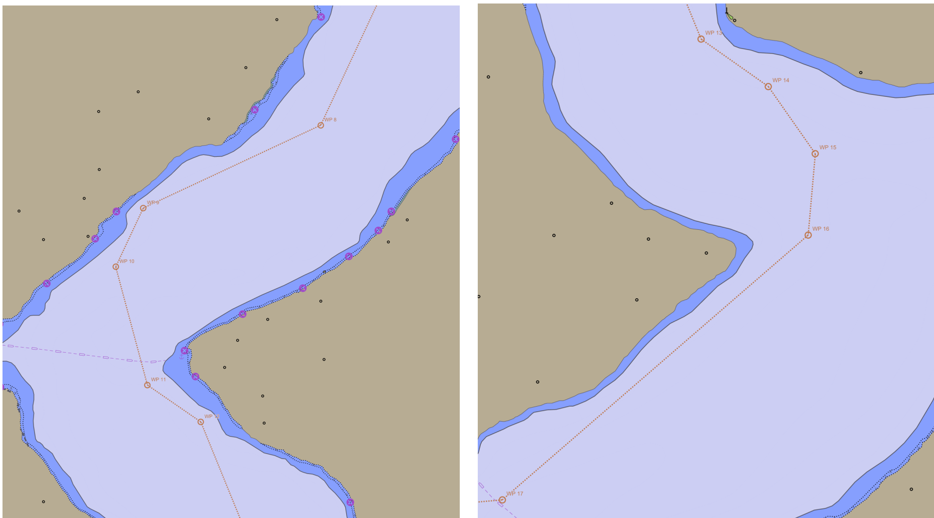



Figure 5.1: A path found from Skjolden in Lustrafjorden to Valldal in Storfjorden, both in Norway




(a) The path avoids small features such as awash rocks

(b) The path manages to navigate narrow areas



(c) The path can contain some slightly sub-optimal turns

(d) The path manages to make smooth turns around sharp edges

Figure 5.2: A closer look at parts of the route presented in figure 5.1

## 5.2 Traffic separation schemes

This section is done with the following parameters.

- time-step = 120s
- margin to obstacles = 20m
- grid cell size = 0.008° × 0.008°

This section show how a path is formed when there are traffic separation zones present. The chart used is a test-chart that was delivered together with the NAVTOR API. This test-chart is designed to contain many of the different features that can arise in navigational charts. The path will start in the small yellow circle on the left of figure 5.3 and end in the small yellow circle on the right. The endpoint is placed inside a traffic separation scheme. The path has to enter a correct traffic lane at a legal point, navigate the roundabout, and then find the goal position. Figure 5.4 shows the state of the forward and backward search tree when the route was found. The tree has only explored routes where the ship has a legal heading. A deviation of 45° in heading was allowed inside the traffic area. A smaller heading deviation could be accepted if the path planner had higher resolution on the possible heading changes, for example 10 degrees instead of 20.

The path through the two trees that managed to make a connection are shown in figure 5.5. The forward search has navigated through the narrow channel on the left, and the backwards search has navigated the traffic separation zones. The final path after post-processing is shown in figure 5.6.

A problem is that the algorithm has no way of knowing that it is sailing around a traffic separation scheme when it should be following it. This is demonstrated in figure 5.7. This traffic separation zone is placed between Cuba and the Bahamas. This separation zone is designed in such a way that there is a margin between the traffic separation zone and the dark blue area where the water is too shallow to sail. The light blue safe area has a depth of 182 to 1829 meters. This margin makes it possible to make a route around the traffic separation zone, without conflicting with any features or specifications by the International Maritime Organization. The path is unable to make a route on the inside of the traffic separation zone as this area is marked as an inshore traffic area.

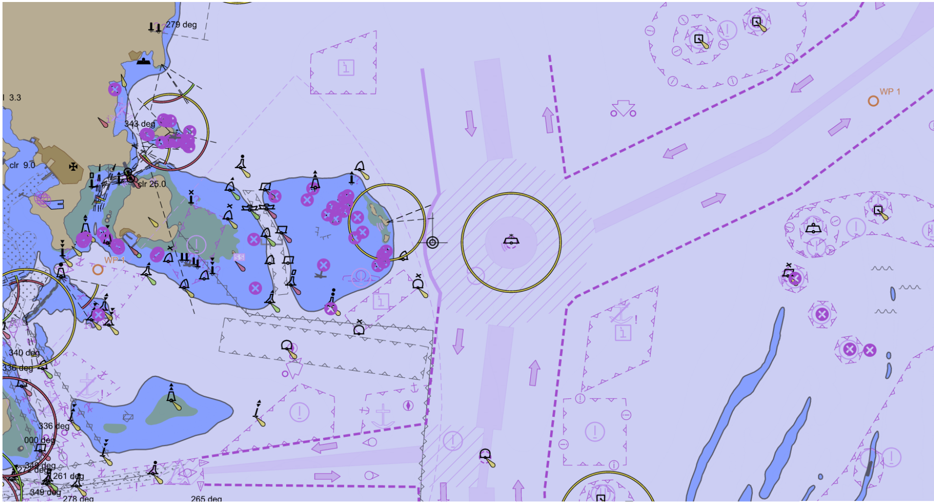



Figure 5.3: An area with many traffic routing features. The dot on the left is the start point of the wanted route, the point on the right is the end point.

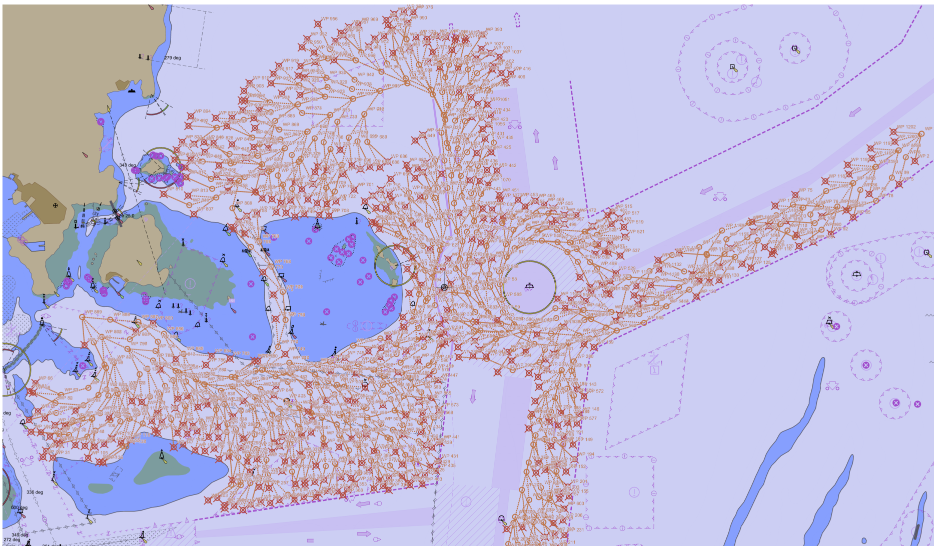


Figure 5.4: All the vertices explored by the forward and backwards tree. The crosses are leaf nodes that did not lead to any further nodes.

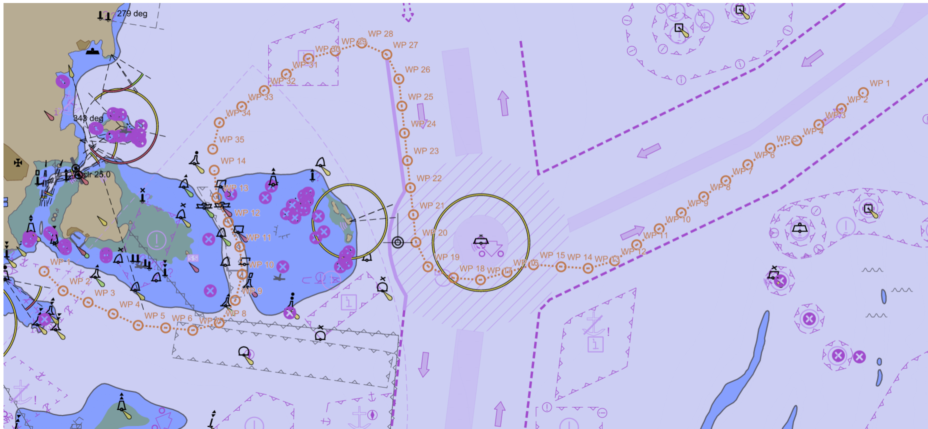


Figure 5.5: The path that ended up being chosen from the forward and backward search tree. The paths are not connected and post processed yet.

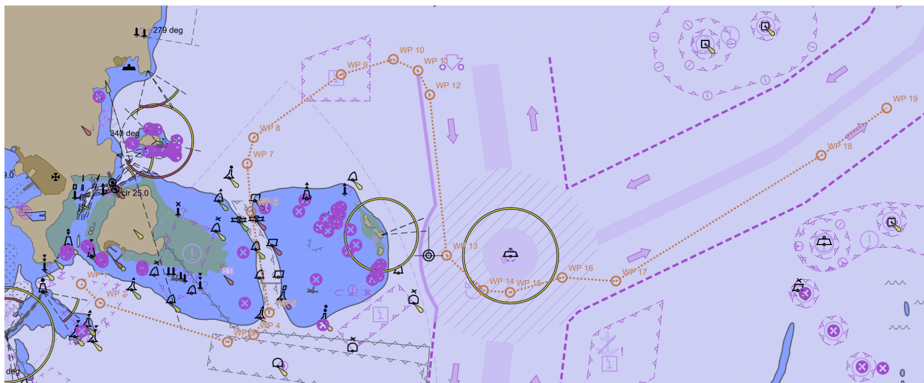


Figure 5.6: The final path after the two paths are connected and post-processing is done.

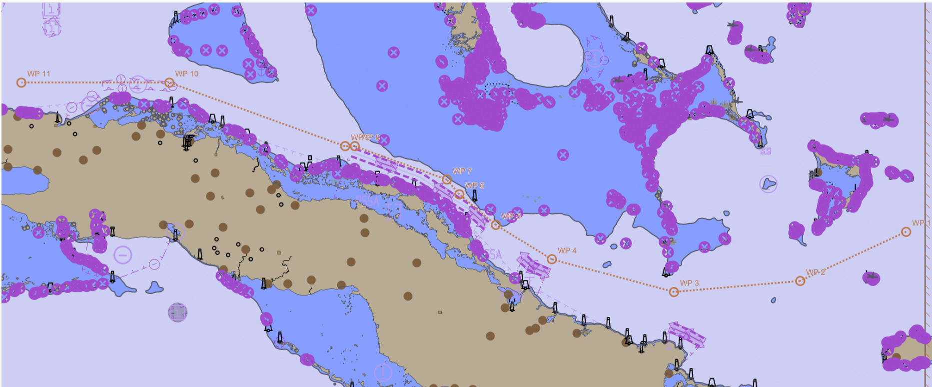



Figure 5.7: Example of another route that arises in an area with traffic routing features north of Cuba.

Figure 5.7 was done with the following settings.

- time-step = 1200s
- margin to obstacles = 800m
- grid cell size = 0.03° × 0.03°

## 5.3 Monodirectional and bidirectional comparison

This section is done with the following parameters.

- time-step = 200s
- margin to obstacles = 20m
- grid cell size = 0.004° × 0.004°

Table 5.1 show that a bidirectional search is clearly faster than a monodirectional search for an example without weather constraints. The search trees that show all explored vertices is presented for the monodirectional case in figure 5.8 and the bidirectional case in figure 5.9.

| | Number of times the chart is accessed |
|---|---|
| Monodirectional | 3226 |
| Bidirectional | 1834 |

Table 5.1: Comparison of a monodirectional (figure 5.8) and bidirectional (figure 5.9) search.

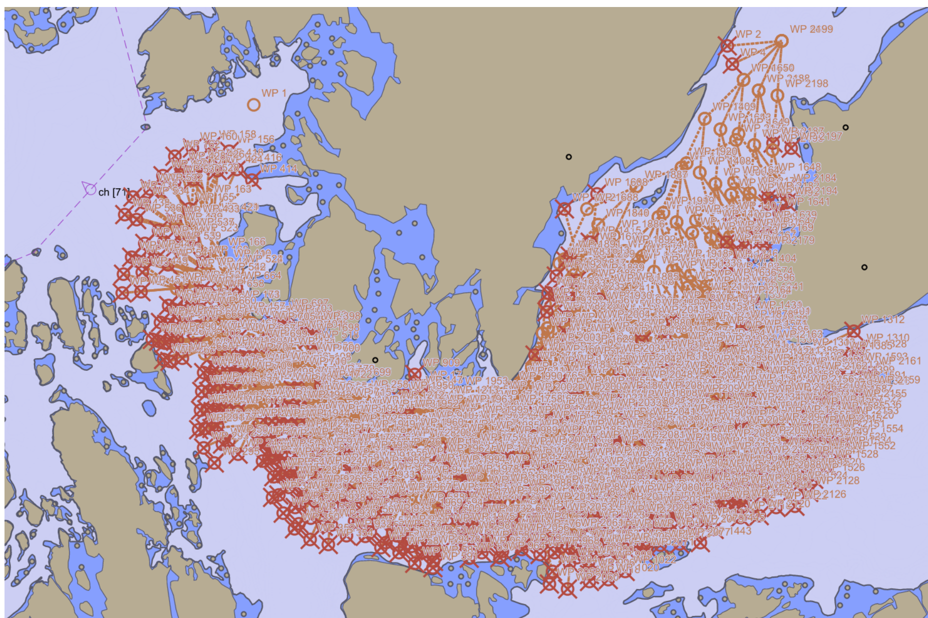



Figure 5.8: The search tree with a monodirectional search.

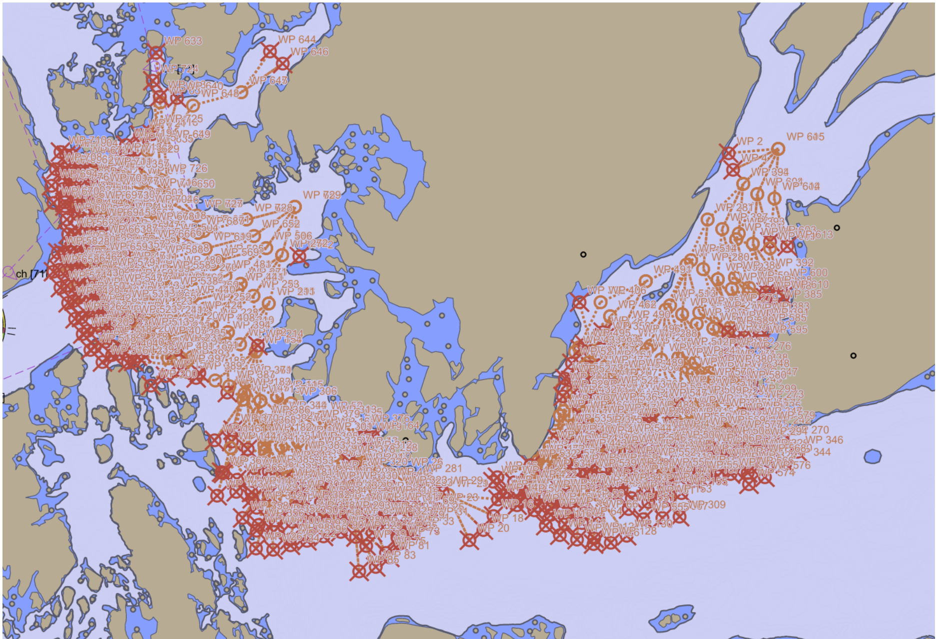



Figure 5.9: The search tree with a bidirectional search.

## 5.4 Weather constraints

This section is done with the following parameters.

- time-step = 200s
- margin to obstacles = 20m
- grid cell size = 0.008° × 0.008°

Figures 5.10 to 5.14 show how the path is gradually made when there is a weather constraint present. Figure 5.10 show the state of the two trees shortly after thy first found a connection. Figure 5.11a and figure 5.11b show the same graph, but with the backwards tree and forwards tree in separate plot. This is to make it easier to distinguish which vertices that belong to the different search trees. The red squares in the plot show the area that is marked as having a dangerous weather condition. In this example the weather is constant. The backwards search tree has expanded its graph through the dangerous area, as the backwards search tree does not know if the weather will still be present when the ship reaches that position. The backwards search tree has then expanded with illegal vertices into the area where the forward search should go, filling up most of the grid cells in that area.

When the forward search tree gets close to illegal parts of the backwards search tree, it will reject those vertices. When a rejected vertex is in the way of the forward search it will be deleted, together with all of its successor vertices. Figure 5.12 (decomposed in figures 5.13a and 5.13b), shows that large parts of the backwards search tree that was illegal has been deleted. The remaining illegal part of the backwards search tree is difficult for the forward search tree to connect to, as the backwards search tree has no more vertices that goes towards the forward tree. Most of the vertices in front of the forward tree go up and to the left. As the forward tree will have problem connecting with the backwards tree, it is important that the forward tree can reject all vertexes within a radius, not only those it can connect to. This was discussed in section 4.5.2.

The final path that is found from the search in figures 5.10 and 5.12 is shown in figure 5.14. This path avoid the dangerous weather conditions. The route was unable to find a path through the small channel right left of the dangerous area as the search had too long time step size and too limited heading change constraint to find a legal path from the backwards search start point to this channel. The channel on the right

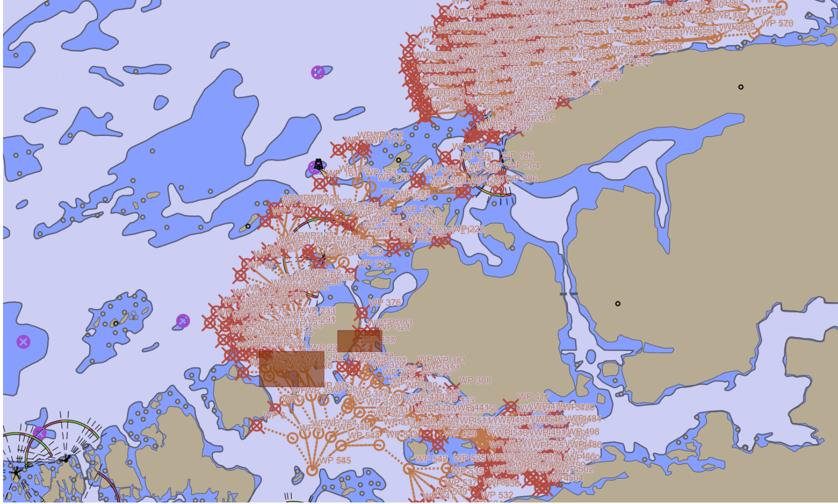
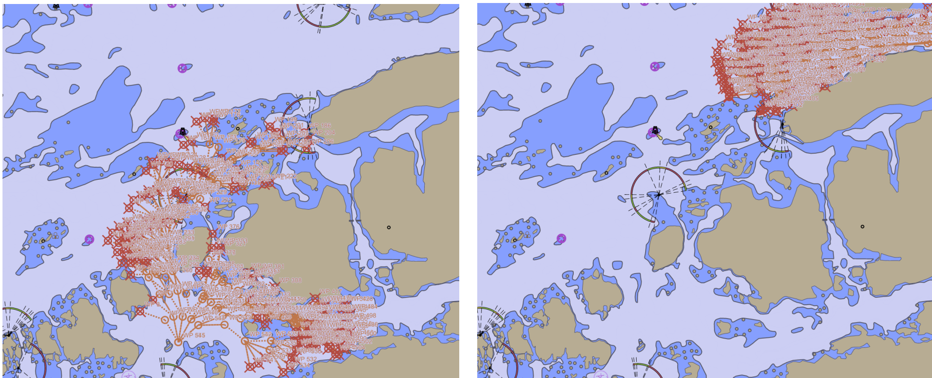


Figure 5.10: The search trees shortly after they they first found a connection. The bottom most tree is the backwards search tree, the one from top right is the forward search tree. The red marked areas are areas with illegal weather conditions.



(a) Backward search tree

(b) Forward search tree

Figure 5.11: The search trees from 5.10 show separately.

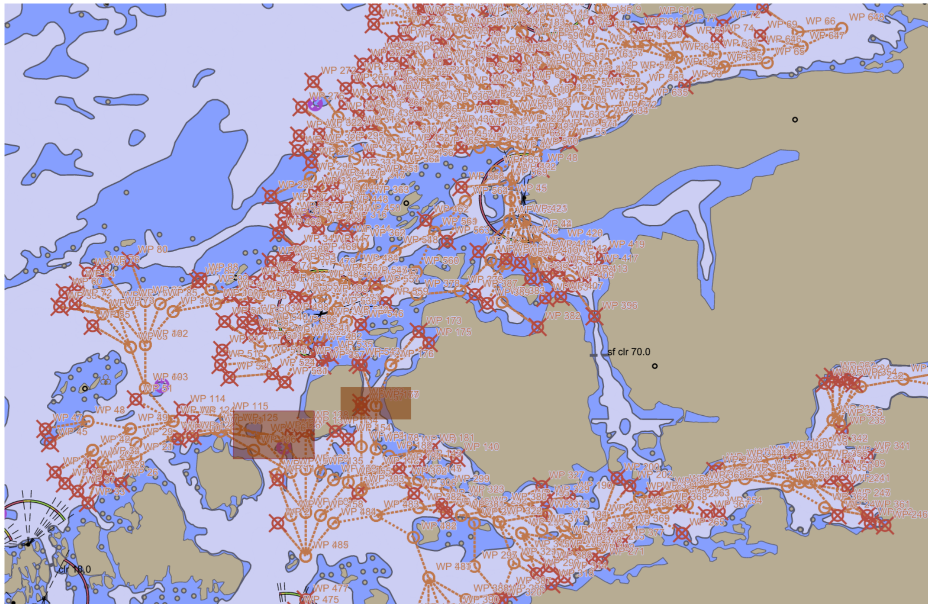
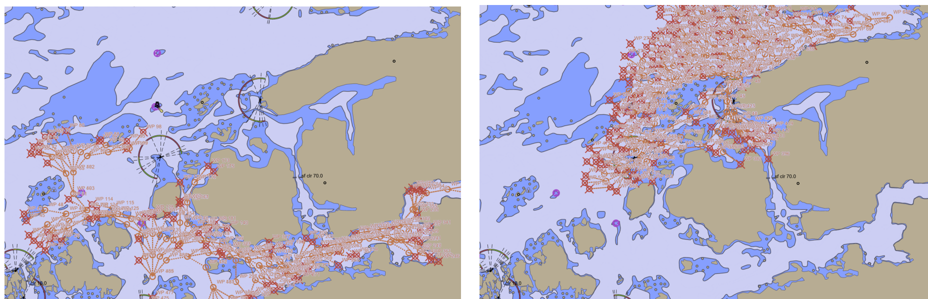



Figure 5.12: The forward search will reject all illegal backwards search vertices close to it. Rejected vertices that are in the way of the forward search are deleted. The red areas are areas with dangerous weather conditions.



(a) Backward search tree

(b) Forward search tree

Figure 5.13: The search trees from 5.12 show separately.

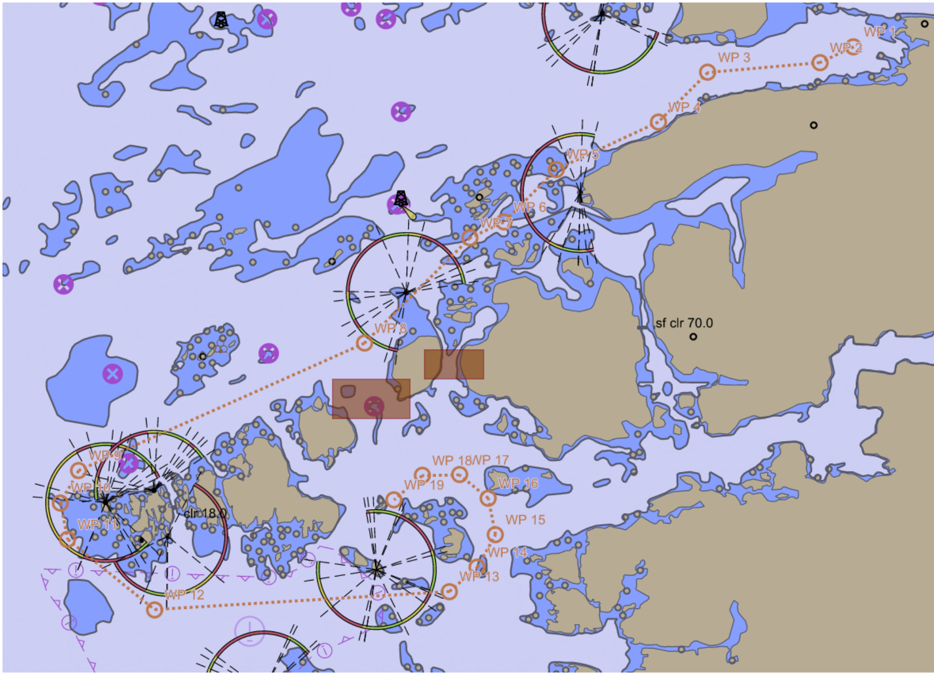



Figure 5.14: The final route that was found with weather constraints.

of the dangerous weather was also not used as the time-step size was to long to be able to find a legal path through it. Outside of these two options, the path managed to find a close to optimal route with weather constraints. The path does not take any large unnecessary detours.

Figure 5.15 shows that the path planner is able to find paths that can wait out weather. In this case the start and end points are switched, such that the start point is close to the illegal weather conditions. The illegal weather conditions will disappear after 2000s, and the forward search tree is allowed to wait for up to 5000s with 1000s intervals. The figure show the forward and backwards search path before they are connected and post-processed. Here the forward path (one point) starts at waypoint 3. This is since the forward search waits at this point for two time-steps, making the 3 first waypoint share the same coordinates. After waiting two 1000s intervals at the start-point, the weather will become safe, and ship can follow the path that the backward search found.

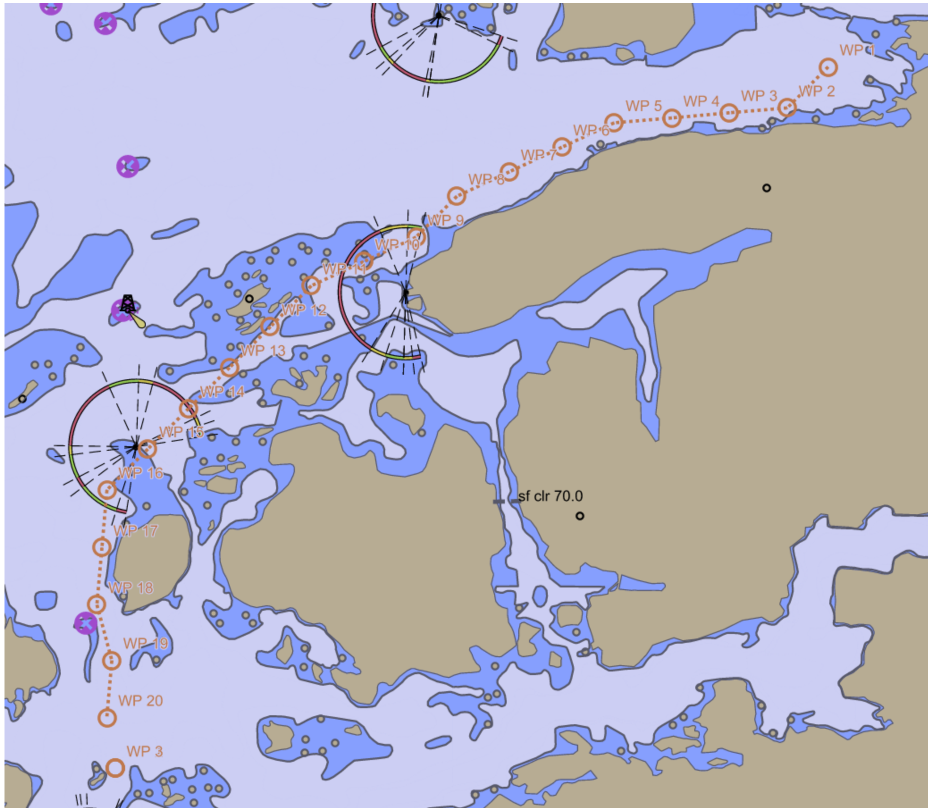



Figure 5.15: A path that waits at the initial vertex until the weather improves. The start vertex was placed at the bottom, and the end vertex on the top-right. The same areas are marked with dangerous weather as in figure 5.15, but after 2000s the weather is no longer dangerous. The forward and backward paths are shown separately without post-processing.

## 5.5 Weather bidirectional comparison

This section will repeat the search presented in figure 5.14 but with a shorter time-step. The previous section had a longer time-step to make it easier to present visually. This section will have a shorter time step to ensure that the search trees manages to find more viable paths. With the longer time-step the the monodirectional search ended up finding different paths than the bidirectional (but had to break with the heading change constraint to be able to connect to the goal point). To better compare the performance the time-step was reduced such that both strategies found approximately the same route. A new weather constraint was added since the shorter time steps lets the path go through the channel on the right. Going through this channel avoids the problem that the backwards search is in the way of the forwards search.

The following parameter where then chosen.

- time-step = 100s
- margin to obstacles = 20m
- grid cell size = 0.004° × 0.004°

The bidirectional search in figure 5.16 has a detour compared to the monodirectional search in figure 5.17. This is an artifact that can occur when parts of the backward search are deleted, as the remaining tree may lack vertices going in some directions. This can be seen in figure 5.18. The empty area in the middle was previously covered by illegal backward search vertexes. The legal part of the backward search was unable to search in that direction as all the grid cells were already occupied. After the illegal parts were deleted, the remaining part was unable to search in that direction. This lead to a connection being made that contained a small detour.

This test was designed to be close to the worst case scenario for a bidirectional search with weather constraints. The weather constraints were placed really close to the backwards search, making large parts of the search done by the backwards search illegal. The backwards search is also blocking the path of the forward search. Table 5.2 shows that it in this case was slower to use a backwards search as most of the backwards search vertices had to be deleted. Figures 5.16 and 5.17 shows that a bidirectional search with weather constraint may lead to sub-optimal routes.

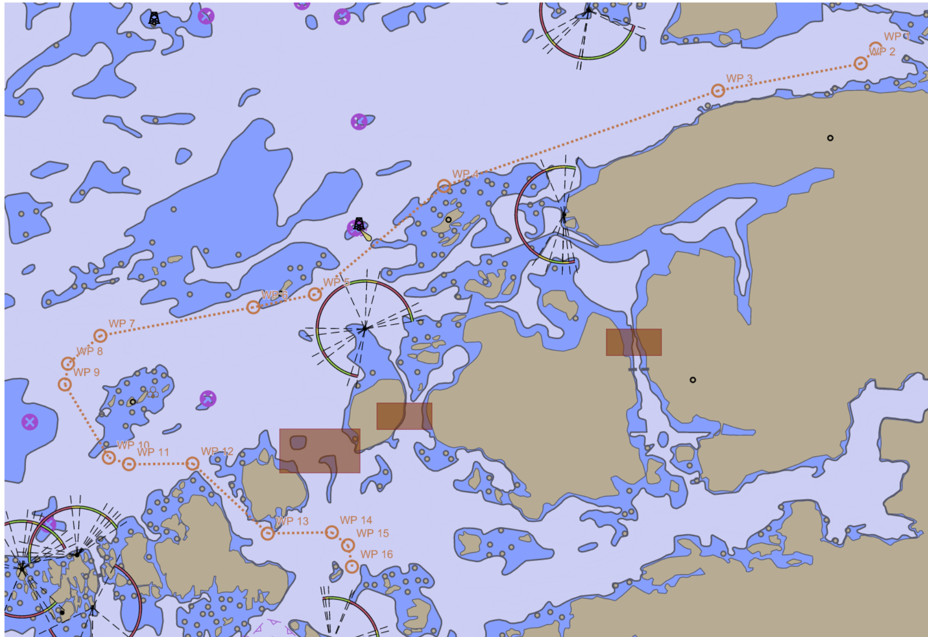



Figure 5.16: The final path with a bidirectional search for a close to worst-case setting.

| | Number of times the chart is accessed |
|---|---|
| Bidirectional search (fig. 5.16) | 22837 |
| Monodirectional search (fig. 5.17) | 10121 |

Table 5.2: Comparison of the number of times the chart is accessed in the bidirectional monodoirectional path with weather constraints.

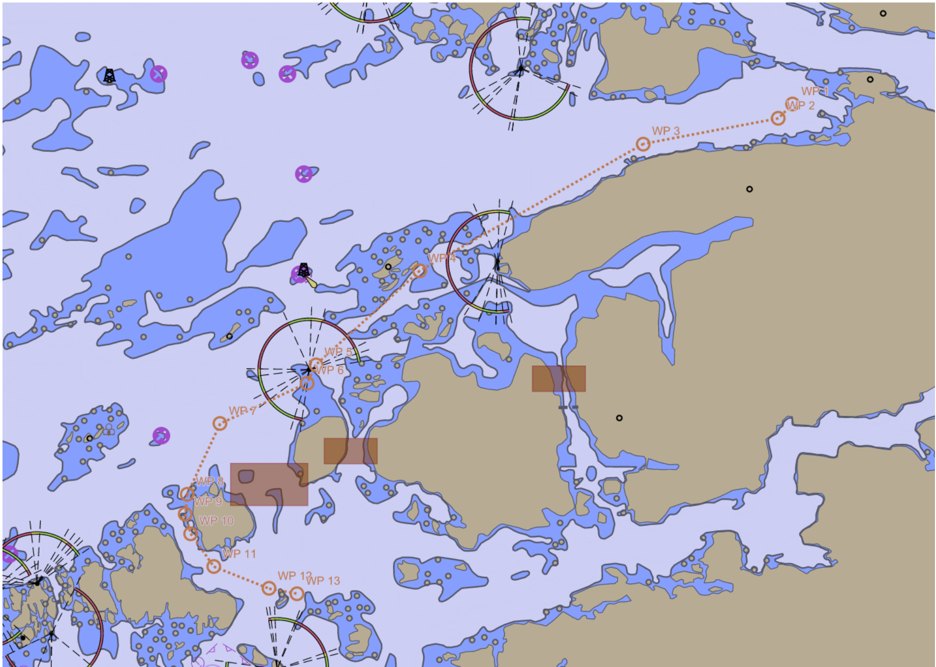



Figure 5.17: The final path with a mondirectional search for a close to worst-case setting.

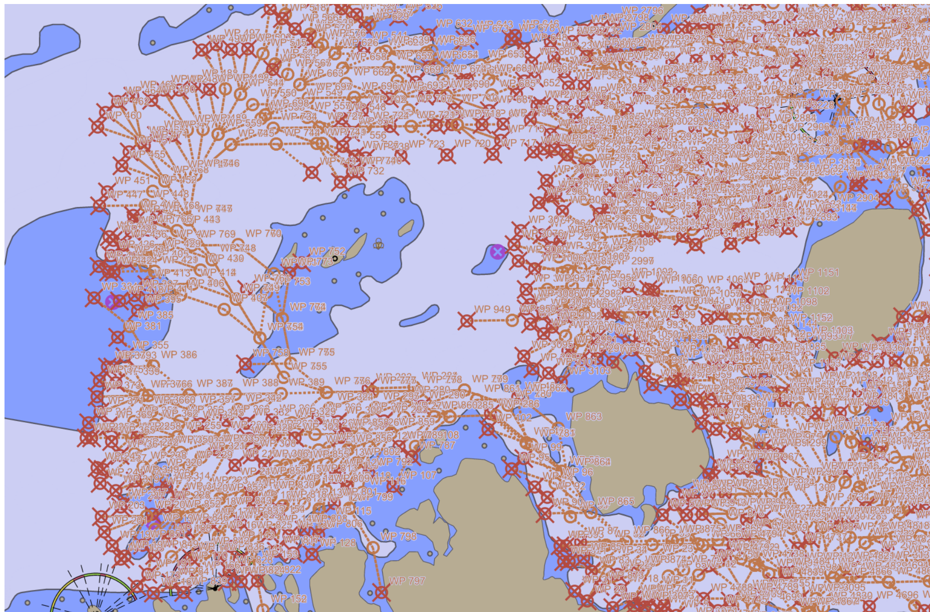


Figure 5.18: The search trees that led to figure 5.16.

A similar test is done in figures 5.19 and 5.20. Here the endpoint is placed further away from the area with illegal weather conditions, making this case closer to an average case than a worst case. In this case the number of time the chart is accessed in the bidirectional search is slightly lower than in the monodirectional case.

Dangerous weather conditions are rare, so using a bidirectional search might be preferable even though it performs worse than the monodirectional search in worst case situations.

This test was done with the following parameters

- time-step = 200s
- margin to obstacles = 100m
- grid cell size = 0.005° × 0.005°

| | Number of times the chart is accessed |
|---|---|
| Bidirectional search (fig. 5.19) | 14340 |
| Monodirectional search (fig. 5.20) | 15727 |

Table 5.3: Another comparison of the number of times the chart is accessed in the bidirectional monodirectional case.

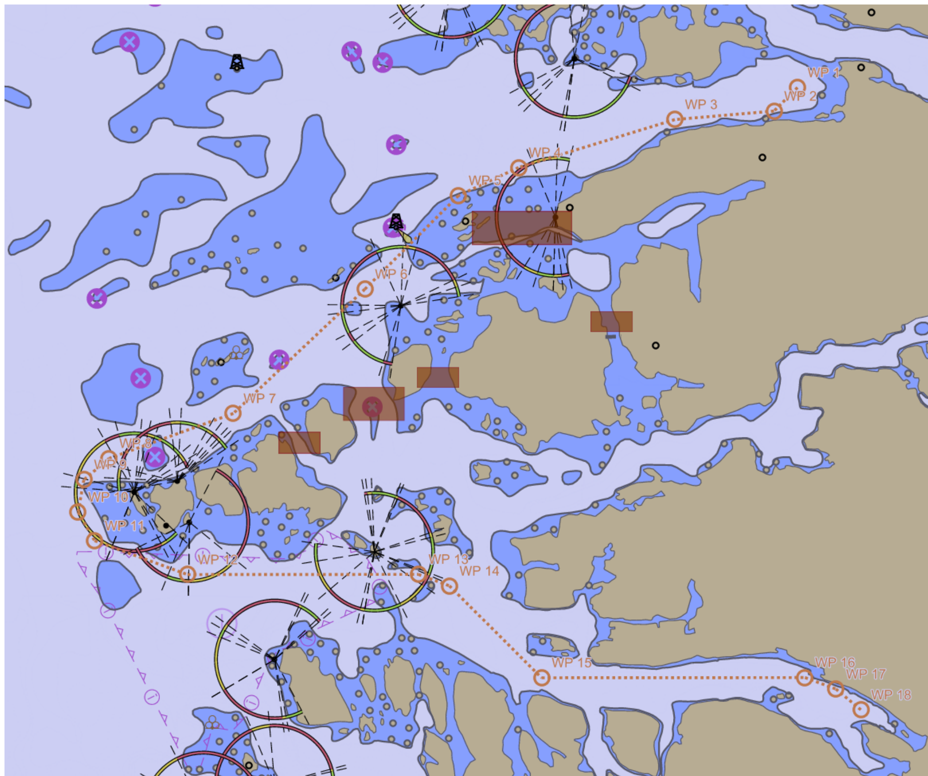



Figure 5.19: The final path with a bidirectional search for a more average case setting.

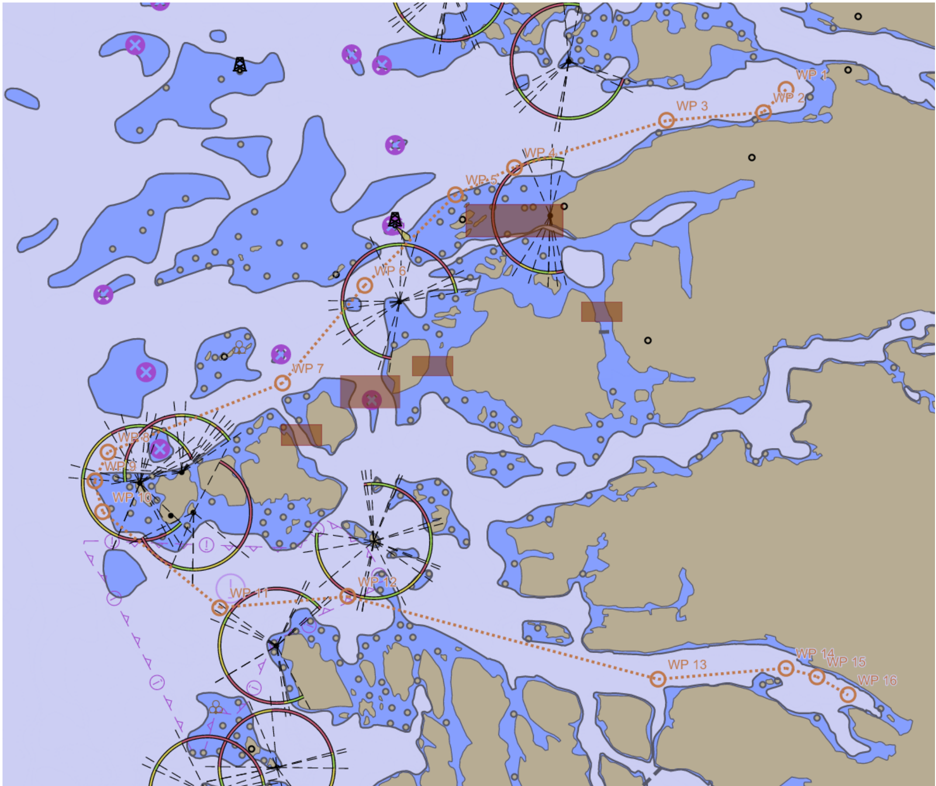



Figure 5.20: The final path with a mondirectional search for a more average case setting.

## 5.6 Simulation results

The route in figure 5.21 was simulated on Marine Technologies (MT) vessel bridge system which is shown in figure 5.22. This bridge system is normally used as a training simulator to train dynamic positioning operators. A simulator was used to feed the bridge system with simulated vessel sensor data. The autopilot of the bridge system controlled the simulated vessel with respect to the waypoints from the route. The simulation was done in real time. Figure 5.23 shows the ship some time after departure, where the black line shows the path the ship has followed. The ship managed to keep inside the safety margins shown as thin dotted lines on each side of the path. The safety margin in the path planning process should have been increased to ensure a larger safety margin before waypoint 19. Nevertheless, the simulated ship managed to successfully sail the entirety of the path without grounding or hitting any obstacles.

The full feedback from Marine Technologies after they simulated the route in figure 5.21 can be read in apendix A.

The route in figure 5.21 had a slightly older version of the post-processing algorithm than the one presented in this thesis. That version of the post-processing algorithm did only attempt to remove single waypoints at a time.

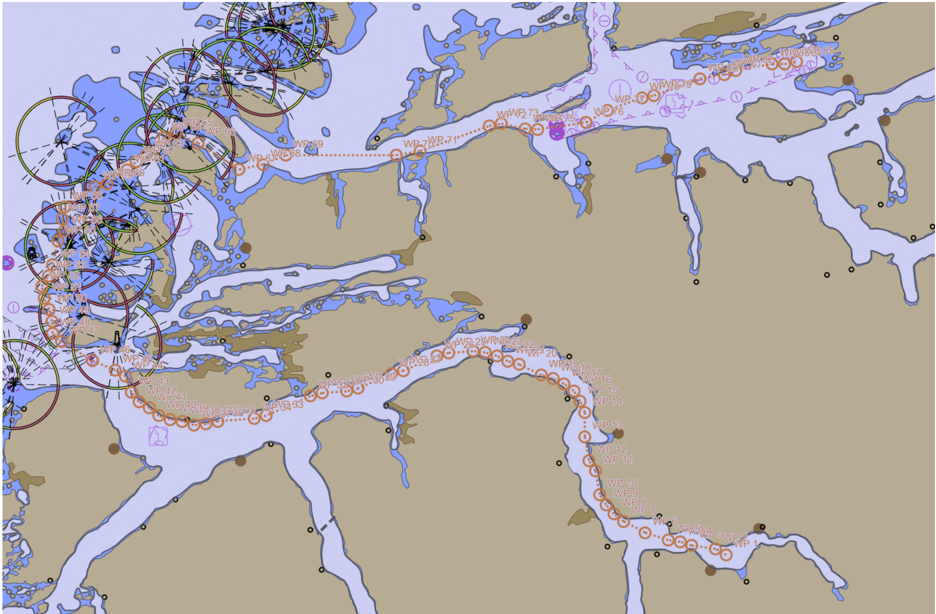


Figure 5.21: A route between Valldal and Røvika.

Figure 5.22: The bridge system at Marine Technologies

Figure 5.23: A screenshot of the simulated ship successfully following the route from figure 5.21.

# Chapter 6

# Limitations and improvements

## 6.1 Computational time

The main problem with this algorithm is the long computational time used to plan a route. The route between two Norwegian fjords presented in figure 5.1 was by far the most complicated route that was calculated. The narrow fjords and complicated island structures along the Norwegian coast forced the time-step to be very short relative to the length of the path. This route required 159187 chart accesses, which took 734 minutes, or 12.2 hours to calculate. In the program 92% of the computational time was used inside the get features function. This results in the get feature function using 0.3 seconds to complete. The other tests used substantially shorter time as the number of chart accesses were substantially fewer. The traffic separation scheme test from figure 5.6 used only 2 minutes and 20 seconds to finish and accessed the chart 4611 times. This results in only using 0.026 seconds to run the get feature function once. The route found north of Cuba in figure 5.7 used 29 minutes and 20 seconds to access the chart 18859 times, resulting in 0.093 seconds per chart access. These tests used substantially shorter time to access the chart than the long distance test between two Norwegian fjords. This difference comes from the fact that the get feature function tests all features within the current chart to see if they overlap the test geometry. The test chart used in the traffic separation scheme example and the chart over Cuba were significantly smaller than the chart over the west coast of Norway. Since they were smaller they contained fewer features making the algorithm finish in

significantly shorter time. Altering the way the get feature function works such that it does not need to test all the features of the entire chart, could substantially reduce the computational time. If it managed to reduce the get feature computational time to the 0.026 seconds that was used in the example chart, then the path between the two Norwegian fjords could be done in 70 minutes.

These calculations were done on a computer running Windows 10 with an Intel(R) Core(TM) i7-7700 CPU running at 3.6 GHz with 32 GB RAM and using the Intel(R) HD Graphics 630 graphics card.

## 6.2 Variable time-step

One essential step for making a path planner that can work between any two places on the earth is to dynamically change the time step depending on how narrow or open the area is. In narrow areas such as small fjords and close to land the time step must be small to be able to circumvent obstacles. Over open areas such as the Atlantic ocean the time step must be longer to be able to get a path in reasonable time.

One strategy for changing the time-step between these two extremes would be to make a grid consisting of cells larger than the longest time-step. All grid cells that contains an obstacle should use the short time-step, and all empty grid cells should use the long time-step. Empty cells bordering a cell containing an obstacle could have a medium long time-step. This strategy was not tested out as I was unable to acquire a navigational chart spanning large open-water areas.

Another strategy is needed to modify the step-length between small and larger canals or fjords. One possibility would be to use the distance to the closest obstacle to calculate the time-step. This is done in Dolgov et al. (2010). To be able to use this strategy an efficient way of finding the closest obstacle has to be developed.

## 6.3 Placement inside channels

Another limitation with the current algorithm is that it does not attempt to place the ship further away from obstacles when possible. It will guarantee that path is further away than the safety margin from the obstacles, but will not try to place the path further away from the obstacle when there is a lot of safe waters. This could be done by figuring out how far away land is and then giving a penalty that decreases with the

distance away from land. This would in the same manner as for finding a variable time-step require an efficient way of finding the closest obstacle.

Giving a penalty that decreases with distance will make it difficult to find route through narrow but safe areas, as these areas will have a high penalty. If this is unwanted behaviour, then a Voronoi diagram can be made in a similar manner to the one discussed in section 3.1.3. The penalty can then be designed to be at a maximum by land, and decrease to zero at the edge of the Voronoi diagram that is in the middle of the channel. This has been done in Dolgov et al. (2010).

Placing the path in the middle of the channel gives the maximum clearance from land, but may make dangerous traffic situations in narrow channels as it might make the ship difficult to pass. COLREGS section A rule 9.a dictates that a ship should keep to the starboard side of narrow channels and fairways (IMO, 1996). Keeping close to safety margin from the starboard side of channels can therefore be preferable.

## 6.4 Variable speed

The current algorithm is designed with constant speed throughout the entire voyage. This speed is chosen to be the designed cruise speed of the vessel. In some circumstances such as close to land or in areas with speed restrictions, the speed should be changed. The way the path planner in this thesis is designed makes it straightforward to change the cruise speed. Each vertex has an associated state, containing the current speed. This value can be changed for vertices that are inside areas with speed restriction or close to land.

Another improvement that can be done is to incorporate the current the ship is exposed to. The current velocity can then be added to the relative velocity to find the actual velocity.

The algorithm will try to find the path that takes the shortest time, and minimizes any penalties that are introduced. Sailing in areas where there is a speed restriction or against the current will take more time, the algorithm will therefore try to sail around these areas.

## 6.5 Tide

As the algorithm supports time-dependent constraints, it is straightforward to make the algorithm handle tide. The navigational chart could use the time when testing the grounding constraints, by using the sea level at that time-point. The forward search could then use the time dependant chart with tides, while the backwards search could use the best case scenario when there was high tide. The forward search would then in the same manner as with dangerous weather condition reject backwards search nodes that break with tide constraints.

## 6.6 Traffic routing objects

The algorithm is able to make legal routes within traffic separation schemes. But as the algorithm interprets the chart literally, it may abuse every imprecise aspect of the chart. The path planner can therefore find a path that goes just outside of the traffic separation scheme instead of following it. An example of this was shown for the traffic separation zone north of Cuba in figure 5.7. This problem will not arise when there is an inshore traffic area on both sides of the traffic separation zone as in the case of the Dover Strait in figure 4.5. This problem is especially prevalent when the traffic routing schemes are marked as lines or points instead of areas. Only edges that randomly overlap these areas will take their information into account.

As the algorithm is now, it has no way of knowing that there is a traffic routing scheme close by it should rather use. One solution which would hinder the path from just missing traffic routing schemes, would be to check a larger area around each edge of the graph. If the area contains a traffic routing scheme and the ship is perpendicular to any side of the routing scheme, then the position could be marked as illegal and deleted. The ship must be within the start and end points of the routing scheme, and not inside the scheme itself. This was not done as this would at least require one more chart access, which would double the computational time required to solve the problem.

Another problem with interpreting the chart literally is when the routing schemes are placed in a manner that requires some human intuition. One example of this can be seen in figure 6.1. In this traffic routing system the separation lanes do not touch the roundabout. The area marked as a roundabout is highlighted in orange. The

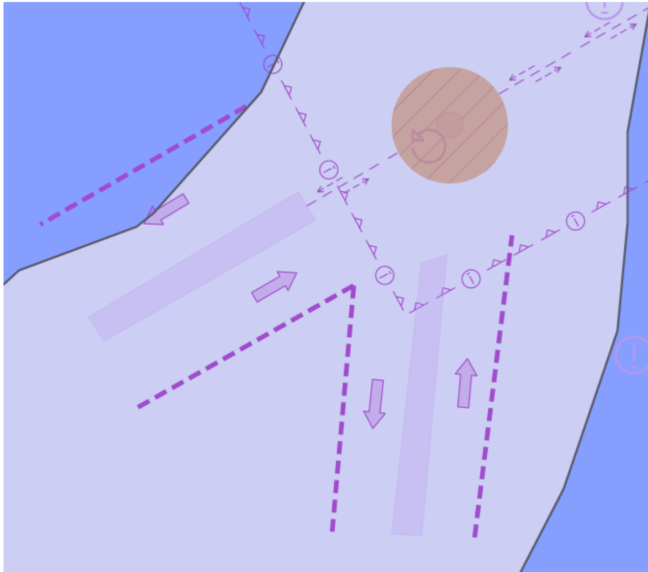


Figure 6.1: An example of a roundabouts feature where the traffic routing features are not touching each other.

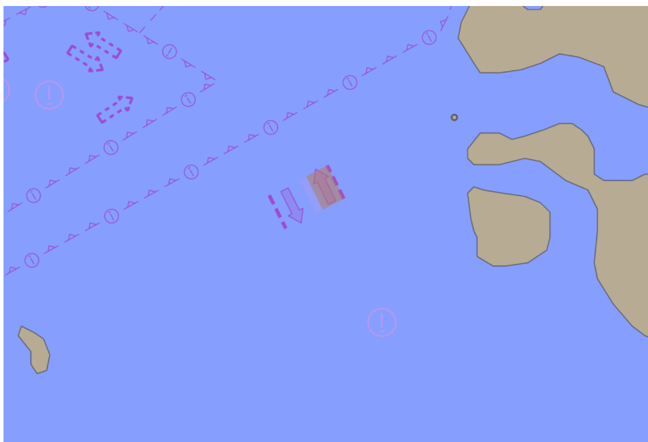


Figure 6.2: An odd small traffic separation area.

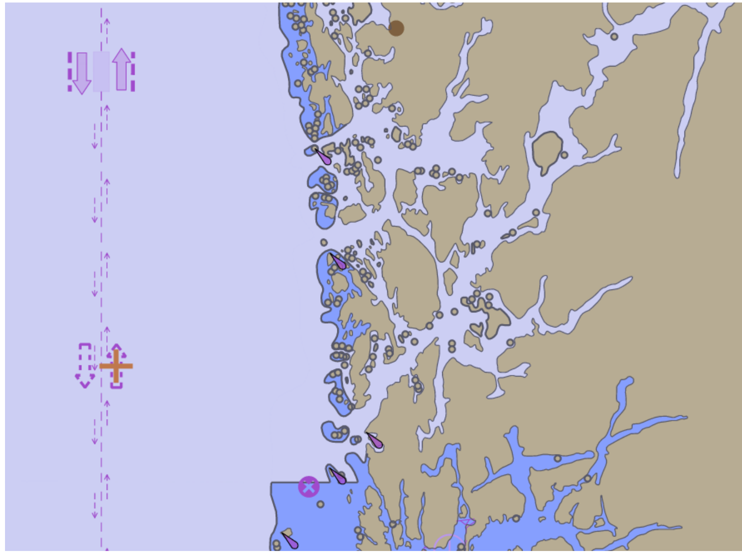


Figure 6.3: A recommended route along the coast of Norway that requires some human intuition.

ship can therefore sail around the roundabout in the opposite direction, as long as it keeps far enough away to not be inside the area marked as a roundabout. Another example is in figure 6.2 where there is a tiny traffic separation zone in the middle of the channel. This scheme is so small that the route can simply sail around it. Even if the route was forced to go through it by using inshore traffic areas, it could ignore the direction the moment it left the short separation zone.

It seems like a lot of the features on the mapped are placed there to give a visual aid for a human operators. They are not placed in a manner that makes navigation unambiguous and consistent. This can be seen in the example of the traffic separation scheme along the Norwegian coast in figure 6.3. The traffic routing elements on the left side of the picture are supposed to mark areas where traffic along the Norwegian coast should go. Instead of using a full traffic separation area, the recommended track consists of a areas, lines, and points. Close to the top of the figure is a full traffic separation zone marked with two lanes and a separation area in between. This area is unambiguous. Instead of extending this area all the way, it is replaced by a recommended route centerline which is only a line (the dotted line

with thin dotted arrows on the sides). How far the recommended routes extends from the centerline is not defined. Further down the route are thicker recommended route arrows. These are encoded as single points. The rightmost arrow is highlighted with a cross, where the center of the cross defines the position of the arrow. This arrow seems to give no new information on how the ship should interact with this traffic routing scheme, and seems to only be a visual aid for a human operator using the chart.

## 6.7 Improving navigational maps for automation

The way the navigational maps are designed today is difficult to use for automation. They can easily be changed to make automation at sea a lot easier. The charts have to be designed in such a manner that they can be interpreted literally. Firstly the traffic routing schemes need to be placed in such a manner that the different features touch. This makes it impossible to make a legal routes that go against the planned traffic flow. This would consist of making the roundabouts in the manner shown in figure 5.3 instead of 6.1. The traffic separation scheme must also fill the entire area where there is a planned traffic flow. Areas such as the minimalistic traffic routing scheme in Norway, figure 6.2, and the small areas on the bottom right of the path north of Cuba, figure 5.7, must be extended to fill the entire channel where the specified traffic flow is intended. To make it simpler to comply with roundabouts, they should have an attribute defining their center point. This would make it straightforward to check if the route follows the roundabout, as the center should always be on the left side of the ship.

For traffic routing schemes inside channels the inshore traffic area feature must be used on both sides of the traffic routing scheme making it impossible for the route to sail just outside of it. A similar feature should be marked on the outside of traffic routing schemes that only have land on one side, such as the one in Cuba, figure 5.7. This feature should mark the distance away from the routing scheme where the routing scheme should rather be used than sailing around it. Something similar should also be included around the traffic routing scheme that goes up along the Norwegian coast, shown in figure 6.3. Her the feature should give a signal that routes going a longer distance along the Norwegian cost should rather use this scheme than sailing slightly closer or further away from land. As it is now, a search algorithm would

not know that there is such a feature that can be used, unless it randomly stumbles upon it.

The traffic routing scheme should only consist of areas, and not lines or points, that mark where the ship should sail. Lines are ambiguous as how far they extend to the sides are undefined. Points should also be avoided as the algorithm has no way of knowing how long this recommendation should be followed.

# Chapter 7

# Concluding remarks

## 7.1 Summary

Figure 5.1 shows that the path planner presented in this thesis manages to make a safe path with respect to grounding constraints for longer paths going from one fjord in Norway to another. The path is close to optimal, and has few waypoints making it easier to follow and more predictable. The path avoids obstacles within the given margin but does not attempt to have a larger clearance when this is possible.

Figure 5.4 shows that the path planner manages to make legal routes within consistent traffic routing schemes. But there is a danger that the path planner rather finds a way outside the traffic routing scheme, as can be seen in figure 5.7. This problem arises as the area just outside traffic separation zones have no special marking showing that the route should rather use the traffic routeing scheme.

Figures 5.9 and figure 5.8 demonstrate that using a bidirectional search is definitive faster in the static case where there are no time-dependent weather constraints. This is no longer clear with time-dependent weather constraints. The bidirectional search was substantially worse than a monodirectional search in the example that was designed to be close to worst case which was shown in figure 5.16. In a more average case, as shown in figure 5.19, using a bidirectional search was slightly quicker. Dangerous weather conditions are rare, so most routes that are going to be planned will probably not be affected by the weather conditions. Using a bidirectional search will therefore probably do more good than harm.

The path produced can become less optimal with weather constraints, as deleting part of the backwards tree may leave behind a substantially reduced tree which can make sub-optimal routes. Even though the path can be less optimal with weather constraints, it is still guaranteed to be safe.

A path planned with the algorithm presented in this thesis is shown to work on the bridge system of Marine Technologies. A simulated ship managed to follow the path without grounding or hitting any obstacles.

## 7.2 Conclusion

The goal of this thesis was to make a path planner that could work as a first step towards a fully automatic ship path planner. The thesis has successfully made a planner that manages to make a safe path with respect to grounding and weather constraints that is kinetically feasible for the ship to follow. This was the main goal of the thesis, as making a safe path is the most important task of a path planner. Even though the algorithm has some shortcomings, the chosen strategy seems to have potential to solve these problems. Many of the problems are a result of the long computational time needed to accessing the chart, when this problem is solved many of the proposed improvements in chapter 6 can be implemented. Some of the shortcomings are consequences of the way navigational charts are designed. These problems will to different degrees be problematic for all ship path planners using navigational chart. This thesis outlined some ideas on how navigational charts could be altered to make automation substantially simpler.

Compared to the commercial solutions that are available today, some of which were presented in section 3.5, this algorithm guarantees a much higher degree of safety, but is substantially slower. This algorithm guarantees that the path is further away from land and all other dangerous obstacles that are marked on the navigational charts than the safety margin. This is the highest safety concern, which none of the commercial solutions managed to uphold.

The algorithm guarantees that the finished route will not be within dangerous weather conditions. The algorithm is able to make a path that sails around dangerous weather conditions, or waits them out at the initial port. The tests in this algorithm were artificially constructed and only done for small areas. Larger-scale tests with realistic weather data should be done before anything can be concluded.

The chosen model has potential to take more data into account, which lets it make more optimal routes. It is straightforward to implement a better ship model which takes current and wind into consideration. The objective of the algorithm can then simply be changed to minimize energy consumption. Optimizing routes with respect to current and wind can improve the fuel-oil consumption of the journey (Kobayashi et al., 2015). It is also straightforward to extend the algorithm to include tides.

## 7.3 Applications

The algorithm presented in this thesis can be used as a assisting tool for human operators making navigational charts. A human operator could use this algorithm to make a path and then check the path manually to see if it is acceptable, potentially doing the small changes that are needed.

The algorithm that was designed to handle time-varying weather constraints can be used in the general case of bidirectional search with time-varying constraints. The developed algorithm was designed to work best with slowly changing constraints, and a graph that is procedurally generated. Procedurally generated graphs can be used in all sorts of path planning problems, such as for underwater drones, areal drones, or land vehicles.

This algorithm can be used as a basis for a full-fledged ship path planner which can take more data into consideration.

# Appendix A

# Feedback from Marine Technologies

The following is feedback from Marine Technologies after they simulated the route in figure 5.21 on their bridge system.

> Marine Technologies LLC (MT) is a U.S.-based company providing vessel control solutions to the international offshore and commercial shipping industries. MT's products include dynamic positioning systems, integrated bridge systems and VSAT communications. The company is headquartered in Mandeville, Louisiana, with offices in Norway, Singapore and Brazil.
>
> MT received a generated path/route from the presented algorithm in this thesis. The waypoints from the route was imported into ECDIS, running on an MT vessel bridge (see figure 5.22), where a simulator was feeding simulated vessel sensor data into the bridge system. The simulator used in this test is an actual training simulator, utilized in training of dynamic positioning operators (DPOs).
>
> After loading the waypoints, the control system was configured to follow the route in real time. It was observed that the vessel followed the autogenerated path without any grounding or collision with land (see figure 5.23). MT believe the presented algorithm is a big step in the right

direction for generating paths/routes automatically, and is impressed with the results achieved by the student in the short amount of time given.

Tor Eirik Østrem, M.Sc. Manager, Sensor Systems

# Bibliography

Bitar, G. I. (2017). Towards the Development of Autonomous Ferries. Master's thesis, Norwegian University of Science and Technology.

Candeloro, M., Lekkas, A. M., and Sørensen, A. J. (2017). A Voronoi-diagram-based dynamic path-planning system for underactuated marine vessels. *Control Engineering Practice*, 61:41–54.

Cormen, T. H., Leiserson, C. C. E., Rivest, R. R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition.* Massachusetts Institute of Technology.

cplusplus.com (2017a). std::list. http://www.cplusplus.com/reference/list/list/. Accessed: 2018-05-19.

cplusplus.com (2017b). std::vector::insert. http://www.cplusplus.com/reference/vector/vector/insert/. Accessed: 2018-05-19.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Dolgov, D., Thrun, S., Montemerlo, M., and Diebel, J. (2010). Path planning for autonomous vehicles in unknown semi-structured environments. *International Journal of Robotics Research*, 29(5):485–501.

Fossen, T. I. (2011). *Handbook of Marine Craft Hydrodynamics and Motion Control.* Wiley.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

IMO (1996). *COLREGS - International Regulations for Preventing Collisions at Sea.* International Maritime Organization.

IMO (2018). Ships' routing. http://www.imo.org/en/OurWork/Safety/Navigation/Pages/ShipsRouteing.aspx. International Maritime Organization, Accessed: 2018-05-20.

Johansen, T. A., Perez, T., and Cristofaro, A. (2016). Ship Collision Avoidance and COLREGS Compliance Using Simulation-Based Control Behavior Selection With Predictive Hazard Assessment. *IEEE Transactions on Intelligent Transportation Systems*, 17(12):3407–3422.

Kavraki, L. E., Švestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580.

Kobayashi, E., Hashimoto, H., Taniguchi, Y., and Yoneda, S. (2015). Advanced optimized weather routing for an ocean-going vessel. In *2015 International Association of Institutes of Navigation World Congress, IAIN 2015 - Proceedings*.

Lavalle, S. M. (1998). Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical report, Iowa State Universiity.

MarineTraffic (2018). Voyage Planner. https://www.marinetraffic.com/no/voyage-planner. Accessed: 2018-05-25.

Maritime Data Systems (2018). SeaRoutes. https://www.searoutes.com. Accessed: 2018-05-25.

Nannicini, G., Delling, D., Schultes, D., and Liberti, L. (2012). Bidirectional A* search on time-dependent road networks. *Networks*, 59(2):240–251.

Pijls, W. and Post, H. (2009). Yet another bidirectional algorithm for shortest paths. Technical Report EI 2009-10, Erasmus University Rotterdam, Erasmus School of Economics (ESE), Econometric Institute.

Rantanen, M. (2014). *Improving Probabilistic Roadmap Methods for Fast Motion Planning*. PhD thesis, UNIVERSITY OF TAMPERE.

Russell, S. J. and Norvig, P. (2014). *Artificial Intelligence: A Modern Approach*. Pearson Education, 3 edition.

SeaFuture Inc. (2018). Netpas Distance. https://www.netpas.net/. Accessed: 2018-05-25.

Shah, B. C. and Gupta, S. K. (2016). Speeding up A* search on visibility graphs defined over quadtrees to enable long distance path planning for unmanned surface vehicles. In *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, volume 2016-Janua, pages 527–535.

Spong, M. W., Hutchinson, S., and Vidyasagar, M. (2005). *Robot Modeling and Control*. Wiley.

Ueland, E. S., Skjetne, R., and Dahl, A. R. (2017). Marine Autonomous Exploration Using a Lidar and SLAM.

Veness, C. (2017). Calculate distance, bearing and more between Latitude/Longitude points. https://www.movable-type.co.uk/scripts/latlong.html. Accessed: 2018-04-10.