NTNU

Norwegian University of
Science and Technology

Path Planning of Autonomous Swarm Tugboats

Rebecca Lillian Cox

Master of Science in Engineering and ICT
Submission date: June 2017
Supervisor: Martin Steinert, MTP
Co-supervisor: Kristoffer Slåttsveen, MTP

Norwegian University of Science and Technology
Department of Mechanical and Industrial Engineering

Summary

This research is part of a study of autonomous tugboats. The study is a cooperation between two students, but the projects are separate. The common goal for the study is to develop a working prototype of an autonomous tugboat swarm. In this part of the study, algorithms for planning the shortest paths for the tugboats have been developed.

A literature review has been conducted to find the best algorithms and framework for the path planning system. The environment has been simplified by using a road map approach called "visibility graph". The shortest path within this graph is found by using A* search algorithm. This will give the shortest path within the non-simplified environment, if one exist.

Algorithms from external libraries have been adjusted to fit the tugboat framework. Some additional algorithms have also been developed, for instance a new method to find paths around a ship in real-time, only knowing its dimensions, position and orientation.

In addition, a speed-up and memory saving version of the shortest path algorithm has been developed, by calculating all shortest paths to and from all points offline. The speed-up was measured in five different test environments, and was measured to be on average 12.3 times faster than ordinary A* search in a visibility graph. The speed-up becomes even more impressive as the complexity of the environment increases.

The path planning system and communication has been implemented using the "Robot Operating System". This system has been tested successfully on a physical test setup.

The physical test setup is a starting-point for future research of autonomous tugboats. With this setup, the development in the study can be conducted using a bottom-up approach. Rather than starting off by creating a system seeming to perfect in theory, this setup can be used for rapid testing of smaller features.

Sammendrag

Denne forskningen er en del at et studie om autonome taubåter. Studiet er et samarbeid mellom to studenter, men med adskilte oppgaver. Studiets felles mål er å lage en fungerende prototype av en sverm autonome taubåter.

I denne delen av studiet har det blitt utviklet algoritmer for a finne båtenes korteste rute fra start til mål.

Prosjektet startet med et omfattende litteratursøk. Basert på funn er det foreslått ulike algoritmer og rammeverk for ruteplanleggingssystemet. Det ble tatt et valg om å forenkle søkedomenet ved å bruke en type "road map"-graf kalt "Visibility graph". Søkealgoritmen A* er blitt brukt for å finne korteste rute innenfor denne grafen. Dersom det finnes en rute fra start til mål vil den også være den korteste ruten i det faktiske miljøet.

Noen algoritmer er funnet i eksterne bibliotek for så å ha blitt tilpasset rammeverket for dette systemet. Algoritmer har også blitt utviklet fra bunnen av. For eksempel er det utviklet en måte å finne ruter rundt et skip i sanntid, ved å kun ha kjennskap til skipets dimensjoner, posisjon og orientering. Det har også blitt utviklet en ny metode for a finne korteste vei i en type "road map"-graf. Metoden bruker mindre minne enn lignende metoder, og er svært rask. En testkjøring er gjennomført i fem ulike miljøer. Gjennomsnittlig fart ble målt til å være 12.3 ganger raskere enn et sammenlignbart søk ved bruk av A*. Forbedringen blir enda mer imponerende når kompleksiteten i miljøet øker.

Ruteplanleggingssystemet har blitt implementert i kommunikasjonsrammeverket "Robot Operating System". Systemet har blitt testet på et fysisk testoppsett med stor suksess. Testoppsettet er et startpunkt for videre forskning innenfor studiet om autonome taubåter. Ved a bruke testoppsettet kan taubåtene utvikles med en "bottom-up" tilnærming. I stedet for å lage et system som er tilsynelatende perfekt i teorien, kan testoppsettet brukes til hyppige og raske tester av mindre funksjonaliteter.

Table of Contents

List of Figures

Abbreviations

APSP	=	All-pairs shortest path
KM	=	Kuhn-Munkres
LOS	=	Line-of-sight
ODA	=	Obstacle, detection and avoidance
Pose	=	Position and orientation
ROS	=	Robot Operating System
SVG	=	Scalable Vector Graphics
Tug	=	Tugboat

Chapter 1

Introduction to Path Planning of Swarm Tugboats

Imagine a vessel aiming to dock in a crowded harbor. The ship usually operates on the open sea several months at the time, so it is not built for doing precise manoeuvres. The future goal is for the captain to call for assistance, and just relax.

A swarm of tugboats will shortly surround the ship. The tugboats are autonomous, meaning thay can sense and manoeuvre in the environment without human input. They have planned their own path to the ship.

Using only their sensors, the individual tugs can calculate how much force they should apply to keep the ship on track to the designated goal. This is done without the need of an external master system coordinating the tugs, or even tugboat-tugboat communication.

This is a typical scenario of how the tugboats this study is aiming to make, will operate.

This study of autonomous tugboats is conducted together with Sondre Midtskogen, but the projects are separate. Midtskogen is a master student, writing a master thesis in product development.

The study of autonomous tugboats was initialized by TrollLABS at the Department of Mechanical and Industrial Engineering (NTNU). It is meant to be a contribution to the study of autonomous boats in Trondheim. Midtskogen has been working on the study from its beginning, the fall of 2016.

Midtskogens goal is to create a working proof of concept on a physical test setup. He wanted to create a platform, which is possible to use for rapid testing, aiming for the ultimate goal of a autonomous tug swarm. This project will contribute to making the test setup work, and take path planning some steps further.

Summarizing the vision of the path planning system, the tugboats,

1. should minimize the communication between each other and with a master

system

2. should plan their own routes and replan them when necessary
3. should always choose the quickest path

The approaches to path planning used in this study, are chosen with the future goals in mind, as explained in each chapter. Nonetheless, the tug prototype in this study is very limited, with only one, simple load cell as sensor, and no GPS.

Communication is required to a great extent, even for making the physical test setup work. In addition, the tugs will not be able to detect collision until they have already collided. This conflicts with the first two goals described above, but the system made is a working prototype, and is only meant to lay a foundation for future work on this study.

1.1 Problem Description and Scope

The working problem descriptiong has been the following:

Develop route planning for a swarm of tugboats. This includes:

- Planning route
- Speeding up existing guidance algorithms
- Creating system interface
- Implementing algorithms on physical test setup

After a period of literature research, the scope was defined to be the following:

- Researching for and developing an algorithm to find shortest path on Euclidean distance in a closed environment.
- Planning movements of tugs, using the shortest path algorithm. X tugs will move to Y($\leqslant X$) positions in the shortest time possible, without colliding with each other.
- Speed up the algorithm to search for shortest path within a road map.
- Finding and implementing an appropriate interface between this system, the control system and a physical test setup, together with Midtskogen.
- Implementing a waypoint generator on the physical test setup.
- This project will *not* consider the problem of avoiding unknown objects, like other boats

Figure 1.1: Tug prototype

1.2 Why the Title and Problem Description Changed

The title and problem description were created before hardly any research were done, and before knowing how limited the tugboats were.

The old title was: ”Onboard Guidance of Autonomous Swarm Tugboats”. The problem description was the same as above, except for the removal of the word ”onboard”.

The ultimate goal is for the tugs to plan and execute every movement on their own, with as little explicit communication with the rest of the system as possible. The tugs are not equipped with a GPS, and their small processors are not able to run the required algorithms. In other words, they are not ready to plan for themselves, *onboard*.

1.3 Previous Work

In Midtskogens pre-master, he did a concept study of how to do automated ship docking using autonomous tugboats. Simultaneously with this path planning research, he has created physical prototypes and the control system for the tugs.

The physical prototype of the tugs can be seen in Figure 1.1. The tug has a weight cell in front, to measure pressure. It has one thruster on each side, powered with batteries.

In addition to the tugs, he has made a physical setup to test the tugs in. This is a paddling pool, with a system for wireless communication.

To compensate for the tugs' absence of GPS, their positions will be calculated using a top-view camera and computer vision.

1.4 Outline

This thesis is organized as follows: The three following chapters each represents a feature in itself. Theory, justification of chosen methods and description of new methods are all included in each chapter. Chapters 2 and 3 are about the first problem description "planning route". Chapter 2 also includes "speeding up existing guidance algorithms". Chapter 4 is about "creating system interface".

Chapter 5 describes the process of solving the last point in the problem description, "implementing algorithms on physical test setup".

In Chapter 6, the suggested solution for all four points in the problem description will be tested, evaluated and discussed. A conclusion of the system as a whole is given in Chapter 7, before suggesting how to move forward in the study of autonomous tugboats in Chapter 8.

Chapter 2

Path Planning

Path planning is the work of finding a route from a start position to an end position. The goal is to find an optimal, collision-free route of some cost function, automatically. In this chapter, the optimal route is the shortest route on Euclidean distance. Therefore, the term ”shortest path” refers to the shortest, collision-free path on Euclidean distance.

The chapter will start off by classifying path planning techniques which have been considered and evaluated, based on a literature study. At the end of each section, the reason for choosing certain techniques and simplifications are explained. Further, a brief description of the architecture of the path planning program is given.

The two following sections will elaborate further on two important features which chosen as a result of the literature study.

From section 2.4 and onwards, new algorithms and methods are presented. A speed-up for the use of A* search is presented. Further, a tool for visualizing calculated paths on a map is shown. The following section presents a way of avoiding collisions with a ship which changes position during run time. The final section will present a graphical user interface tool for creating environments.

2.1 Classification of Path Planning Techniques

Path planning techniques can in general be distinguished between global and local path planning. A combination of the two is often called hybrid path planning. It is assumed that the goal's location is known and stationary.

2.1.1 Global Path Planning

Global path planning utilize domain-specific knowledge, such as maps, and plans a route from start to a predefined goal based on this information. This should be

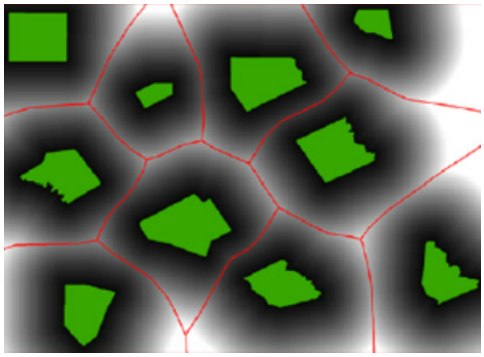


Figure 2.1: Voroni diagram (Dong (2008))

done in an optimal or close to optimal way, on one or more parameters, such as time. The global plan will avoid static objects.

From literature, efficient global path planning techniques can be divided into optimization methods and heuristic search methods (Liu et al. (2015)).

Optimization Methods can optimize paths on advanced characteristics, such as fuel saving, weather conditions and danger level (Niu et al. (2016)). These algorithms are typically computational expensive, and therefore not suited for real time applications.

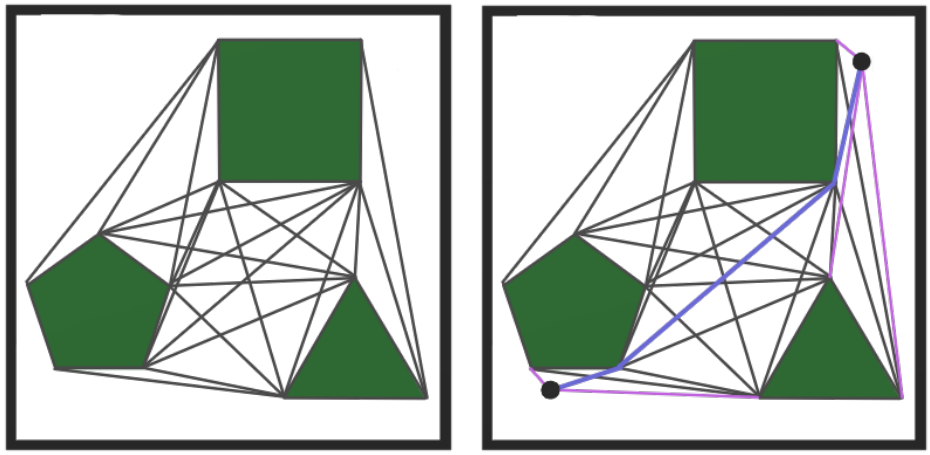
Heuristic search methods are characterized by the use of a heuristic search function within a discrete grid (Campbell et al. (2012)). Heuristic search strategies are often called *informed search* strategies, as they use some information beyond the what is defined in the actual problem definition (Russell and Norvig (2014)). By using knowledge about the domain, the search can be guided in a more efficient way than blind search. A good choice of heuristics is vital for the search method to be robust and efficient (Signifredi et al. (2015)).

The boats can travel anywhere on the surface of the water giving a two-dimensional search space for finding routes. To search though every possible route in this space will be very time consuming. Most of the possible routes will never even be an optimal path, and can safely be removed from the search space for speed purposes.

Tugboats belong to one harbor, and they will only travel within a fixed area. Within this area, there are known static objects. This implies that path planning calculations avoiding static objects can be done in advance.

To reduce the search space to one dimension, a road map approach (Niu et al. (2016)) can be used. This type of approach will find a set of traversable roads, which can be searched.

Two road map approaches were considered. The first one is called Voroni diagram (Wu et al. (2013)) and an illustration of the method is shown in Figure 2.1. The method creates a road map by maximizing the distance from every obstacle. In the figure, the green areas are obstacles, the red lines makes up a road map, and the shaded areas from black to white illustrates how far from an obstacles the area



(a) Visibility graph **(b)** Blue: Shortest collision-free path

is.

The other considered approach is visibility graphs. A visibility graph is a graph with nodes placed on every corner of all obstacles and edges connecting the nodes, which can see each other (i.e. a straight line can be drawn between them) (Lozano-Pérez and Wesley (1979)). Figure 2.2a shows an example.

To find a collision-free path between any two points in the environment, the visibility graph is extended to include the start and end points as new nodes, as illustrated in Figure 2.2b. The purple lines are extensions to the original visibility graph, and the blue line is the shortest path.

It is possible to use Visibility graphs to find optimal paths on Euclidean distance. This is a great foundation for finding the shortest path on time. Dubins (1957) stated that "The shortest path (minimum time) between two configurations (x,y, ψ) of a craft moving at constant speed U is a path formed by straight lines and circular arc segments." The straight lines can be the ones found in a visibility graph.

A Voroni diagram can be calculated in $O(n \log n)$ time, which is better than visibility graphs time of $O(n^2)$ (Niu et al. (2016)). As the environemnt is known, this calculation is only done once, and can even be done offline, where calculation speed is no issue. The Voroni diagram is not optimal, like the visibility graph. Because of its optimality and the fact that speed is unimportant, the choice was made to use visibility graphs in this study. The visibility graph library used is made by Obermeyer and Contributors (2008).

2.1.2 Local Path Planning

Local path planning is path planning where the boat (or any robot) moves in an unknown or dynamic environment, and the algorithm reacts to obstacles or changes in the environment (Buniyamin et al. (2011)). Changes can be noticed

by for instance sensors or cameras on the boat. By using local path planning, a new route to avoid the obstacles is created. Local path planning is also known as Obstacle Detection and Avoidance (ODA). (Zhuang et al. (2011)).

2.1.3 Hybrid Path Planning

An autonomous system with only local or only global planning is infeasible in the real world. Usually, an combined approach, often called *hybrid planning*, is used. Global and local planning are often structured in two different layers, when using a hybrid architecture.

It is chosen to implement a layered structure in this study, as this is done successfully in many papers (Larson et al. (2006), Casalino et al. (2009), Svec and Gupta (2012)). The global layer will provide an optimal path given the static objects. The local layer, will provide routes to avoid colliding in dynamic obstacles which positions are known.

A layered structure makes it possible to implement extra layers at a later stage, and still make use the planning in other layers.

2.2 Searching for Shortest Path: A* algorithm

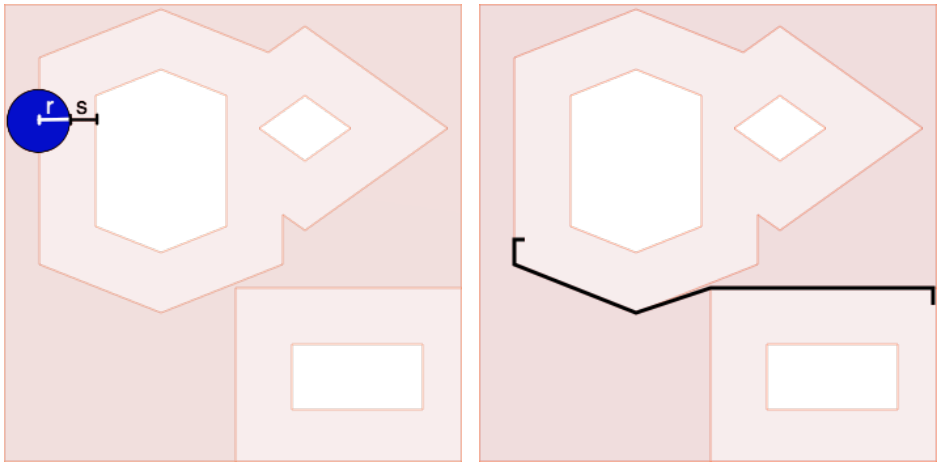
Shortest path from start to end points is found by searching through vertices in the systems visibility graph using the best-first search algorithm, A* (Russell and Norvig (2014)), with the Euclidean distances from point to point as costs.

A* is a very popular search algorithm in the robotics community (Blaich et al. (2012), Jouandeau and Yan (2012), Casalino et al. (2009)), as it is very efficient. In each node, A* adds the accumulated cost from the start node to an estimated cost of reaching the goal node. By doing this, A* tends to focus its search to the most relevant nodes in the search graph (Maren et al. (2001)). The algorithm will create an optimal shortest path on distance if the heuristics are admissible (Niu et al. (2016)).

An admissible heuristic, is an heuristic that never overestimates the cost to reach the goal (Russell and Norvig (2014)). Euclidean distance is a heuristic with this property. The algorithm is also complete, meaning, it will always find a solution if one exist (Russell and Norvig (2014)).

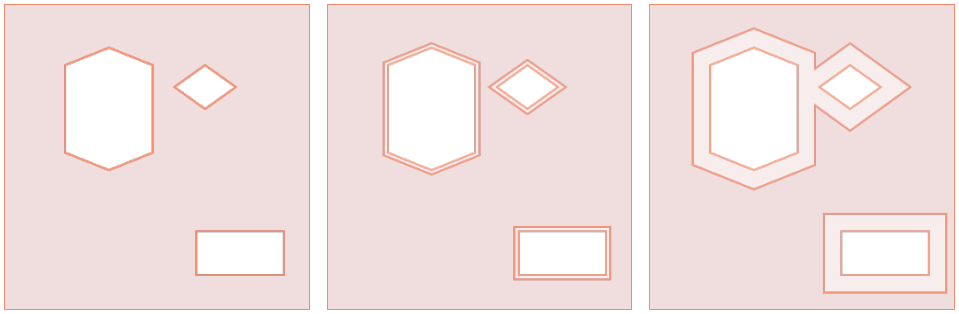
2.3 Safety Margin

The path planning algorithm that will be used, will plan a route from point to point, neglecting the fact that boats have a certain dimension. A widely used technique to add area to the robots planned for, is to rather enlarge the obstacles (Erdmann and Lozano-Perez (1986)). The area around the obstacles must be a boat radius to compensate. In addition to the compensation, extra safety margin around obstacles should in general be added. In Figure 2.3a, the boats radius is r and the extra safety margin is s. The observant reader will notice that the safety



(a) Safety margin = r + s **(b)** Tug starts inside a safety area

Figure 2.3: Working with safety areas



(a) No safety margin **(b)** Small safety margin **(c)** Large safety margin

Figure 2.4: A simple environment with different safety margins

margin should be rounded on obstacle corners, if it was true that it was a constant distance from the obstacle. Nevertheless, they are made as corners on purpose, as the A* algorithm searches through vertices. A rounded corner would have to be simplified to one or more vertices at a later stage, anyway.

Adding safety margin (and boat compensation) can be done in the path planning program. A function in the environment class use "The Clipper library" (Johnson (2014)) to expand all obstacles with a constant value. See Figure 2.4. Notice how the polygons merge if the safety margins overlap (Figure 2.4c).

Tugs may start or end within safety margins. For instance, they have to be close to the dock while docking a ship. To make the distance travelled inside a safety zone minimized, the shortest path algorithm adds the closest point outside the safety zone to the shortest path, as seen in Figure 2.3b.

2.4 Reduce Online Calculations: All-pairs Shortest Path

One way to reduce online computation time, is to calculate the shortest path between all vertices in the environment offline. In Figure 2.2b, they are the grey paths. The shortest paths between all obstacle vertices are precalculated by searching though the gray lines and the solution is saved in a look-up table. The look-up table method is inspired by the conference paper, Guzman et al. (2005).

2.4.1 "Go-via" matrix

Take a look at Figure 2.5. Figure 2.5b is the all-pairs shortest path matrix (or look-up table) of the environment in Figure 2.5a. A matrix entry, $m(row, col)$, represents which vertex to visit next, when making a path from row to col. That is why it is chosen to call the matrix the "go-via" matrix. An example of use of the "go-via" matrix follows.

The goal is to find the shortest path between vertex b and vertex l, and the procedure goes like this:

1. Look up $m(b, l)$, to find c
2. Look up $m(c, l)$, to find k
3. Look up $m(k, l)$, to find l
4. Look up $m(l, l)$, to find l
5. Join the path: b $\rightarrow$ c $\rightarrow$ k $\rightarrow$ l

In adition the the APSP matrix, another matrix with the shortest path lengths between all vertices in the environment is made. Looking up distances in such a matrix makes the algorithm avoid repeatedly calculations of the same distances.

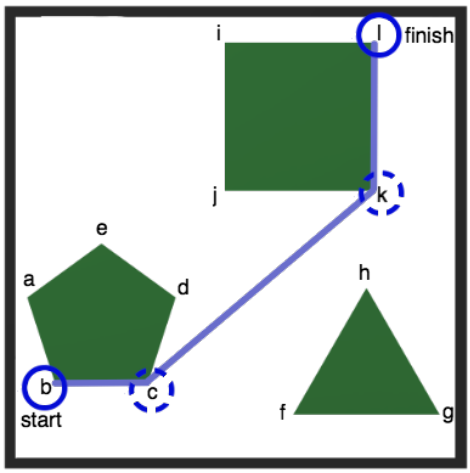
2.4.2 Algorithm Extension

Start and finish points are rarely on the vertices. The shortest path between any two points is found by a few extra calculations.

For the start and end points, all distances between the points and the vertices within their respectively visibility polygon must be found. In Figure 2.6, these vertices are the orange points within the yellow area, representing the visibility polygon of the black point.

The distance calculations within the visibility polygon have to be done regardless of search method used within a road map approach.

Finding the shortest path between any two points, is done by following the pseudocode in Algorithm 1. In the algorithm, apsp is a function which use the "go-via" matrix to find the shortest path between the two vertices in the argument.



(a) Environment with labels on vertices

From / To

To \ From	a	b	c	d	e	f	g	h	i	j	k	l
a	a	b	b	e	e	b	b	e	i	e	e	i
b	a	b	c	c	a	f	f	c	a	c	c	c
c	b	b	c	d	d	f	f	h	d	j	k	k
d	e	c	c	d	e	f	f	h	i	j	k	k
e	a	a	d	d	e	d	h	h	i	j	k	i
f	b	b	c	d	d	f	g	h	i	j	k	k
g	f	f	f	f	h	f	g	h	h	h	k	l
h	e	c	c	d	e	f	g	h	j	j	k	k
i	a	d	d	d	e	j	l	j	i	j	j	l
j	e	c	c	d	e	f	h	h	i	j	k	i
k	e	c	c	d	e	f	g	j	j	j	k	l
l	i	k	k	k	i	k	g	k	i	i	k	l

(b) "Go-via"-matrix

Figure 2.5: All-pairs shortest path.

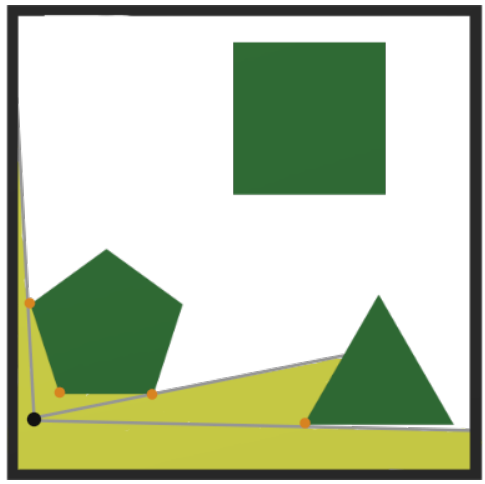


Figure 2.6: The visibility polygon of a point is the area the point can see in the environment. The visibility polygon of the black point is the yellow area

Algorithm 1: Finding shortest path using APSP

```
Data: apsp, apsp-cost s, f, visGraphS, visGraphF
Result: Shortest path from s to f
for each v in visGraphS do
    for each w in visGraphF do
        cost = dist(s,v) + apsp-cost(v,w) + dist(w,f)
        if cost < currentLowestCost then
            currentLowestCost = cost
            currentBestVertices = (v,w)
return (s - apsp(currentBestVertices) - f)
```

2.4.3 Advantages of Suggested APSP

There are several advantages of the suggested APSP method. First of all, after it has been calculated for the first time, it is very efficient to find shortest path in a known environment. It is also efficient to replan a route in real time, to avoid dynamic obstacles, like other boats.

The novice way to store the shortest path from all to all points, is keep the all the points of every possible path in memory. In comparison to this method, the suggested method is very memory saving, and therefore highly scalable. This is because the suggested method only stores the next point in the shortest path to any point.

2.5 Visualizing Calculated Routes

The easiest way to verify if the algorithms return the expected paths is by visualizing them.

A modified version of Clipperlibs SVG builder was made for rapid visualization. The original SVG builder made an SVG (scalable vector graphics) image a set of polygons. The modified version adds an optional number of polylines, which is a set of waypoints, making up a piecewise linear curve. An example is shown in Figure 2.3b.

2.6 Local Replanning Around Ship

The ship is the only dynamic obstacle in this simplified tug world. Tugs must be able to plan routes around it.

A scenario is to move one tug or several tugs from start to goal or formation around a ship. By assuming the ship stays still in a known position for one scenario, it could be added to the global planner as a static object. By doing so, the visibility graph would have to be calculated for each scenario. Calculating the visibility graph repeatedly is inefficient, and by doing so, the fact that the environment is known

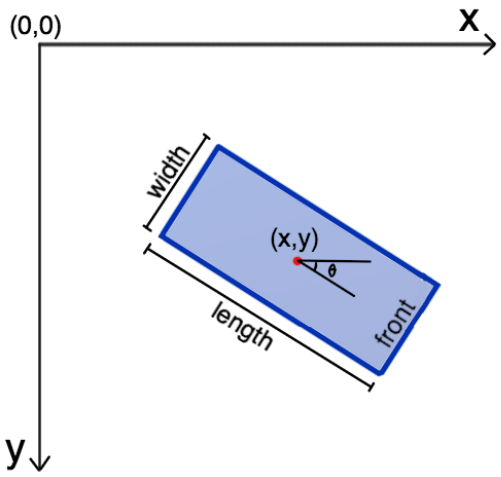


Figure 2.7: Ships position (x,y), orientation θ, length and width

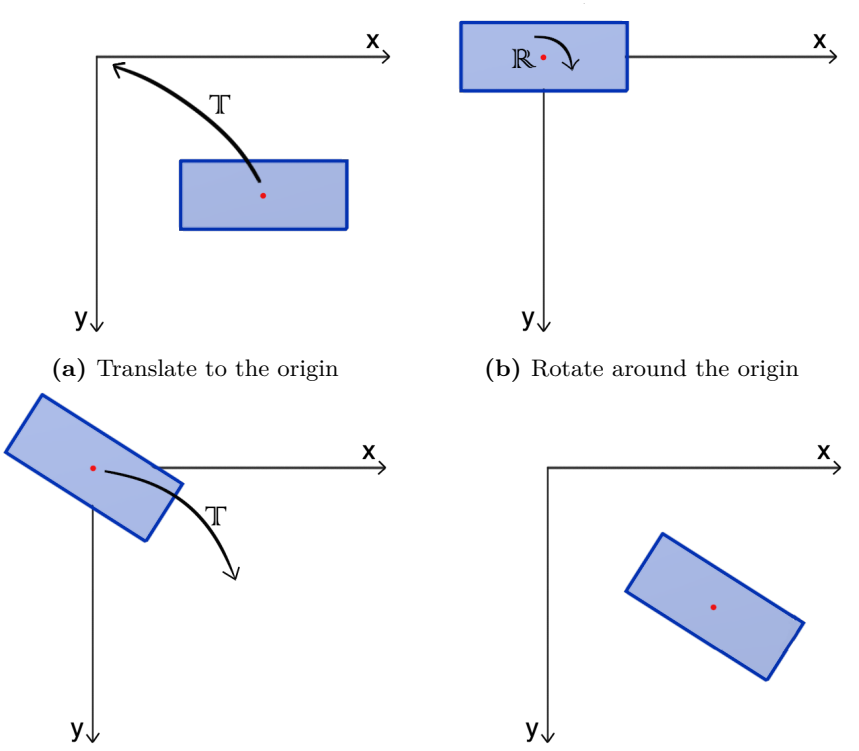
is unutilized. The system would not even tolerate that the ship had moved during the scenario.

In this system, a ships position and orientation can be found. Its dimensions, width and length, is known. This information is used to do local planning around the ship. A ships position is given as its geometric midpoint in x,y-coordinates in the world coordinate system with (0,0) in the top left corner. Its orientation is an angle between $-\pi$ and π, where its nose is pointing to the right at an angle of zero (Figure 2.7).

The ship is simplified to be a rectangle in the x,y-plane. This simplification is sufficient for many ship types. Given its orientation, position, width and length, it is possible to pinpoint the space occupied by the ship in the water, by following Algorithm 2.

The last line in the algorithm may need some extra explanation. $\mathbb{T}(x, y)$ is a matrix which translates a point by x and y, by doing matrix multiplication on the column vector of the point. $\mathbb{R}(\theta)$ is another matrix, which rotates the a point by θ degrees in a clockwise direction (counter clockwise in an "L"-shaped coordinate system) around the origin, (0, 0). To rotate a point around any other point but the origin, a translation to the origin and an inverse translation must be done respectively before and after the rotation is done. This is exactly what is done in Algorithm 2. Merging all four column vectors to a single matrix, will yield the same result in matrix multiplication as multiplying column by column. The area of the ship is defined by its four corners, so all internal points will follow the transformation. See Figure 2.8.

When a tug is heading towards its second to last waypoint, it will calculate



(a) Translate to the origin

(b) Rotate around the origin

(c) Translate back to original position

(d) Ship rotated around its own center

Figure 2.8: How to rotate a ship around its own mid point

Algorithm 2: Pinpointing ship in world coordinates

Data: orientation (θ), position (x,y), length (l), width (w), ship (empty 2x4 matrix)

Result: The four corners of the ship in world coordinates

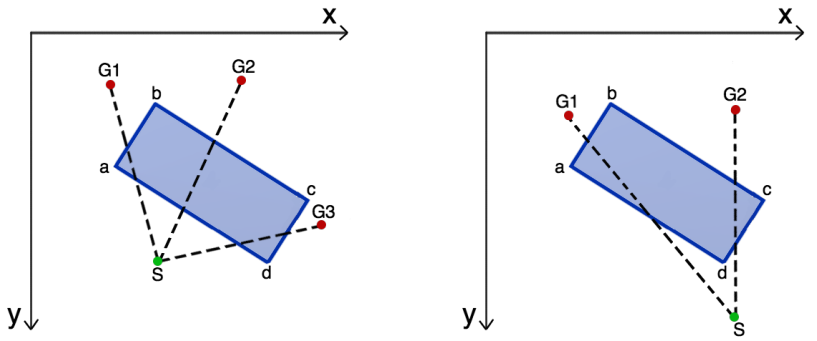
ship(0) = (x - l/2, y - w/2)

ship(1) = (x - l/2, y + w/2)

ship(2) = (x + l/2, y + w/2)

ship(3) = (x + l/2, y - w/2)

ship := $\mathbb{T}(x, y)$ * $\mathbb{R}(\theta)$ * $\mathbb{T}(-x, -y)$ * ship



(a) **(b)**

Figure 2.9: How a line between two points can intersect the ship

which new waypoints to add to its path in order to avoid colliding with the ship. Following Algorithm 3 will add the waypoints, giving the shortest path around the ship possible, at the time of calculation.

Figure 2.9 shows how lines can intersect the ship. Lines can also intersect the ship equivalent from other directions. Using Algorithm 3, the following result would be evident for Figure 2.9:

Fig.	Start	Goal	Vertex added	Comments
(a)	S	G1	a	Common vertex of lines $\overline{da}$ & $\overline{ab}$
(a)	S	G2	a&b	Calc. length of path S-a-b-G2 & S-d-c-G2
(a)	S	G3	d	Common vertex of lines $\overline{ad}$ & $\overline{dc}$
(b)	S	G1	a	Common vertex of lines $\overline{da}$ & $\overline{ab}$
(b)	S	G2	c	Common vertex of lines $\overline{dc}$ & $\overline{cb}$

Algorithm 3: Planning shortest path around a ship

check intersections between line from start-goal (SG-line) and all four edges of ship.
if *SG-line intersects two adjacent edges* **then**
  add common point of the two edges to path
else if *SG-line intersects two opposite edges* **then**
  calculate and choose shortest path around ship
else
  do not replan

The suggested replanning algorithm has an constant asymptotic upper bound ($O(1)$), much more efficient than recalculating visibility graph ($O(n^2)$).

The algorithm for finding the shortest path around a ship does not take the environment into account. It assumes that a boat is free to move on either side of

the ship at any time. This is typically true for a ship outide of a harbor, but of course, not the case if the ship is too close to an obstacle or environment borders.

2.7 Creating the Environment

The proposed algorithms of this study requires environments defined by points marking vertices of obstacles. The program will read text files in the following format.

The first lines contain the four x,y-coordinates representing the outer boundary, listed counter clockwise in an "L"-shaped coordinate system. The following lines contain coordinates which represents obstacles, split by empty lines shifts. They are listed clockwise. The following list would create a 150x100px environment containing one rectangular shaped obstacle.

```
0,0
150,0
150,100
0,100

40,40
40,60
70,60
70,40
```

Simple environments like this can be made manually, but making larger environments is quite tricky to make. This is why a graphical program, hereby called Environment Creator is made.

Environment Creator loads a chosen image of a map and displays it on screen. The user mark a rectangular outer boundary (marking where the tugs are allowed to move), by using the mouse to click on the bottom left and top right of the allowed area, followed by the enter button. A red rectangle marking the outer boundary will appear. The program can easily be extended to allow other shapes than a rectangle as outer boundary, but it is not necessary, as will soon become obvious.

Further, the user can mark obstacles by clicking around them in clockwise or counter clockwise direction. Red circles will appear at each point marked, as well as lines marking the polygon after the user has pressed enter. An example of the the harbor in Trondheim is shown in Figure 2.10.

A harbor usually has docks and other large obstacles reaching beyond the allowed area. These can be marked in the same way as other obstacles, but with edge points outside the allowed area. This is why only a rectangular outer boundary is needed, as one can mark shapes on the edges simulating restricted areas.

The environment coordinates are saved to a text file, where the coordinates are pixels of the image. The number of pixels in the image is known, and so will the size of the area the map image covers be. From this, a scale between pixels and meters can easily be calculated.

Figure 2.10: Trondheim harbor (Norge i bilder (2016))

In the test setup, a birds-eye photo image of the environment was taken. This is image is equivalent to a satellite photo or a map in a real harbor.

Chapter 3

Planning Paths and Motion in a Swarm Context

The previous chapter presented a method for planning optimal paths on Euclidean distance for *one* tug. The optimal routes of single robots are not necessarily the best combined plan, nor even always feasible. This chapter will elucidate the main decisions made in moving from one to many tugs. Since the tugs are uniform and work together, they are referred to as a swarm.

A single robot is free to move as long as it avoids static objects, invariant of time. More robots in the system requires planning in time, as two robots cannot be at the same place at the same time. Path planning as a function of time is often called motion planning. Section 3.1 will briefly describe the main way of distinguishing between different motion planning approaches.

Path and motion planning algorithms are often explored in a context where start and target positions are given. In this study, the tugs are not aware of where to go next until a master system will notify them. How to choose which tugs to assign to which target is described in section 3.2.

It has been discovered that the tugs will never cross on the edges of a road map, given some conditions. This will be derived in the last section, before describing how this fact has been taken into use in motion planning.

3.1 Motion Planning Approaches

When not only the path, but the trajectory of robots are planned, it is referred to as motion planning. The field of motion planning is often categorized as *centralized planning* and *decoupled planning*, denoting the amount of autonomy in planning. It is worth noting that there are various approaches along the spectrum of the two extremes (van den Berg and Overmars (2005)).

3.1.1 Centralized Planning

Centralized systems treats all robots as one composite robot, and typically planes within the combined search space of all robots. One robot, or a master system is responsible for the coordination of all robots. In theory, centralized systems *will* find a solution, if one exists.

3.1.2 Decoupled Planning

In decoupled planning, each robot is responsible for planning its own routes, ignoring the existence of other robots. This can be done using ordinary path planning techniques presented in chapter 2, followed by a process for resolving conflicts (Bennewitz et al. (2002)).

Resolving can be done in numerous ways, including "prioritized planning" (van den Berg and Overmars (2005)). In this approach, the robots are given different priorities. The robots are planned for in descending prioritization, avoiding static objects and treating the already planned for robots as dynamic objects.

If there are n tugs, there are n factorial ways to make prioritizing schemes. It is too costly to test all schemes (Azarm and Schmidt (1996)), so searching algorithms is commonly used to find possible solutions (Maren et al. (2001)).

Planning trajectories before they are executed, often require the duration of subsections of the paths to be known in advance. A subsection can be an edge of a road map, or a constant distance in other methods. This duration may be hard to estimate.

3.1.3 Choosing Approach

Centralized planners are complete, but complexity grows exponentially to the number of robots. Decoupled approaches are not complete, but highly scalable. The choice between centralized and decentralized approaches is usually determined by the tradeoff between computation time complexity and the amount of completeness lost (LaValle and Hutchinson (1998)). In this study, there are yet two arguments to decide on method.

The future goal of this study is to make intelligent tugs, not an intelligent system to control tugs. Another goal is to bring the necessary communication to a minimum. For those reasons a decoupled approach should be chosen in the future.

Because of the limited amount of sensors on the prototype tugs, the tugs will use the communication media to a large degree. The processors on the tugs are also very limited, and would not be able to run path plannings algorithms. Therefore a centralized system was chosen.

3.2 Preliminary Decision Making

The goal is to plan the motion of several identical robots, given a set of goal locations. The goals are not preassigned to robots. The only requirement to fill

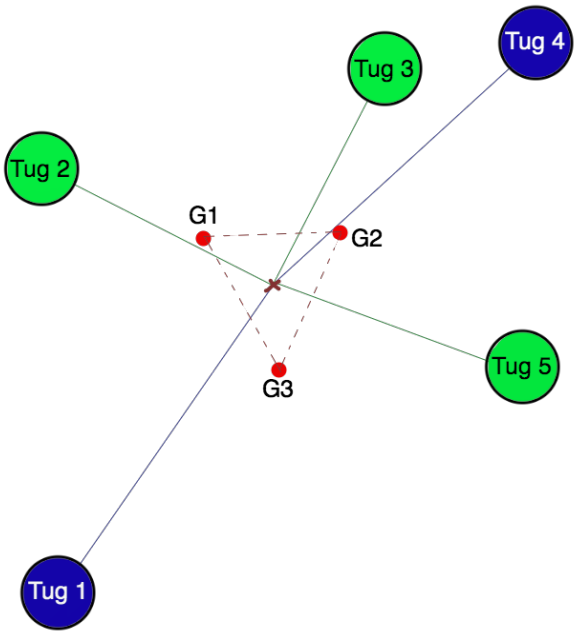


Figure 3.1: The green tugs are the three closest to the red points mid point. They are chosen as the best set of tugs, and sent to the next algorithm (target assignment)

that each goal with an unique robot. This is called a permutation invariant multi-agent motion planning problem Yu and Lavalle (2013).

There may be more robots than goals, so the algorithm will choose which robots to use. There can not be more goals than robots.

3.2.1 Selecting Closest Tugs

It is assumed that there will always be an equal amount or more available tugs than requested positions to be filled (goals). The only way to find the perfect set of tugs, is to calculate the shortest paths for all available tugs to all goals, which can be quite time consuming.

In a harbor scenario, tugs will in general be requested to surround a ship. An approximation of the best set of tugs to assign is given as the set of tugs with the shortest Euclidean distance to the mid point of the requested goals. An example can be seen in Figure 3.1. The mid point of N goals is calculated as the centre of mass:

$$(x_{mid}, y_{mid}) = \frac{1}{N} \sum_{i=1}^{N} (x_i, y_i) \tag{3.1}$$

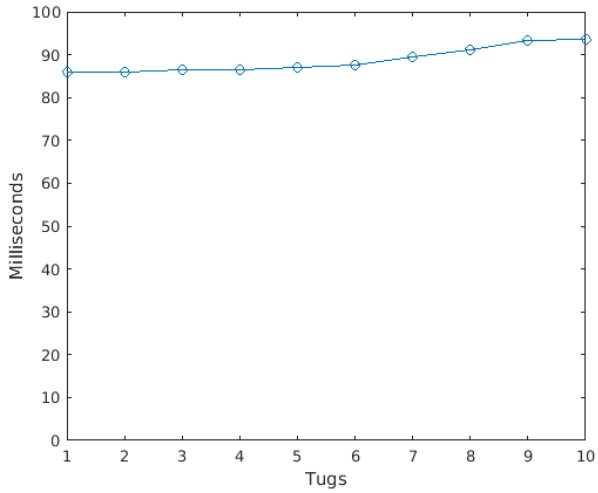


Figure 3.2: MUNKRES speed

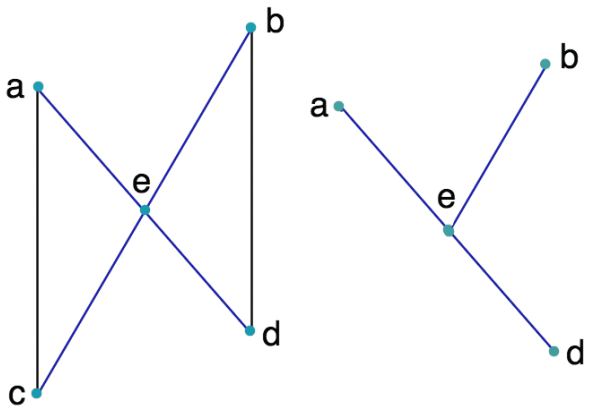
3.2.2 Target Assignment

The ”assignment problem” can be used to minimize the sum of path lengths for a set of permutation invariant tugs. Given n workers, n tasks, and the costs of each worker doing each task, the assignment problem is to assign each task to a worker, in the combined cheapest way possible. One way to solve this problem is by using the Kuhn-Munkres (KM) algorithm (Munkres (1957)). A implementation of the KM algorithm written by John Weaver (www.saebyn.info) is used in this study.

When the number of tugs and goals are the same, the KM algorithm is used to assign goal locations to tugs. The tugs are the workers, and the length of their shortest path to all finish points are the costs.

the KM algorithm has $O(n^3)$ efficiency, which is large, and should in general be avoided. However, n, being the number of tugs going to a set of end locations, is usually small, not ever bigger than six in the physical prototype. Figure 3.2 shows the average time of five runs of the KM algorithm with an increasing number of tugs, from 1 to 10. Note that the time varies from 85.8 to 93.6 milliseconds. This is only a 9.1% increase in run time, on 10 times as many tugs. By this, it is concluded that the part of KM giving the asymptotic speed of $O(n^3)$ is not expensive for small n, it is rather the overhead cost.

The suggested algorithms for can be used in the future, even though communication should be held to a minimum. There will always be need for communication to inform when a ship requires assistance. The master system could use KM before to assign goal points to the tugs. The tugs would search for shortest paths in identical, internal road maps, and would therefore still minimize the total path distance.



(a) Paths (in blue) will never cross like this

(b) A path will never have its goal on another path

3.3 Motion Planning Method

A method for planning the motion of a group of robots has been developed. Given that they plan routes within the same road map, it is possible to avoid collision without robot-robot communication. The first section will prove that paths of robots would never cross, if they had no area. The next section will explain how to use the proof can be used, even though robots have a certain size. The last section will explain an efficient way to plan the motion of several tugs, using the proof.

3.3.1 Paths Will Never Cross

This study use a road map based approach with the line-of-sight (LOS) method. This means that possible paths will always consist of straight lines from node to node, defined on a road map.

This section will prove why the tugs only need to be coordinated on the nodes of the road map graph, and not on the edges. This requires paths to be are combined so that the total distance travelled by the tugs are minimized, and that a tug can be planned for as a point (no area). A solution to cope with the last requirement will be discussed later.

Figure 3.3a will be used as illustration, but the proof is valid for any crossing, straight lines of a path. By the triangle inequality (Khamsi and Kirk (2011)),

$$\|ac\| \leq \|ae\| + \|ec\| \tag{3.2}$$

and

$$\|bd\| \leq \|be\| + \|ed\| \tag{3.3}$$

If the routes cross, the total sum of the paths lengths is

$$L = (\|ae\| + \|ed\|) + (\|be\| + \|ec\|) \tag{3.4}$$

Which is the same length as if they switch end point when crossing at e,

$$L = (\|ae\| + \|ec\|) + (\|be\| + \|ed\|) \tag{3.5}$$

By adding both sides equation 3.2 and 3.3, and acknowledge that the right side is equal to the right side of equation 3.5,

$$L \geq \|ac\| + \|bd\| \tag{3.6}$$

which implies that crossing never gives the shortest path combined.

By the triangle equality, it can also be shown that one path's goal point will never be on another path. This is a special case of the previous evidence. This will be shown by using Figure 3.3b, and showing that paths bg_1 and ag_2 combined always are shorter than or equal to the length of bd and ae combined.

$$\|ae\| \leq \|ag_2\| \tag{3.7}$$

$$\|bd\| \leq \|be\| + \|ed\| \Rightarrow \|be\| \geq \|bd\| - \|ed\| \tag{3.8}$$

$$(\|ae\| + \|ed\|) + \|be\| \geq (\|ae\| + \|ed\|) + \|bd\| - \|ed\| \tag{3.9}$$

which simplifies to

$$(\|ae\| + \|ed\|) + \|be\| \geq \|ae\| + \|bd\| \tag{3.10}$$

The two sides of Equation 3.10 are only equal if $e = d$. Two tugs having the same goal is never the case, and it is therefore correct to say one paths goal point will never be on another path.

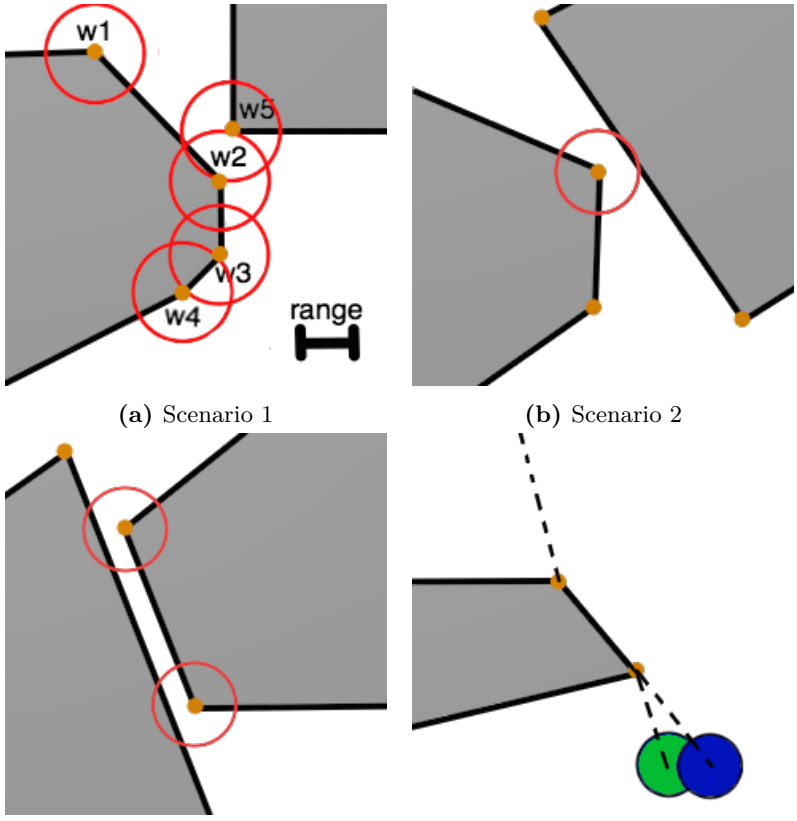
The inequalities derived in 3.6 and 3.10 are very important in the algorithm implemented. They imply that if an optimal resource assigner, like the Kuhn-Munkres algorithm, is used, the paths will never cross. However, they might, and will often meet in the same waypoint. Therefore, when planning the tugs motion in time, it is only checked for collision in waypoints.

Another interesting fact is that two tugs will never meet in two different points, unless the two points are connected with the shortest path of both tugs. This is the case since the tugs will always choose the same path between two points, as they use the same shortest path algorithm. This fact is not used in the current planning algorithm, but could be used to reduce the number of waypoints to check for collision.

3.3.2 From Point to Area

Tugs do, of course, have an area, so some adjustments have to be done in order to use the planning method suggested above. The adjustments are easiest explained by explaining Figure 3.4.

Figure 3.4a shows a close-up of an environment. The gray areas are obstacles with unspecified safety zones. A predefined range around each vertex is marked in



(a) Scenario 1 **(b)** Scenario 2

(c) Scenario 3 **(d)** Scenario 4

Figure 3.4: Scenarios where the tugs area will matter

red. If a point is within another points range, tugs cannot occupy them both at once. The points cannot be planned for at the same time, avoiding collision in that area. In the figure, w2 and w5 are within each others range, and so are w3 and w4.

In the two next scenarios, an obstacles point (Figure 3.4d) or edge (Figure 3.4c) is close to another objects edge. The solution to this problem could be to add an extra dummy point on the other objects edge, and plan for this point as if it was an ordinary vertex. The two scenarios are not taken care of in the implemented algorithm. In the harbor scenario in the physical prototype test, scenario 1 and 2 will never occur.

The final scenario (Figure 3.4b), illustrates two tugs on two close points on either a start or goal location. The scenario is not realistic in a start scenario, as the start points are always the position the tugs are already in, and they cannot start on top of each other. In theory, the goal positions could be like this, but it is assumed that the control algorithm never asks for this.

3.3.3 Planning Movements

The novice way to plan for several robots is to send one robot at the time and wait for it to finish. The tugs would never collide, and it would fulfill the objective of the shortest path algorithms, which has been to minimize the path length. However, this objective was only chosen to be a good starting point for the real objective; minimizing total completion time.

It was proved in section 3.3.1 that tugs' path never will cross, but they might merge into the same path for many waypoints. This is often the case when the tugs are located far away, but at the same side of the ship, and are requested to surround it.

In order to avoid collision in or close to the waypoints (as if the green and blue tug in Figure 3.4b were heading for the same waypoint before colliding), only one tug can head towards a waypoint at the time. The other tug(s) must wait on its previous waypoint(s) until the occupying tug has moved on.

This method only checks for collisions with other tugs once per waypoint, rather than for every movement of all tugs. This saves a lot of computation time. It is also avoids tug-tug explicit communication, which is an objective for the future.

Chapter 4

Tug Communication

With the vision of future autonomous tugs in mind, this chapter will discuss what type of communication should be used.

Coordinating the robots require some kind of communication, either implicit or explicit. Section 4.1 will explain the difference between the two communication methods, discuss how the system communicates today, and how they should communicate when the tugs become more independent. By taking the limitations of the current physical prototype into account, a type of communication is suggested.

Further, core features and rationale for choosing the communication framework, The Robot Operating System, is given.

Finally, a framework for testing the communication interface is presented. It is a simulation, which is also used to visualize how tugs interact.

4.1 Communication: Explicit or Implicit

Some sort of communication is crucial for any system consisting of more than one robot to work. Communication may be divided into explicit and implicit communication (Doriya et al. (2015)). Communication is explicit if a communication media is used to broadcast or unicast messages (Yan et al. (2013)). Messages can, for instance, be position and orientation information. This is the most researched type of communication.

The other type of communication is implicit. If robots use implicit communication, they find information from the other robots in the system via the environment or sensors (Yan et al. (2013)).

Communication via the environment is generally inspired by nature. For example, ants in a colony looking for food leaves odorous chemicals in trails leading to good food sources. Other ants will then know where to go by looking for chemicals in the environment (Dorigo et al. (2008)).

Sensor communication is to use sensors to perceive change in the environment to, for example, calculate the position of other robots (Yan et al. (2013)).

Explicit communication is accurate, but it often requires a reliable communication system to work. An error in communication media may in extreme cases lead to system failure and collisions. In addition, the communication load on the system will increase as the number of robots increases (Yan et al. (2013)).

Using implicit communication, on the other hand, the environment information will not be as accurate as when using an explicit pattern. However, the stability, reliability and fault tolerance are better (Yan et al. (2013)).

Applying a combination of the two communication methods is often a good choice. In the future, explicit communication should be used only in the very beginning of a mission. The tugs must know where to position themselves around a ship, and where to push the ship to. When those two points are known, only implicit communication should be used. The tugs will know how to position themselves by sensing necessary information from the environment, and will work as a swarm towards a common goal.

For the prototype in this study, only explicit communication is used. Again, this is because the tugs do not have sensors to obtain sufficient information about the environment to avoid collisions.

4.2 The Robot Operating System

As stated earlier, this study of autonomous tugboats is a cooperation between two students, but the two projects are quite independent. In this project, paths are created and waypoints sent. One of the goals for the other project is to control the tugs to these waypoints. In order to keep the projects independent, some interfaces had to be made before starting. It was chosen to use The Robot Operating System framework (ROS) for this purpose. First, a short introduction to some important ROS components will be given, followed by some evaluation of why ROS is chosen.

4.2.1 ROS Communication System

The Robot Operating System (ROS) is an open-source framework for developing robot software. One of ROS' core components is its communication model. The main concepts will be described.

A ROS node is an executable file within a ROS package, which performs computation, and communicates with other nodes using a ROS client library (Understanding ROS Nodes (2016) Nodes (2012)).

A node can *advertise* that it will *publish* messages to a certain *topic*. These published messages will be *broadcasted*, and available for all other nodes in the system. The nodes which wants to "read" these messages, *subscribes* to the topic.

A node can also advertise a *service*. A service node receives pair of messages, one for the request and one for the reply. It executes its service and responds in the reply message, which is sent back to the requesting node (Services (2012)). This is similar to normal function calls across nodes.

4.2.2 Evaluating ROS

A resource conflict may occur if several robots try to use the same communication media (Yan et al. (2013)). This is not a problem with ROS. ROS can also handle some communication errors. For instance, ROS is used to publish the position of the tugs 24 times per seconds. If a tug misses one of these messages, it will not break down, because it assumes the last published message is still valid. Gerkey and Matarić (2002) summarizes why a publish/subscribe paradigm is good for robot systems:

"Robots can move in and out of communication range, the communication system itself can drop messages, and the robots can experience many different failures. In order to tolerate these dynamics, the communication layer should never address robots by name. Instead, robots should communicate anonymously through broadcast means."

The tools and libraries of ROS make communication efficient and easy, and it can run across multiple computers. It is widely used in the robot community, and is well documented online.

ROS is a type of explicit communication, as discussed in section 4.1. Unlike many other explicit communication systems, ROS can handle less reliable communication.

Interfacing in the ROS framework is straightforward. It does not matter what a ROS node looks like or what it does, as long as it advertises topics of the correct predefined messages. This is perfect for this study, as the control system is totally separated from the path planning system for most of the study.

Another great feature with this type of interfacing, is its' flexibility for changes. For instance, the positions of the tugs in the physical prototype is given by a computer vision system. In a real harbor, position would probably be given by a GPS. As long as the GPS coordinates are translated to the predefined ROS messages, the path planning system wouldn't have to change anything.

With a system of stand alone components it is easy to add new functionality without editing old components. This may be important if the study of autonomous tugboats is continued by other students in the future.

4.3 Simulation

In order to know how motion planning and communication will work, without testing it in the real world, a simulation tool was made. The simulation tool must imitate the real world, including a time dimension.

4.3.1 Simulation Tool: Gazebo

Gazebo is a popular program used for simulation and visualization of robots using ROS (Koenig and Howard (2004)). ROS can be integrated with Gazebo through a set of interfaces, meaning a robot can be simulated using only ROS messages.

Gazebo has a service to provide position and orientation (pose) for robots, which is needed for the motion planner to work. A node for translating the messages from gazebo format to the format used in the control system was made.

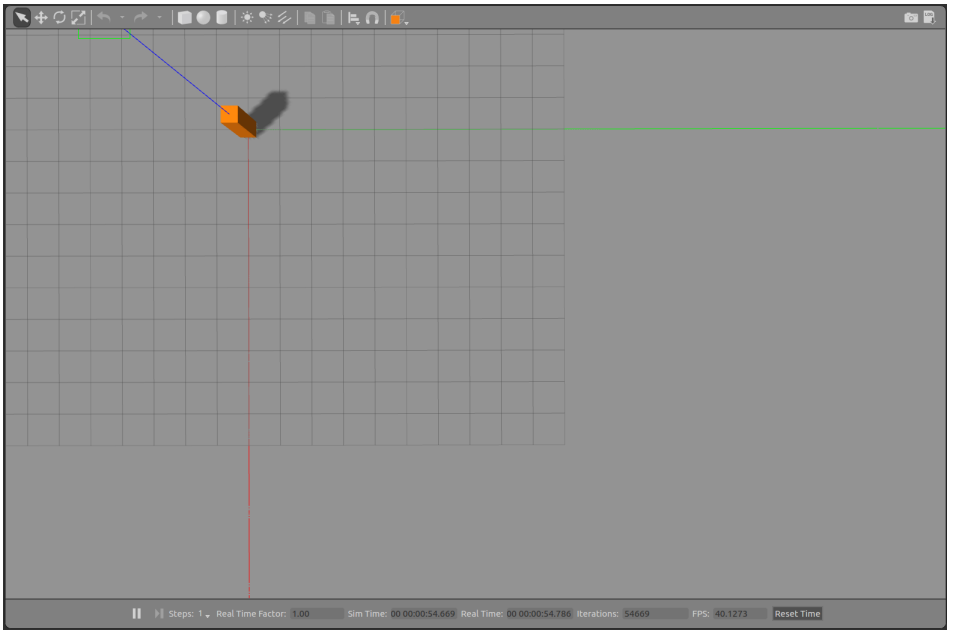
Dummy nodes publishing other necessary messages was made, like the one publishing goal points to surround the ship.

Two screen shots of Gazebo in action can bee seen in Figure 4.1. The tugs are modelled as tall, orange boxes. They are not cylinders as the prototypes are, to make it easy to see their orientation. It should be noted that the simulation is a simplified version of the real world. The tugs move in a constant speed, and exactly to the requested positions. The simulation tool was used only as a means to test the ROS interface and executing trajectories.

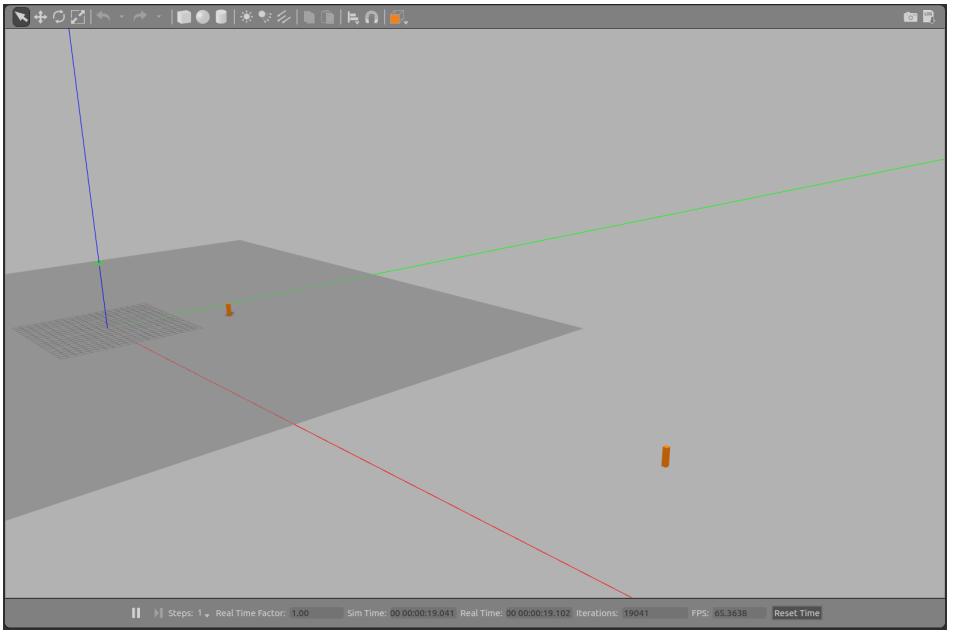
4.3.2 Rationale for Simulation Tool

It was desirable to do testing and simulation with as little modification of the original code as possible. One of the advantages with ROS is that it is easy to test without the need of modifications. A ROS node communicates with other nodes through predefined and broadcasted messages. Nodes are normally not aware of who they are communicating with. Messages needed can therefore easily be faked by a simulator, making nodes execute, just as in a real scenario.

In addition to a simulation, it was chosen to visualize the motion of the tugs. A visualization is highly useful for rapid testing, as it can present information fast, and in an intuitive way. For instance, it is much easier to acquire insight of two tugs relative position with a live image of them, rather than coordinates printed on a screen.



(a) Gazebo top view



(b) Gazebo side view

Figure 4.1: Gazebo simulation and visualization tool. Videos of simulation can be seen by following the instructions of Figure 6.3b

Chapter 5

Implementing Algorithms on a Physical Test setup

This chapter will go through the process of implementing the path planning system on a physical test setup. It will start off by presenting the physical test setup, and explaining how the path planning system communicates both internally and with the external system. The external system refers to the physical test setup and control system, which are Midtskogens work.

Thereafter the first meeting point between the control algorithms and path planning algorithms will be described. Further, it will explain some challenges of moving from a simulation and into the real world. The next section will summarize some benefits of programming in the workshop. The last section explains why the use of visualization was so important.

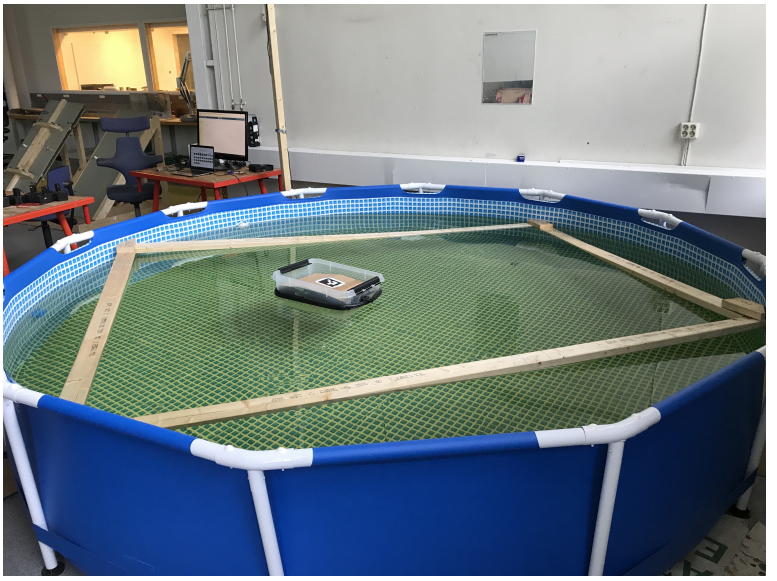
5.1 Physical Test Setup

The test setup is a pool with a 2x3 meter timber frame marking the environment. It is shown in Figure 5.1a. The floating plastic box is the ship. Figure 5.1b shows the same environment, but with the addition of two timber obstacles. The green and white objects are tugs. A closeup image of a tug is shown in Figure 1.1.

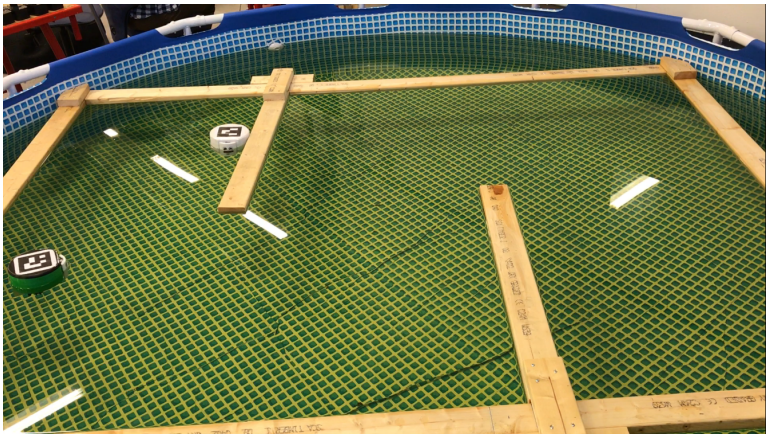
The environment is monitored by a camera. Both the ship and the tugs have unique ArUco markers (Garrido-Jurado et al. (2014)) on top. The markers are detected through the camera by using the computer vision library OpenCV (Bradski (2000)), calculated the markers pose.

5.2 Communication System

An illustration of the communication system is shown in Figure 5.2. The green boxes are Midtskogens responsibility and are regarded as black boxes by this sys-



(a) Environment without obstacles



(b) Environment with obstacles

Figure 5.1: Test setup

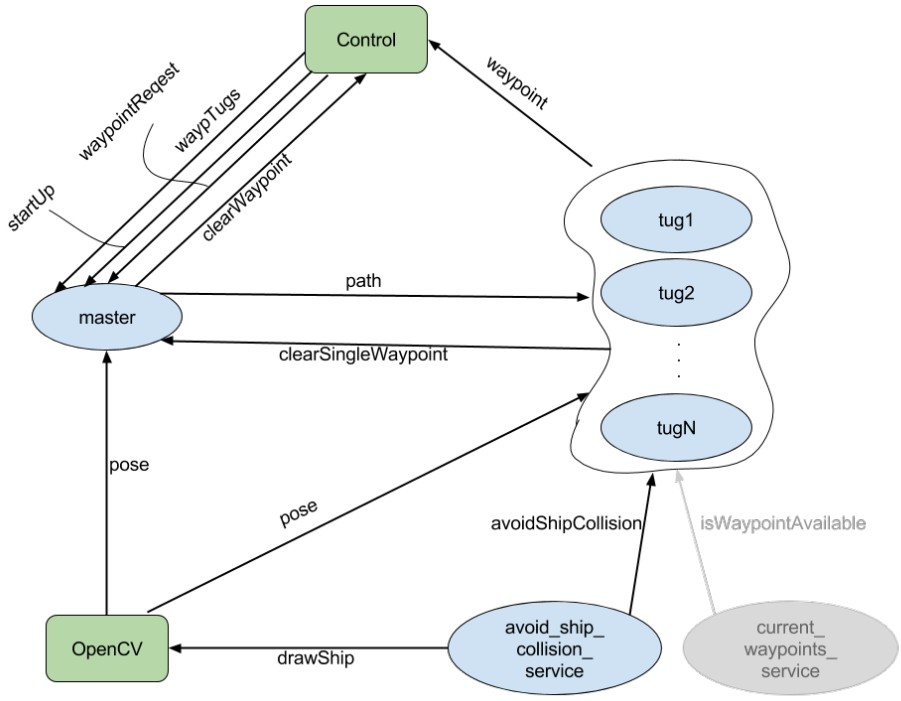


Figure 5.2: Communication diagram

tem. The ellipses represents ROS nodes. The ”tugX”-nodes are meant to illustrate an arbitrary number of tugs. All nodes with arrows pointing into the shape surrounding them are communicating with each tug individually.

An arrow represents a topic or a service. An arrow points at a node which subscribes to a topic or uses a service advertised by the node in the other end. Which topic or service it subscribes to is indicated along the arrow.

The grey node and service are not yet tested in the physical test setup, but is a part of the communication in the simulation program.

5.3 Combining Two Separate Projects

By deciding using ROS (Section 4.2), the interface between the path planning system, OpenCV and the control system should be clear. The same message definitions were used, and the two systems published from the topics agreed upon. This resulted in a more or less pain-free compilation of the combined code.

That being said, both systems had assumptions regarding each other, resulting in several problems when trying to run the combined system for the first time. Some issues only made the tugs act a bit clumsy. For instance, the two systems assumed opposite angles when referring to the orientation of the ship and tugs.

Other problems were fatal, like the timing of at which point the path planning system and the control system switched responsibility of the tugs.

5.4 Simulation vs. Real World

In the simplified simulation of the system (Section 4.3), the tugs moved directly to their assigned waypoints, went directly to the next waypoint without changing speed, and stopped instantly upon arrival. None of the messages were lost by communication errors with the tugs, and the simulation system sent messages needed by the path planning system at the exact right time. This is *not* how the real world works. The following will provide some examples of how the initial path planning system did not cope with the real world.

The communication were not as reliable as assumed. It became very clear after having observed the radio transceivers used in the physical test setup in Figure 5.3. Every time the red light flashed, like on the transceiver on the right, a communication error occurred. For a five minute testrun, it was measured that 14% of all message were dropped. The path planning program sometimes relied on a specific message, for the whole system to work. For instance, the program would not start at all, until position messages of all initiated tugs were received. This is still the case.

Sometimes the position system on the physical test setup believed it detected tugs that did not exist. This confused the path planning system, which wanted to combine the tugs in an optimal way every time a new tug was added to the system.

The tugs did not move in a straight line, as they did in simulation. This was expected, and taken into account in some aspects of the path planning system. However, the irregular movement of the tugs created problems far beyond those anticipated.

In simulation, the goal points for the tugs did not change once they were set. In the test setup, they changed every second. Often just by millimeters, but still, they changed. Because of this, the path planning algorithm updated the combined best path for the new goal points every second. First, the combining algorithm is the most expensive algorithm of the path planning system, and use of it should be kept to a minimum. More important, the tugs need some time to change directions, so a frequent change in waypoints make them act ”wobbly”.

The fact that the goal points *could* change after they were set, was handled to some degree. With a real ship, it was assumed that the maximum movement a ship would make until the tugs arrived, were not big enough to affect the route, other than the section between the last waypoint and the goal point. This turned out to be a poor assumption in the physical test setup. The ”ship” moved much more than expected, without updating the tugs routes.

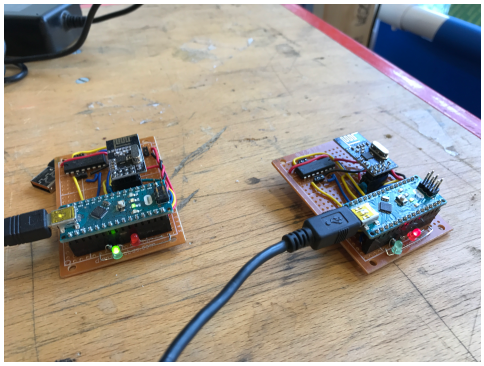


Figure 5.3: Radio transceivers

5.5 Programming in the Workshop

Most of the implementation of the path planning program on the physical test setup, were done by a computer, right next to the setup in the workshop. This made it possible to test the code consecutively.

All the main algorithms were created before the physical test setup and control algorithms were finished. This was due to the fact that the test setup was not ready at the desired point in time. For that reason, only changes in the code regarding communication were done in the workshop. It became clear that some of the algorithms developed, were only clever in theory. For instance, if a tug starts inside a safety area, the system will provide the tug with a waypoint *just* outside the area. The tug will quickly drift back into the safety area, since its control system is not perfect. This scenario and more real world issues, will be elaborated in the Result chapter.

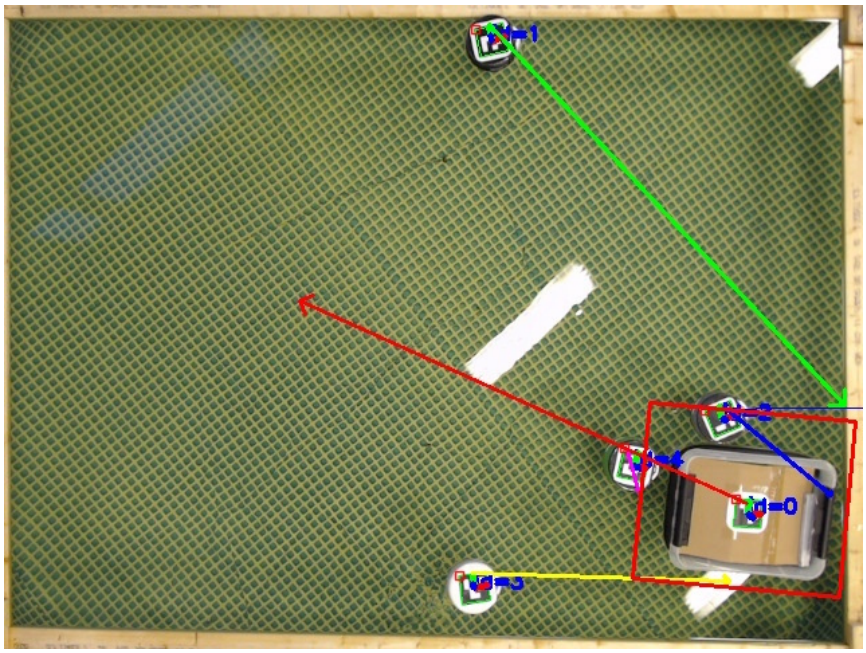
5.6 The Importance of Visualization

When testing the algorithms on the tugs in the pool it was sometimes hard to identify what caused the unexpected behaviour of the tugs. Could it be errors the control system? Did the tug run out of battery? Did the tug receive the wrong waypoint? Even though ROS makes it possible to monitor all messages going through the system, the messages are mostly just numbers, like coordinates, or boat speed. It was hard to find meaningful information from this, especially in real-time.

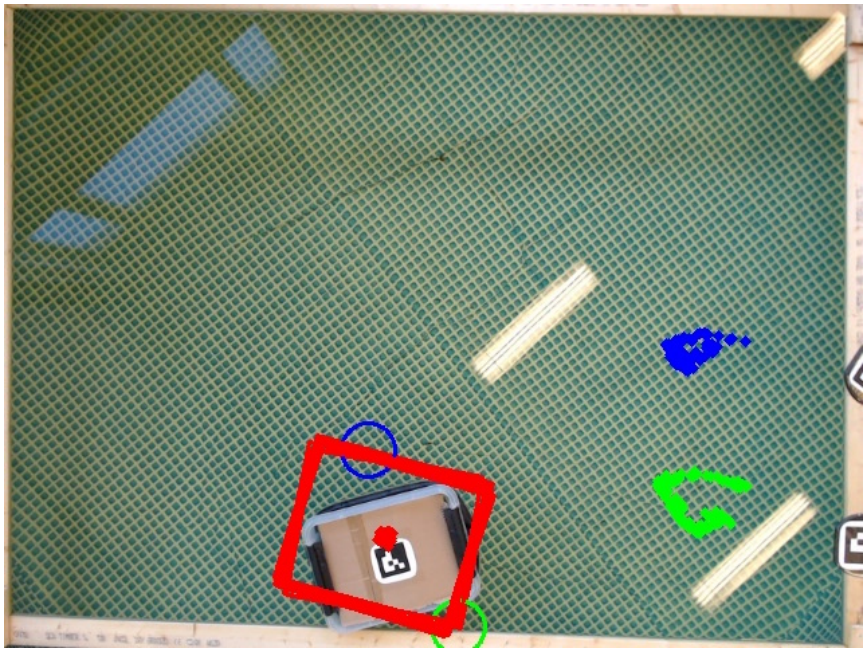
Midtskogen eventually created arrows, pointing towards the current published waypoints, on the live image. An example can be seen in Figure 5.4a The visualization was extremely helpful when debugging.

The angle issue regarding the orientation of ship (section 5.3) was realized after creating a visualization of how the path planning around ship system ”saw” the ship. As shown in Figure 5.4b, the system pinpointed the ship in the opposite

rotation (about the horizontal axis) of its actual orientation.



(a) Visualization of waypoints



(b) Opposite orientation angle

Figure 5.4: Visualization on live videos

Chapter 6

Results and Discussion

Some results from tests executed by the implemented system will be presented.

Finding the shortest routes were tested by drawing the found routes on the corresponding map, and inspecting them. This is presented in the first section. Some of the experiences from testing the ship collision avoidance planner are discussed.

In section two, the efficiency of the speed-up of the search algorithm has been measured. It is compared to two other algorithms by running them in environments of various sizes.

In the following section, the motion planning algorithm is tested. By simulating a scenario of six tugs in Gazebo, it was observed how the tugs interact with each other. The simulation has made it possible to test the system interface before connecting to the real control system.

Eventually, the chapter will lead to a review of a successful demo on a physical test platform, with Kongsberg Maritime in the audience.

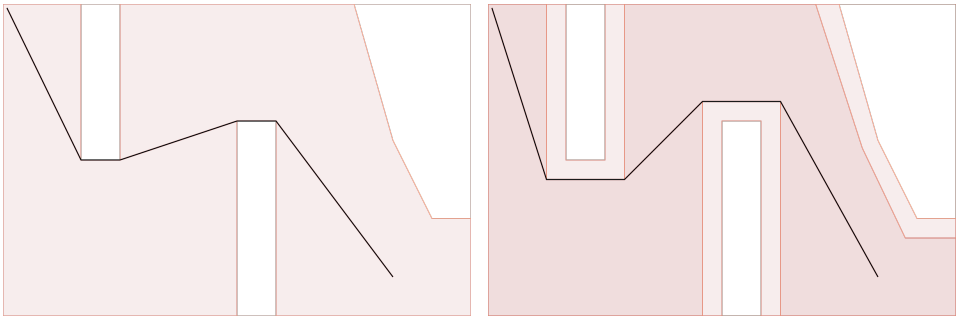
6.1 Route Planning

The route planning system gives the shortest path under certain conditions. The conditions will be presented and discussed, and some results from running the algorithms are illustrated.

6.1.1 Finding Shortest Path in a Static Environment

Shortest, collision free path on Euclidean distance between two points is found every time the algorithm is run, as long as the following preconditions are fulfilled:

- An environment defined by four corner x,y-points.
- Obstacles are defined by sets of x,y-points making up a piecewise linear polygon. The first point connects with the last.



(a) Without safety margin **(b)** With safety margin

Figure 6.1: Shortest path on an environment with few obstacles

- The environment is static.
- The width and/or length (in meters) of the environment is known (for calculating scale).
- The start and end point must be points outside the obstacles, yet they can be within the obstacles safety area.
- The size of the object to plan a path for, like radius of the tug.
- No dynamic objects, other than the tugs and the ship.

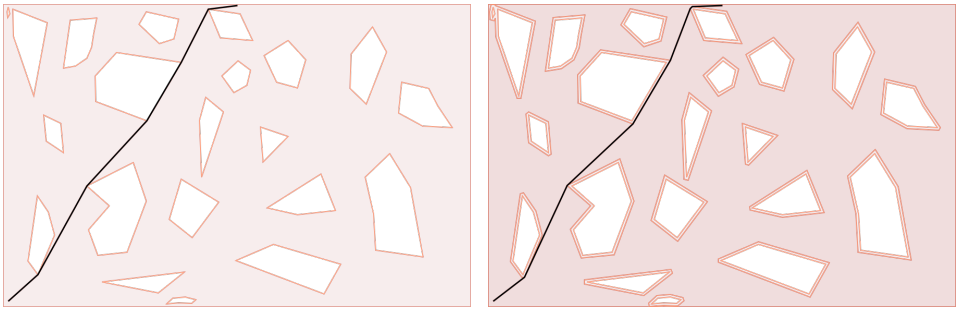
The output of the algorithm is a polyline. The polylines points were printed on screen and drawn and an SVG image together with the environment (section 2.5). Visual inspection was done to confirm the routes were valid, and they also seemed to be the shortest. The algorithms are tested on the following two environments:

1. An environment with a high density of obstacles
2. An environment with few obstacles

The first environment is made to substantiate that the algorithm will provide the shortest path on any environment. A harbor is not likely to have a number of obstacles randomly placed. The second environment is created to be more similar a real harbor. The results of running the shortest path algorithm in the two environments are shown in Figure 6.2 and 6.1.

6.1.2 Route Around Ship

It has been developed a simple and fast algorithm to find route around a ship. The algorithm runs in a constant speed. It has been implemented as a ROS service node.



(a) Without safety margin **(b)** With safety margin

Figure 6.2: Shortest path on an environment with many obstacles

(a) Physical test setup: Five tugs in formation (youtu.be/3ic82NUy_6w)

(b) Gazebo Simulation: Two tugs interacting (youtu.be/HSPQWyEtiXg)

(c) Two tugs in an environment with obstacles (youtu.be/yal7_CQ_8XI)

Figure 6.3: Scan QR code with a QR code reader app on a smart phone to see videos. The app will open the YouTube links in the sub captions of the three codes.

The algorithm works perfectly when the ship is in an area of an environment without obstacles or other tugs. The route planner does not take obstacles or outer boundary into consideration, so it fails if the ship is close to either.

The algorithm does not take other tugs into consideration when planning route for one tug. In the physical test setup, this sometimes caused the tugs to collide when they came close to the ship. However, no group planning were implemented in the test setup. This problem will be taken care of in the motion planning algorithm implemented and tested in simulation. A successful formation can be seen by scanning the QR code in Figure 6.3a.

6.2 Speed Up Guidance Algorithm

It was chosen to speed up of the main task of this study, which is to find the shortest path in real-time.

The first step towards increasing speed, was to calculate the visibility graph of the environment only once, not on every inquiry. This was possible without reducing the quality of the path, as the tugs are always operating in the same environment.

The second improvement was to calculate the shortest path between all the vertices in the environment in advance. This will not reduce the quality of the path, either. The paths are still the same, just precalculated.

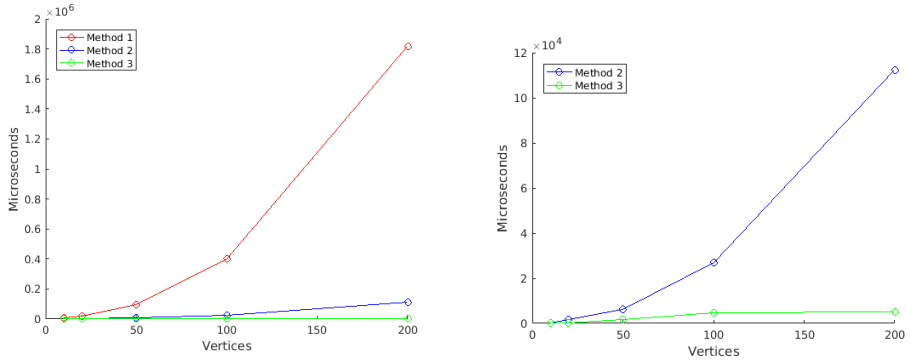
To summarize, the following methods have been tested:

1. Calculating visibility graph repeatedly and search using A* searching algorithm
2. Precalculate visibility graph and search with A*
3. Precalculate all-pair-shortest path, and only search through the start and goal points' visibility polygons

The methods will be referred to as method 1, method 2 and method 3, respectively.

To quantify the improvements of the two speed-ups, the methods have been tested in five different environments, with an increasing number of nodes. The environments were constructed by using Environment Creator (2.7) and making a set of obstacles with a predetermined number of points. Start and end point were chosen to be set far from each other. Images of the environments with the shortest path can be found in appendix B.

Tee methods has been tested on a PC with IntelÂő Xeon(R) CPU E3-1271 v3 @ 3.60GHz x 8, with an dedicated Gallium 0.4 on NVE7 graphics card. The computation times of three different methods can be found in Table 6.1. Figure 6.4 shows graphs of the same computation times.



(a) All three methods **(b)** The two best methods

Figure 6.4: Computation times of three different methods

Table 6.1: Computation times of three different methods

Vertices in total	Time (1) [ms]	Time (2) [ms]	Time (3) [ms]
10	5.792	0.366	0.096
20	19.66	1.559	0.258
50	97.75	6.340	1.553
100	400.1	26.97	4.833
200	1819	112.2	5.150
Average	468.5	29.49	2.378

As shown in Figure 6.4a, the first method is exponential, and not scalable at all, hence the reason why it was chosen to be improved.

Figure 6.4b shows a close up of method 2 and 3. It is clear that an increasing number of vertices increases the computation time of method 2. The final, and best method in for all environments, is method 3. It increases slowly in the number of vertices, mostly because the start and goal point has more vertices in their visibility polygons.

For the five environment tested, method 3 is 197 times faster than method 1 on average. Method 3 is on average 12.3 times faster than method 2. Both are impressive numbers in theory, but in reality, only the difference between method 1 and 3 is noticeable. Even though method 3 is 12.3 faster than method 2, the difference between them are only milliseconds. However, for N tugs in the system, the shortest path algorithm will run N times every time a tug is connected or disconnected to the system. If the system had 200 vertices, only *nine* tugs are needed for the execution time to reach 1 second, *before* the time of the Kuhn-Munkres algorithm is added. This would be noticeable and even problematic in real time.

The speed-up of precalculating all-pairs shortest path (method 3) is remarkable, and is obviously chosen for use in the final system.

6.3 Simulation: System Interface and Tug Interaction

The interface was tested in a simulation before it was tested in the physical test setup. The following scenario was tested. Of six randomly placed tugs, four tugs had to fill four locations around a imaginary ship. These positions was not known until run time. The scenario was tested in the environments in Figure 6.1a.

The result of the simulation was as expected. Three tugs went towards their best calculated next waypoint. One tug waited until its waypoint was available. When the waiting tug started, it did not stop until it had reached its goal. All tugs followed their path. None of the tugs paths crossed, nor did they collide.

Two of the tugs had four of the same waypoints. A 27 second video of the two tugs can be seen by scanning the QR code in Figure 6.3b. One of the tugs wait for the other tug, as expected. The distance between the start point and first waypoint of the tug starting in (0,0) is the longest of all waypoint-waypoint distances. The tug had to wait for such a long distance, and that is why it never reached the other tug. The suggested motion planning approach (only one tug approaching a waypoint at a time) seems to be a sturdy solution in simulation.

6.4 Physical Test Set-up: A Successful Demo

8th of June, 2017, a demo was held at the workshop at MTP Gløshaugen. Espen Strange from Kongsberg Marine, supervisor Martin Steinert, and some students interested in the study, were present.

In the first scenario, five tugs were in the pool. Four were requested to surround the ship, one on each side. Keep in mind that the ship is a floating plastic box. The environment was empty, i.e. without obstacles. Only the outer boundary was framed. Initially, the ship was located at approx the center of the environment. The tugs all chose the point closest to them, as expected. The tugs which had to move towards one side of the ship, went via the corner giving the shortest path. After the arrival of two tugs, the system released these tugs to the control system, as expected. The two remaining tugs were released to the control system consecutively as they arrived.

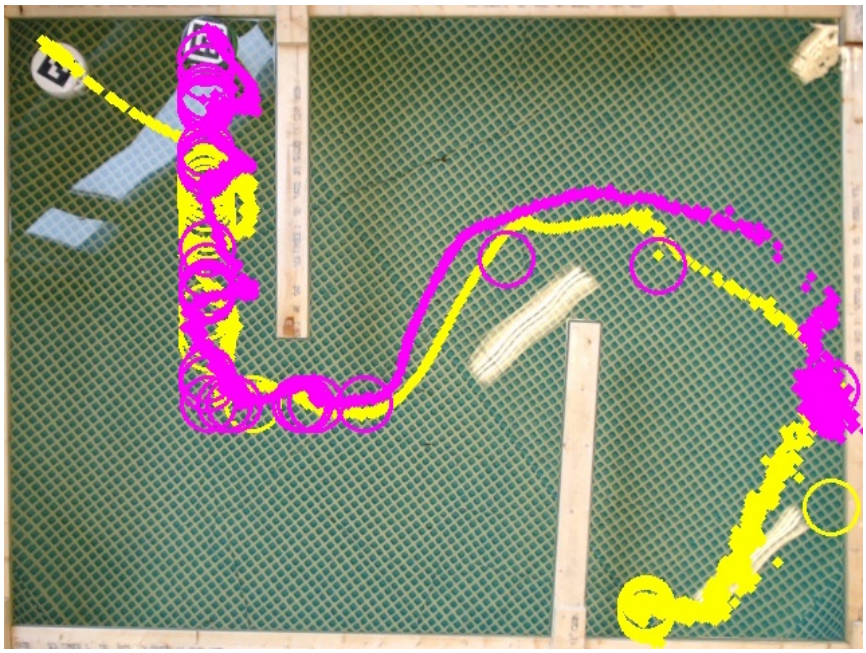
In the second scenario, the path planning algorithms were tested to a greater extent. The ship was removed, but two obstacles were added to the environment. See Figure 6.5. Two tugs were placed in one side of the environment. At run time, the tugs were requested to move towards two points on the opposite side of the environment. This required the system to plan routes based on its internal map of the environment, and release waypoints at the right time.

The tugs moved from start to goal as expected. However, there are three issues that need to be addressed. First, the tugs sometimes overshoots when trying to reach a waypoint. Since they did not pass the waypoints' acceptance radius, they have to turn around and go back. This occurs even though they are on their way to the next waypoint when they overshoot.

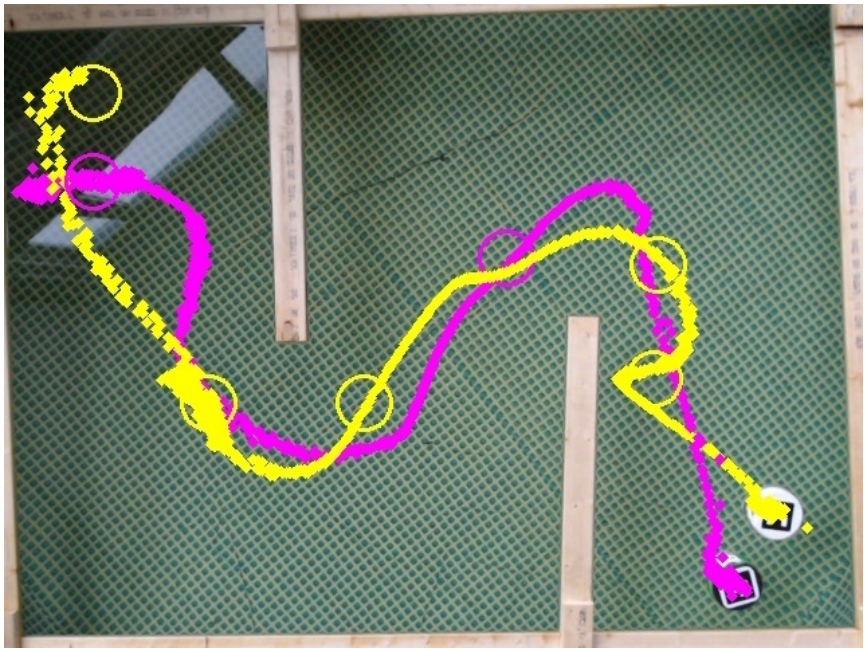
The second issue is that the tugs tend to collide a lot. This occurs when they are approaching the same waypoint. In Figure 6.5a, this is an obvious problem in the beginning (left in image). This issue is addressed in the simulation code, but time did not allow it to be implemented on the physical test setup before the demo. Even though it was not implemented, it was demonstrated that collision avoidance algorithms are needed. Also it seems only to allow one tug to move to a waypoint at the time to be a good solution.

The final issue regards how to handle points within safety margins of the obstacles. As seen in Figure 6.5a, the tugs are bouncing back and fourth into the safety margin in the start phase. Every time they are inside, they receive a new waypoint *just* outside the safety margin. This leaves no room for drifting when heading towards the next waypoint, and hence, it recalculates its route every time it drifts back. A solution to this problem could be to give a waypoint further from the safety margin from the very beginning. Figure 6.5b shows a test-run where the tugs did not drift into the safety margin. The demonstration was unfortunately not caught on tape, but a successful test-run can be seen by scanning the QR code in Figure 6.3c.

Despite the addressed issues, the demonstration went really well. As seen in Appendix A, Kongsberg was very interested and impressed by it. The tugs completed their given tasks, although in a kind of clumsy manner. Discovering the issues presented above so quickly is *exactly* why the physical test setup is great for rapid testing of code.



(a) Path at demo



(b) Path of tugs outside of safety margin

Figure 6.5: Paths of two test runs

Chapter 7

Conclusions

The main goal of this study was to research for and develop algorithms for path planning, and ultimately implementing them on a physical test set-up. The following will highlight the most important features of the system, and illustrate how the results contribute towards further development in this study of autonomous tugboats or other similar researches.

An optimal algorithm for finding the shortest path in an environment of obstacles has been developed. The algorithm uses the A* search algorithm in a visibility graph. Within the visibility graph of an environment, the shortest path on Euclidean distance is guaranteed to be found, if such a path exists. This characteristic is provided at the expense of the smoothness of the path. The large angle between the waypoints cause the tugs to make abrupt movements.

In the tugboatsÂt environment, a harbor, the position of the static obstacles will be known beforehand. This is taken advantage of in the path planning algorithm, by creating a special type of look-up table of âĂIJall-to-allâĂİ shortest path. This look-up table is inspired by literature, but new in the context of a road map-approach, and the way it saves memory[1]. The look-up table might be used in any road map-approach. This is of value to other researchers, and makes it easy to replace the visibility graph in the future.

A new and effective way to find the shortest path around a ship, is presented. In this study, the ship had to be simplified to a four-sided rectangle, and its position and orientation had to be known. This low-cost method worked well in the obstacle-free scenario of the physical test set-up. In the scenario with obstacles, the algorithm chose paths intersecting with obstacles. The method should be extended to include an intersection test with environment obstacles and outer boundary.

The fact that the paths of two robots (of zero size) will never cross has been proved, provided they both follow the shortest, combined road map-approach path. Methods to account for the robot having a certain size have also been suggested.

Making use of this proof, a method to execute trajectories for a group of tugs

[1] to the best of the authors knowledge

has been developed, which makes it possible to avoid robot-robot communication completely. The method checks for waypoint occupation before releasing the waypoint, rather than constantly checking for robot-robot collision. This results in an efficient and scalable method. The method has been tested successfully in a simulation.

Finally, the path planning system has been implemented on a physical test set-up. The test set-up will be of great advantage in further development of autonomous tugboats. It is low-cost, easy to use, and makes it possible to test new features on a daily basis. By doing rapid and continuous tests, issues typically found in real life systems can become evident at an early stage.

Chapter 8

Outlook

The path planning algorithms implemented are not suited for a real harbor. First and foremost, obstacle detection and avoidance is needed in order to avoid collision with other boats or dynamic obstacles. The tugs would need more sensors in order to implement a working ODA.

The tugs would have to follow the rules of the sea. Boats are, for instance, required to pass other boats in a certain manner.

The tugs created in the future is not necessarily disk-shaped.

In order to still use a safety margin to account for the size of the tugs, the safety margin must be half of the largest dimension, not the radius. This will probably increase the safety area. A large safety area may cause areas to merge. Paths in between would not be found, even though the ship would be able to pass by moving with a certain orientation.

At at this point, paths are calculated by a master. The path planning system is made so that they could plan their own path if they had a more powerful processor. Tugs should be able to do path calculations onboard in the future.

The paths calculated sometimes require the tugs to do sharp turns. This is mainly because of the way the visibility graphs are created, but also due to how the tugs move out of safety areas.

There are other existing road map approaches which are smoother than the visibility graph. For instance, a hybrid between the Voroni diagram and visibility graph, called the Voroni-visibility diagram (Wein et al. (2007)). By using this approach, the guarantee of finding the shortest path on Euclidean distance is no longer valid. However, it might give quicker paths, since smoother paths are easier for the tugs to execute. In order to implement a Voroni-visibility diagram, the tugs must be able to follow paths, not only follow waypoints, as they currently do. This is a task for the control system.

Another solution which might work with the current control system, is illustrated in Figure 8.1. The red points and lines represents the path around an obstacle. In order to make the path smoother, it is possible to add more waypoints,

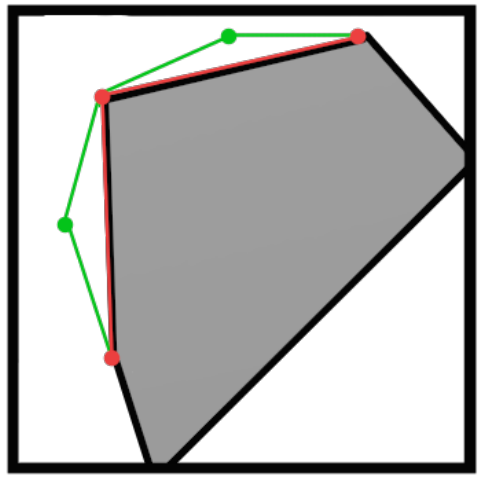


Figure 8.1: Add waypoints for a smoother path

for instance, by adding the green points to the original path.

A working method for planning routes for tugs is presented, but a method to plan for a ship is also needed in the near future. Many principles from this system might be used, but it needs to be extended. For instance, the ship is probably not disk-shaped, and the same modifications as for tugs with a different shape, has to be done.

In addition, the ship orientation must in some cases be planned for. For instance, when docking the ships' long side against a dock, it needs to turn before reaching its final waypoint.

Bibliography

Azarm, K., Schmidt, G., 1996. A decentralized approach for the conflict-free motion of multiple mobile robots. Advanced robotics 11 (4), 323–340.

Bennewitz, M., Burgard, W., Thrun, S., 2002. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. Robotics and Autonomous Systems 41 (2-3), 89–99.

Blaich, M., Rosenfelder, M., Schuster, M., Bittel, O., Reuter, J., 2012. Fast Grid Based Collision Avoidance for Vessels using A * Search Algorithm. 17th International Conference on Methods and Models in Automation and Robotics (MMAR), 385 – 390.

Bradski, G., 2000. Opencv. Dr. Dobb's Journal of Software Tools.

Buniyamin, N., J, W. N. W. A., Sariff, N., Mohamad, Z., 2011. A Simple Local Path Planning Algorithm for Autonomous Mobile Robots. International journal of systems applications, Engineering & development 5 (2).

Campbell, S., Naeem, W., Irwin, G. W., 2012. A review on improving the autonomy of unmanned surface vehicles through intelligent collision avoidance manoeuvres. Annual Reviews in Control 36 (2), 267–283.

Casalino, G., Turetta, A., Simetti, E., 2009. A three-layered architecture for real time path planning and obstacle avoidance for surveillance USVs operating in harbour fields. OCEANS '09 IEEE Bremen: Balancing Technology with Future Needs.

Dong, P., 2008. Generating and updating multiplicatively weighted voronoi diagrams for point, line and polygon features in gis. Computers & Geosciences 34 (4), 411–421.

Dorigo, M., Birattari, M., Blum, C., Clerc, M., Stützle, T., Winfield, A., 2008. Ant Colony Optimization and Swarm Intelligence: 6th International Conference, ANTS 2008, Brussels, Belgium, September 22-24, 2008, Proceedings. Vol. 5217. Springer.

Doriya, R., Mishra, S., Gupta, S., 2015. A brief survey and analysis of multi-robot communication and coordination. International Conference on Computing, Communication and Automation, ICCCA 2015, 1014–1021.

Erdmann, M., Lozano-Perez, T., 1986. On multiple moving objects. IEEE International Conference on Robotics and Automation 3, 1419–1424.

Garrido-Jurado, S., noz Salinas, R. M., Madrid-Cuevas, F., Marín-Jiménez, M., 2014. Automatic generation and detection of highly reliable fiducial markers under occlusion. Pattern Recognition 47 (6), 2280 – 2292.
URL http://www.sciencedirect.com/science/article/pii/S0031320314000235

Gerkey, B. P., Matarić, M. J., 2002. Sold!: Auction methods for multirobot coordination. IEEE Transactions on Robotics and Automation 18 (5), 758–768.

Guzman, D. D., Dellosa, S., Inventado, P., Lao, P., 2005. A * Path Planning in Real-time Dynamic Environments. Conference paper: 5th Philippine Computing Science Congress.

Johnson, A., 2014. Clipper - an open source freeware library for clipping and offsetting lines and polygons. http://www.angusj.com/delphi/clipper.php, accessed: 2017-02-13.

Jouandeau, N., Yan, Z., 2012. Decentralized Waypoint-based Multi-robot Coordination. In: Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2012 IEEE International Conference on. pp. 175–178.

Khamsi, M. A., Kirk, W. A., 2011. An introduction to metric spaces and fixed point theory. Vol. 53. John Wiley & Sons.

Koenig, N., Howard, A., 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on. Vol. 3. IEEE, pp. 2149–2154.

Larson, J., Bruch, M., Ebken, J., 2006. Autonomous navigation and obstacle avoidance for unmanned surface vehicles. Proceedings of SPIE 6230 I (2006), 17–20.

LaValle, S., Hutchinson, S. a., 1998. Optimal motion planning for multiple robots having independent goals. Robotics and Automation, IEEE ... 14 (6), 912–925.

Liu, Z., Zhang, Y., Yu, X., Yuan, C., 2015. Unmanned surface vehicles: An overview of developments and challenges. Annual Reviews in Control 41, 71–93.

Lozano-Pérez, T., Wesley, M. A., 1979. An algorithm for planning collision-free paths among polyhedral obstacles. Communications of the ACM 22 (10), 560–570.

Maren, B., Wolfram, B., Sebastian, T., 2001. Constraint-based Optimization of Priority Schemes for Decoupled Path Planning Techniques. KI 2001: Advances in Artificial Intelligence, 78—-93.

Munkres, J., 1957. Algorithms for the assignment and transportation problems. Journal of the society for industrial and applied mathematics 5 (1), 32–38.

Niu, H., Lu, Y., Savvaris, A., Tsourdos, A., 2016. Efficient Path Planning Algorithms for Unmanned Surface Vehicle. IFAC-PapersOnLine 49 (23), 121–126.

Nodes, 2012. `http://wiki.ros.org/Nodes`, accessed: 2017-05-29.

Norge i bilder, 2016. `http://www.norgeibilder.no`, accessed: 2017-04-10.

Obermeyer, K. J., Contributors, 2008. The VisiLibity library. `http://www.VisiLibity.org`, release 1.

Russell, S. J., Norvig, P., 2014. Artificial intelligence: a modern approach (3rd edition). Prentice Hall.

Services, 2012. `http://wiki.ros.org/Services`, accessed: 2017-05-29.

Signifredi, A., Luca, B., Coati, A., Medina, J. S., Molinari, D., 2015. A general purpose approach for global and local path planning combination. In: Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on. IEEE, pp. 996–1001.

Svec, P., Gupta, S. K., 2012. Automated synthesis of action selection policies for unmanned vehicles operating in adverse environments. Autonomous Robots 32 (2), 149–164.

Understanding ROS Nodes, 2016. `http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes`, accessed: 2017-05-29.

van den Berg, J. P., Overmars, M. H., 2005. Prioritized motion planning for multiple robots. 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems.

Wein, R., Van den Berg, J. P., Halperin, D., 2007. The visibility–voronoi complex and its applications. Computational Geometry 36 (1), 66–87.

Wu, B., Wen, Y., Huang, Y., Zhu, M., 2013. Research Of Unmanned Surface Vessel (USV) Path-Planning Algorithm Based on ArcGIS. ICTIS 2013@sImproving Multimodal Transportation Systems-Information, Safety, and Integration, 2125–2134.

Yan, Z., Jouandeau, N., Cherif, A. A., 2013. A survey and analysis of multi-robot coordination. International Journal of Advanced Robotic Systems 10.

Yu, J., Lavalle, S. M., 2013. Multi-agent Path Planning and Network Flow. Algorithmic Foundations of Robotics X 86, 157–173.

Zhuang, J. Y., Su, Y. M., Liao, Y. L., Sun, H. B., 2011. Motion planning of USV based on Marine rules. Procedia Engineering 15, 269–276.

Appendix

Appendix A

Feedback from Kongsberg Maritime

An email from Espen Strange (Kongsberg Maritime) regarding the live demo 8th of june 2017.

Gmail Rebecca Cox <rebcoxx@gmail.com>

Autonomous Tugs Project

1 e-post

Espen Strange <espen.strange@km.kongsberg.com> 9. juni 2017 kl. 11:51
Til: Sondre Næss Midtskogen <sondrenm@stud.ntnu.no>, Rebecca Cox <rebcoxx@gmail.com>
Kopi: Martin Steinert <martin.steinert@ntnu.no>, Arne Rinnan <arne.rinnan@km.kongsberg.com>

Hei Sondre og Rebecca!

Thank you for a very interesting demonstration of the swarm based autonomous tug bots and the test setup that you have developed. The project has taken a very interesting approach in terms of using swarm based behavior in autonomous tug bot operation and at the same time keeping system complexity to a necessary minimum. In terms of an actual operation I think you have worked on solutions to tackle several relevant aspects such as bot positioning, navigation, obstacle avoidance and distribution of bots around the vessel. You have also run into unforeseen problems and behavior which is of interest in further iteration. You have also demonstrated what seems to be a very efficient test setup where changes can be quickly implemented and tested allowing for very fast iterative development and at the same time provide a platform for deeper learning and research. I found this project results so far very interesting for both actual development and further research into the field of autonomy - well done!

Med vennlig hilsen / Best regards,

Espen Strange

Senior Designer

Products and Services

Kongsberg Maritime

Phone +47 33 03 20 00

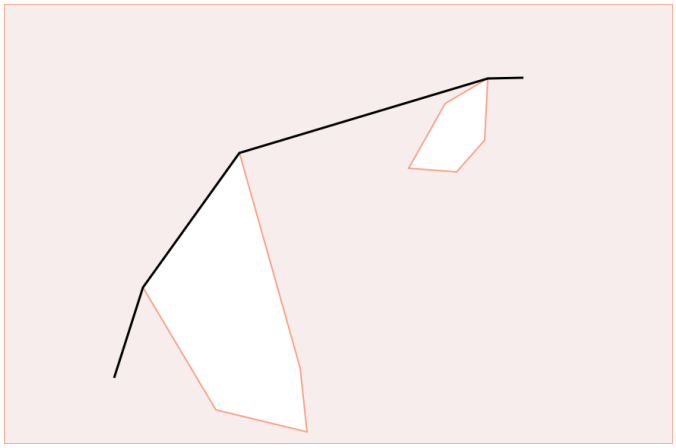
Mobile +47 48 08 46 59

espen.strange@km.kongsberg.com

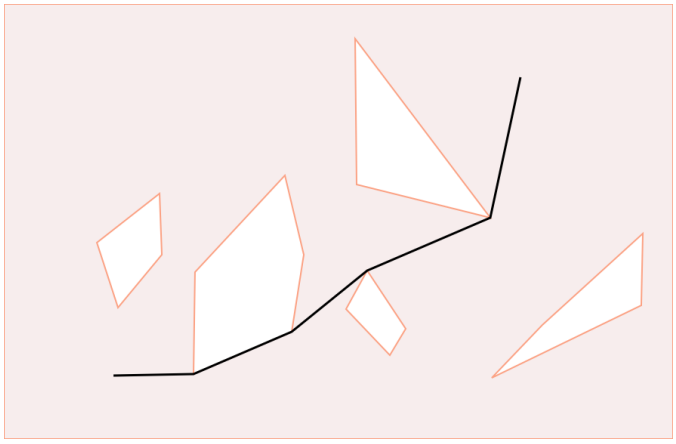
www.kongsberg.com

Appendix B

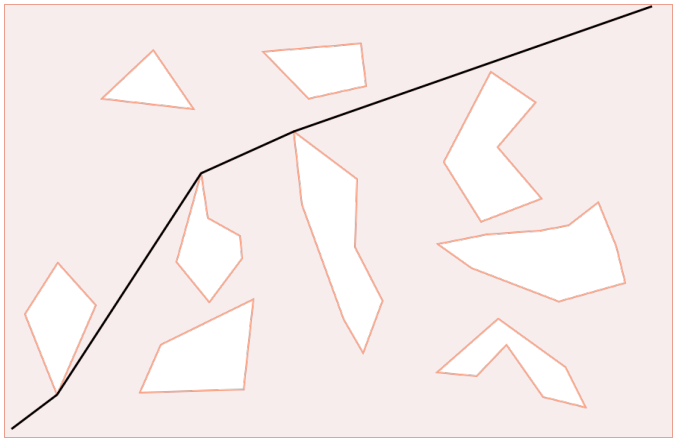
Figures



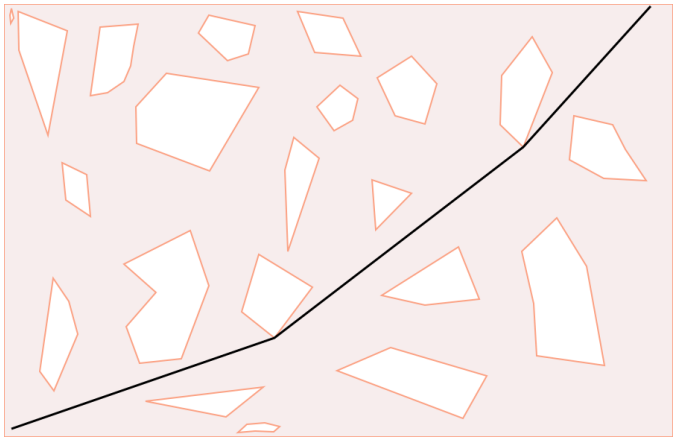
(a) Environment with 10 vertices



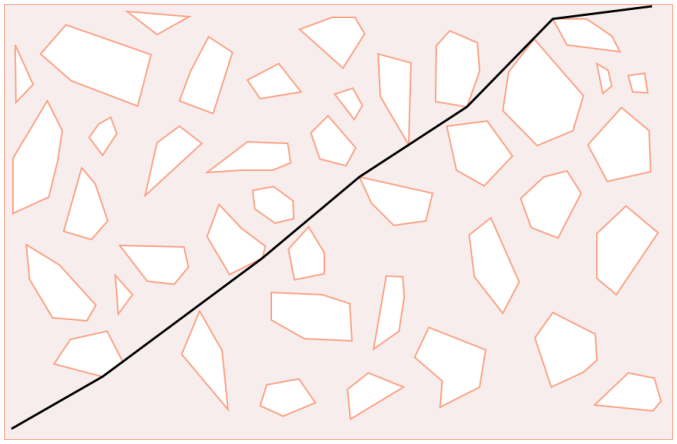
(b) Environment with 20 vertices



(c) Environment with 50 vertices



(d) Environment with 100 vertices



(e) Environment with 200 vertices

Figure B.1: Testing environments with shortest path

Appendix C

Code

The code is sorted on ROS packages. The packages are:

- Master
- Geometry
- Search
- Coordination
- Tug Gazebo

C.1 Master

Contents

- tug_communicator.hpp
- tug_constants.hpp
- tug_waypoint_publisher.hpp
- avoid_ship_collision_service.cpp
- current_waypoints_service.cpp
- master_node.cpp
- tug_communicator.cpp
- tug_waypoint_publisher.cpp
- munkres_speed_test.cpp
- test_shortest_path.cpp
- test_speed.cpp

C.1.1 include

tug_communicator.hpp

```
#include "coordination/tug_assign_paths.hpp"
#include "geometry/tug_boat.hpp"
#include "geometry/tug_environment.hpp"
#include "search/tug_route_around_ship.hpp"
#include "search/tug_shortest_path.hpp"
#include "std_msgs/UInt8.h"
#include "tug_constants.hpp"
#include "tugboat_control/BoatPose.h"
#include "tugboat_control/ClearWaypoint.h"
#include "tugboat_control/Path.h"
#include "tugboat_control/Waypoint.h"

#include <memory>
#include <ros/package.h>
#include <ros/ros.h>
#include <sstream>
#include <std_msgs/UInt8MultiArray.h>
#include <string>

namespace Tug
{
  class Communicator
  {
  public:
          Communicator(Environment &environment, double scale, double accept_waypoint_radius);
          void print_path(Tug::Polyline path);
```

```
    void replan();
    tugboat_control::Path polyline_to_path_msg(const Tug::Polyline &path,
                                               int tug_id, int order_id);
    void polyline_to_path_msg(const Tug::Polyline &path, int tug_id, int order_id,
                tugboat_control::Path &path_msg);
    void remove_end_point_from_planner(const tugboat_control::ClearWaypoint::ConstPtr &msg);
    void publish_new_waypoint(const Tug::Point pt_cur, int tug_id);
    int  find_order_id(const Tug::Point &pt);
    void remove_tug_from_control(int tug_id);
    void add_new_tug(int id);
  bool tug_id_already_in_system(int id);
  bool tug_is_under_my_control(int id);
  void replan_route_for_one_boat(int msg_id, const Tug::Point &newGoal);
    void callback_waypoint(const tugboat_control::Waypoint::ConstPtr& msg);
    void callback_boat_pose(const tugboat_control::BoatPose::ConstPtr& msg);
    void callback_ship_pose(const tugboat_control::BoatPose::ConstPtr &msg);
    void callback_available_tugs(const std_msgs::UInt8MultiArray::ConstPtr &msg);
  void callback_new_tug(const std_msgs::UInt8::ConstPtr &msg);

private:
      Tug::Environment environment_tug_;
          double accept_waypoint_radius_;
      double scale_;
    std::shared_ptr<Assign_paths> assigner_ptr_;

      ros::NodeHandle node_;
      ros::Publisher tug_arrived_pub;
      ros::Publisher ship_waypoint_pub;
      ros::Publisher path_pub;
```

```
    std::map<int, Tug::Boat> tugs_;
    std::map<int, Tug::Point> end_points_;
    std::vector<int> tugs_under_my_control_;
    std::vector<tugboat_control::ClearWaypoint> order_ready_to_publish;
    std::map<int,int> msg_and_tug_;

};

}
```

tug_constants.hpp

```
#ifndef SIMULATION
#define SIMULATION true
#endif

//Gazebo: whatever fits environment
//Physical test setup: 220
#ifndef SCALE
#define SCALE 40
#endif

//Gazebo: SCALE_OUT = 1
//Physical test setup: Same as SCALE
#ifndef SCALE_OUT
#define SCALE_OUT 1
#endif

#ifndef TUG_SPEED
#define TUG_SPEED 8
#endif

#ifndef CLOSE_POINTS_RADIUS
#define CLOSE_POINTS_RADIUS 0.1
#endif
```

tug_waypoint_publisher.hpp

```
#ifndef WAYPOINT_PUBLISHER_H
#define WAYPOINT_PUBLISHER_H

#include "tugboat_control/WaypointAvailable.h"
#include "tugboat_control/AvoidShipCollision.h"
#include "tugboat_control/Path.h"
#include "tugboat_control/ClearWaypoint.h"
#include "tugboat_control/BoatPose.h"
#include "tugboat_control/Waypoint.h"
#include "tug_constants.hpp"

#include <memory>
#include <ros/ros.h>
#include <std_msgs/UInt8.h>

namespace Tug
{
  class Waypoint_publisher
  {
  public:
    Waypoint_publisher(int id, double scale);

    void set_path(const tugboat_control::Path::ConstPtr &msg);
    void update_position(const tugboat_control::BoatPose::ConstPtr &msg);
    void set_id(int id){id_ = id;};
    int id() const {return id_;};
    void call_path_around_ship_service(const tugboat_control::Waypoint &start,
                                       const tugboat_control::Waypoint &finish,
```

```
                                        std::vector<tugboat_control::Waypoint> &result);

    tugboat_control::BoatPose get_position(){return newest_pose_;}
    std::vector<tugboat_control::Waypoint> get_path() const {return path_;}
    tugboat_control::Waypoint get_goal(){return path_.back();}
    bool is_waypoint_available(const tugboat_control::Waypoint &pt);
    void publish_current_waypoint();
    int no_waypoints_left(){return path_.size() - current_waypoint_index_;};

  private:
    int id_=-1;
    std::vector<tugboat_control::Waypoint> path_;
    int current_waypoint_index_ = 0;
    bool go_to_next_waypoint();
    bool is_already_on_hold = false;

    double scale_;
    double acceptance_radius;

    tugboat_control::BoatPose newest_pose_;
    int order_id_;


    ros::NodeHandle node_;
    ros::Publisher wayp_pub;
    ros::Publisher arrival_pub;

    ros::ServiceClient client_is_avail;
    ros::ServiceClient client_avoid_ship;
  };
```

```
}

#endif //WAYPOINT_PUBLISHER_H
```

C.1.2 src

avoid_ship_collision_service.cpp

```
#include "ros/ros.h"
#include "tugboat_control/AvoidShipCollision.h"
#include "tugboat_control/BoatPose.h"
#include "geometry/tug_polyline.hpp"
#include "search/tug_route_around_ship.hpp"
#include "std_msgs/Float64MultiArray.h"
#include "tug_constants.hpp"
//#ifndef SCALE
//#define SCALE 220.0
//#endif

namespace
{
  Tug::Route_around_ship route_around_ship_(0, 0.6*SCALE, 0.7*SCALE);
  tugboat_control::BoatPose newest_pose_msg_;
  ros::Publisher tug_corner_pub;
}

void callback_ship_pose(const tugboat_control::BoatPose::ConstPtr &msg)
{
  tugboat_control::BoatPose new_bp;
  new_bp.x = msg->x*SCALE;
  new_bp.y = msg->y*SCALE;
  new_bp.o = msg->o;
  new_bp.ID = msg->ID;
        newest_pose_msg_ = new_bp;
```

```
  //2 next lines: Move ship so it is drawn on screen. Not necessary for service to work
  Tug::Point pt(newest_pose_msg_.x, newest_pose_msg_.y, -1);
  route_around_ship_.move(pt, newest_pose_msg_.o);

  std::vector<double> points = route_around_ship_.ship_corners();
  if (points.size()>0)
  {
    std_msgs::Float64MultiArray points_msg;
    points_msg.data = points;
    tug_corner_pub.publish(points_msg);
  }
}

bool find_path(tugboat_control::AvoidShipCollision::Request &req,
                           tugboat_control::AvoidShipCollision::Response &res)
{
        double speed = req.from.v;
        Tug::Point pt(newest_pose_msg_.x, newest_pose_msg_.y, -1);

        route_around_ship_.move(pt, newest_pose_msg_.o);
  Tug::Point pos = route_around_ship_.get_ship_position();

        Tug::Point start(req.from.x, req.from.y, -1);
        Tug::Point end(req.to.x, req.to.y, -1);

  Tug::Polyline points_to_move_around_ship = route_around_ship_.best_route(start, end);
  res.path.push_back(req.from);
  for (int i = 0; i < points_to_move_around_ship.size(); ++i)
  {
          tugboat_control::Waypoint wp;
```

```
            wp.x = points_to_move_around_ship[i].x();
            wp.y = points_to_move_around_ship[i].y();
            wp.v = speed;
            res.path.push_back(wp);
        }
        res.path.push_back(req.to);
        return true;
    }

    int main(int argc, char **argv)
    {
        ros::init(argc, argv, "avoid_ship_collision_service");

        ros::NodeHandle node;
        newest_pose_msg_.x = -1000;
        newest_pose_msg_.y = -1000;
        newest_pose_msg_.o = 0;
        ros::Subscriber sub_shipPose = node.subscribe("shipPose", 1, callback_ship_pose);

        ros::ServiceServer service = node.advertiseService("avoidShipCollision", find_path);


        ros::NodeHandle node_;
        tug_corner_pub = node_.advertise<std_msgs::Float64MultiArray>("drawShip", 100);

        ros::spin();

            return 0;
    }
```

current_waypoints_service.cpp

```
#include "ros/ros.h"
#include "std_msgs/Bool.h"
#include "tug_constants.hpp"
#include "tugboat_control/Waypoint.h"
#include "tugboat_control/WaypointAvailable.h"

namespace
{
  std::map<int, tugboat_control::Waypoint> waypoints_;
}

void set_waypoint(const tugboat_control::Waypoint::ConstPtr &msg)
{
  int id = msg->ID;
  try
  {
    tugboat_control::Waypoint scaledpt;
    scaledpt.x = msg->x*SCALE_OUT;
    scaledpt.y = msg->y*SCALE_OUT;
    waypoints_.at(id) = scaledpt;
  }
  catch(const std::out_of_range &oor)
  {
    tugboat_control::Waypoint scaledpt;
    scaledpt.x = msg->x*SCALE_OUT;
    scaledpt.y = msg->y*SCALE_OUT;
    waypoints_.insert(std::pair<int,tugboat_control::Waypoint>(id, scaledpt));
  }
```

```
}

bool check(tugboat_control::WaypointAvailable::Request &req,
           tugboat_control::WaypointAvailable::Response &res)
{
  for (std::map<int, tugboat_control::Waypoint>::iterator i = waypoints_.begin();
                                                          i != waypoints_.end();
                                                          ++i)
  {
    if (i->first == req.tugID)
    {
      continue;
    }
    if (sqrt(pow(req.waypoint.x - i->second.x, 2) + pow(req.waypoint.y - i->second.y,2))
        < CLOSE_POINTS_RADIUS*SCALE)
    {
      res.ans.data = false;
      return true;
    }
  }
  res.ans.data = true;
  return true;
}


int main(int argc, char **argv)
{
  ros::init(argc, argv, "current_waypoint_service");
  ros::NodeHandle node;
```

```
  ros::Subscriber sub = node.subscribe("waypoint", 20, set_waypoint);
  ros::ServiceServer service = node.advertiseService("isWaypointAvailable", check);

  ros::spin();
  return 0;
}
```

master_node.cpp

```
#include "geometry/tug_boat.hpp"
#include "geometry/tug_environment.hpp"
#include "search/tug_shortest_path.hpp"
#include "tug_communicator.hpp"
#include "tug_constants.hpp"
#include "tugboat_control/BoatPose.h"
#include "tugboat_control/Waypoint.h"

#include <ros/package.h>
#include <ros/ros.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "master_node");
  ros::NodeHandle node;

  if (argc < 2){ROS_ERROR("Need environment filename as argument"); return -1;};

  std::vector<std::string> all_args(argv, argv + argc);
  all_args.assign(argv + 1, argv + argc);
  std::string filename = all_args.at(0);

  try
  {
    Tug::Environment environment_tug = Tug::Environment(filename, 1, 0.01);
  }
  catch(...)
  {
```

```
  ROS_ERROR("File does not exist"); return -1;
}

Tug::Environment environment_tug = Tug::Environment(filename, 1, 0.01);
//environment_tug.add_constant_safety_margin(0.2*SCALE);

Tug::Communicator communicator(environment_tug, SCALE, 0.3);

ros::Subscriber sub_startup = node.subscribe("startup",
                                             20,
                                             &Tug::Communicator::callback_new_tug,
                                             &communicator);
ros::Subscriber sub_goal = node.subscribe("waypointRequest",
                                          20,
                                          &Tug::Communicator::callback_waypoint,
                                          &communicator);

ros::Subscriber sub_tug_locations = node.subscribe("pose",
                                                   20,
                                                   &Tug::Communicator::callback_boat_pose,
                                                   &communicator);
ros::Subscriber sub_available_tugs =
                      node.subscribe("waypTugs",
                      10,
                      &Tug::Communicator::callback_available_tugs,
                      &communicator);
ros::Subscriber sub_clearWaypoint =
                       node.subscribe("clearSingleWaypoint",
                       20,
                       &Tug::Communicator::remove_end_point_from_planner, &communicator);
```

```
28    ros::spin();

      return 0;
    }
```

tug_communicator.cpp

```cpp
#include "tug_communicator.hpp"

namespace Tug
{
  Communicator::Communicator( Environment &environment, double scale,
                              double accept_waypoint_radius)
  {
    scale_ = scale;
    accept_waypoint_radius_ = accept_waypoint_radius;
    environment_tug_ = environment;
    tug_arrived_pub = node_.advertise<tugboat_control::ClearWaypoint>("clearWaypoint", 20);
    path_pub = node_.advertise<tugboat_control::Path>("paths", 20);
    assigner_ptr_ = std::make_shared<Assign_paths>(environment);
  }

  void Communicator::print_path(Tug::Polyline path)
  {
    std::stringstream path_string;
    for (int i = 0; i < path.size(); ++i)
    {
      path_string << path[i];
      if (i < path.size()-1)
      {
        path_string << " - ";
      }
    }
    ROS_INFO("Shortest path: %s", path_string.str().c_str());
  }
```

```
bool Communicator::tug_is_under_my_control(int id)
{
  for (int i = 0; i < tugs_under_my_control_.size(); ++i)
  {
    if (id==tugs_under_my_control_[i])
    {
      return true;
    }
  }
  return false;
}

void Communicator::polyline_to_path_msg(const Tug::Polyline &path,
                                        int tug_id,
                                        int order_id,
                                        tugboat_control::Path &path_msg)
{
  path_msg.orderID = order_id;
  path_msg.tugID = tug_id;

  for (int i = 0; i < path.size(); ++i)
  {
    tugboat_control::Waypoint wp;
    wp.x = path[i].x();
    wp.y = path[i].y();
    wp.v = TUG_SPEED;
    wp.ID = tug_id;
    path_msg.data.push_back(wp);
  }
}
```

```
void Communicator::replan_route_for_one_boat(int order_id, const Tug::Point &newGoal)
{
  try
  {
    int tug_id = msg_and_tug_.at(order_id);

    if (!tug_is_under_my_control(tug_id))
    {
      return;
    }
    Tug::Point start = tugs_.at(tug_id).get_position();
    Tug::Point finish = newGoal;
    Tug::Polyline spath;
    Tug::Shortest_path sp_node(environment_tug_, start, finish, spath);

    tugboat_control::Path path_msg;
    polyline_to_path_msg(spath, tug_id, order_id, path_msg);

    if (spath.size() > 0)
    {
      path_pub.publish(path_msg);
    }
  }
  catch(const std::out_of_range &oor)
  {
    ROS_WARN("out of range in function replan_route_for_one_boat");
  }
}
```

```
void Communicator::replan()
{
  ROS_INFO("replan");


  std::vector<Tug::Boat> tugs_to_plan_for;
  for (int i = 0; i < tugs_under_my_control_.size(); ++i)
  {
    try
    {
      tugs_to_plan_for.push_back(tugs_.at(tugs_under_my_control_[i]));
    }
    catch(const std::out_of_range &err){}
  }

  if (tugs_to_plan_for.size() == 0){return;}

  assigner_ptr_->assign_on_combined_shortest_path(tugs_to_plan_for,
                                                  end_points_,
                                                  environment_tug_);
  for (int i = 0; i < tugs_to_plan_for.size(); ++i)
  {
    Polyline path = tugs_to_plan_for[i].get_path();

    if (path.size() > 0)
    {
      tugboat_control::Path path_msg;
      polyline_to_path_msg(path, tugs_to_plan_for[i].id(),
                           find_order_id(path.back()), path_msg);
```

```
      try
      {
        msg_and_tug_.at(path_msg.orderID) = path_msg.tugID;
      }
      catch(const std::out_of_range &oor)
      {
        msg_and_tug_.insert(std::pair<int,int>(path_msg.orderID,path_msg.tugID));
      }

      path_pub.publish(path_msg);
    }
    else
    {
      ROS_WARN("Could not find possible route");
    }
  }
}


void Communicator::callback_waypoint(const tugboat_control::Waypoint::ConstPtr& msg)
{
  if (tugs_under_my_control_.size() == 0 || tugs_.size() == 0)
  {
    ROS_WARN("No tugs under my control");
    return;
  }
  for (std::map<int, Boat>::iterator i = tugs_.begin(); i != tugs_.end(); ++i)
  {
    Point pt(i->second.get_position());
```

```
  if (pt.x()==-1 && pt.y()==-1)
  {
    ROS_WARN("Positions of tugs not set");
    return;
  }
}

Tug::Point pt(msg->x*scale_, msg->y*scale_, environment_tug_);
int order_id = msg->ID;

try
{
  if (end_points_.at(order_id).x() == pt.x() && end_points_.at(order_id).y() == pt.y())
  {
    return;
  }
  if (pow(end_points_.at(order_id).x() - pt.x(), 2) +
      pow(end_points_.at(order_id).y() - pt.y(), 2) < 0.05*scale_)
  {
    end_points_.at(order_id) = pt;
    replan_route_for_one_boat(order_id, pt);
  }
}
catch(const std::out_of_range &oor)
{
  for (std::map<int, Boat>::iterator i = tugs_.begin(); i != tugs_.end(); ++i)
  {
    Point pos = i->second.get_position();
    if (sqrt(pow(pos.x() - pt.x(), 2) + pow(pos.y() - pt.y(), 2)) <
                                      accept_waypoint_radius_*scale_)
```

```
        {
          ROS_INFO("A tug is already at goal");
          return;
        }
      }
      //If waypoint is new
      end_points_.insert(std::pair<int, Tug::Point>(order_id, pt));
      replan();

    }
  }

  void Communicator::remove_tug_from_control(int tug_id)
  {
    for (std::vector<int>::iterator i = tugs_under_my_control_.begin();
                                    i != tugs_under_my_control_.end();
                                    ++i)
    {
      if (*i == tug_id)
      {
        try
        {
          tugs_under_my_control_.erase(i);
          return;
        }
        catch(...){return;}
      }
    }
  }
```

```
int Communicator::find_order_id(const Tug::Point &pt)
{
  for (std::map<int,Tug::Point>::iterator i = end_points_.begin();
                                          i != end_points_.end();
                                          ++i)
    {
      if (i->second == pt)
        {
          return i->first;
        }
    }
  return -1;
}

void Communicator::remove_end_point_from_planner(
                    const tugboat_control::ClearWaypoint::ConstPtr &msg)
{
  try
    {
      end_points_.erase(msg->orderID);
      order_ready_to_publish.push_back(*msg);
      remove_tug_from_control(msg->tugID);

      if (end_points_.size() == 0 || order_ready_to_publish.size() > 1)
      {
        for (int i = 0; i < order_ready_to_publish.size(); ++i)
        {
          tug_arrived_pub.publish(order_ready_to_publish[i]);
        }
```

```
        order_ready_to_publish.clear();
      }
    }
    catch(...)
    {
      ROS_WARN("Could not erase end point with order id %d", msg->orderID);
    }
  }

  void Communicator::callback_boat_pose(const tugboat_control::BoatPose::ConstPtr& msg)
  {
    if (tugs_.size() == 0)
    {
      return;
    }

    int id = msg->ID;

    Tug::Point pt(msg->x*scale_, msg->y*scale_, environment_tug_);
    try
    {
      tugs_.at(id).update_position(pt);
    }
    catch(const std::out_of_range &oor)
    {
      ROS_WARN("Tug %d has not started yet", id);
      return;
    }
  }
```

```
void Communicator::callback_available_tugs(
                        const std_msgs::UInt8MultiArray::ConstPtr &msg)
{
  tugs_under_my_control_.clear();
  for (int i = 0; i < msg->data.size(); ++i)
  {
    tugs_under_my_control_.push_back(msg->data[i]);
  }
}

void Communicator::add_new_tug(int id)
{
  Boat tug;
  tug.set_id(id);
  tugs_.insert(std::pair<int, Tug::Boat>(id, std::move(tug)));
  ROS_INFO("Added tug %d", id);
}

bool Communicator::tug_id_already_in_system(int id)
{
  try
  {
    tugs_.at(id);
    return true;
  }
  catch(const std::out_of_range &oor)
  {
    return false;
  }
}
```

```
void Communicator::callback_new_tug(const std_msgs::UInt8::ConstPtr &msg)
{
    if (tug_id_already_in_system(msg->data))
    {
        return;
    }
    //startup message
    add_new_tug(msg->data);
}

}
```

tug_waypoint_publisher.cpp

```
#include "tug_waypoint_publisher.hpp"
#include <stdexcept>
namespace Tug
{

  Waypoint_publisher::Waypoint_publisher(int id, double scale)
  {
    id_ = id;
    scale_ = scale;
    acceptance_radius = 0.05;
    wayp_pub = node_.advertise<tugboat_control::Waypoint>("waypoint", 20);
    arrival_pub = node_.advertise<tugboat_control::ClearWaypoint>
                                ("clearSingleWaypoint", 20);
    client_is_avail = node_.serviceClient<tugboat_control::WaypointAvailable>
                                        ("isWaypointAvailable");
    client_avoid_ship = node_.serviceClient<tugboat_control::AvoidShipCollision>
                                          ("avoidShipCollision");
  }

  void Waypoint_publisher::update_position(const tugboat_control::BoatPose::ConstPtr& msg)
  {
    tugboat_control::BoatPose pose;
    pose.x = msg->x * scale_;
    pose.y = msg->y * scale_;
    pose.o = msg->o;
    newest_pose_ = pose;

    tugboat_control::Waypoint current_wp;
```

```
  if (current_waypoint_index_ >= path_.size())
  {
    return;
  }

  current_wp = path_[current_waypoint_index_];
  bool new_waypoint_set = false;
  //Check if position is within radius of current waypoint
  if(sqrt(pow(newest_pose_.x - current_wp.x, 2) +
          pow(newest_pose_.y - current_wp.y, 2)) < acceptance_radius*scale_)
  {
    new_waypoint_set = go_to_next_waypoint();
  }
}

void Waypoint_publisher::set_path(const tugboat_control::Path::ConstPtr &msg)
{
  if (msg->tugID == id_)
  {
    path_.clear();
    path_ = msg->data;
    current_waypoint_index_= 0;
    order_id_ = msg->orderID;
  }
  if (path_.size() == 2)
  {
    if (!SIMULATION)
    {
      call_path_around_ship_service(path_[0], path_[1], path_);
      current_waypoint_index_ = 0;
```

```
    }
  }

}

void Waypoint_publisher::call_path_around_ship_service(
                                        const tugboat_control::Waypoint &start,
                                        const tugboat_control::Waypoint &finish,
                                        std::vector<tugboat_control::Waypoint> &result)
{
  tugboat_control::AvoidShipCollision srv;
  srv.request.from = start;
  srv.request.to = finish;

  if(client_avoid_ship.call(srv))
  {
    result.clear();
    result = srv.response.path;
  }
  else
  {
    ROS_WARN("Service avoid_ship failed to reply");
  }
}

bool Waypoint_publisher::is_waypoint_available(const tugboat_control::Waypoint &pt)
{
  //Waypoint available service not tested in real test setup
  if (!SIMULATION)
  {
```

```
    return true;
  }
  tugboat_control::WaypointAvailable srv;
  srv.request.waypoint = pt;
  srv.request.tugID = id_;

  if (client_is_avail.call(srv))
  {
    if (srv.response.ans.data == true)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  ROS_WARN("Service WaypointAvailable failed to reply");
  return false;
}

void Waypoint_publisher::publish_current_waypoint()
{
    tugboat_control::Waypoint wp;
    wp.x = path_[current_waypoint_index_].x/SCALE_OUT;
    wp.y = path_[current_waypoint_index_].y/SCALE_OUT;
    wp.ID = id_;
    wp.v = TUG_SPEED;
    wayp_pub.publish(wp);
}
```

```
bool Waypoint_publisher::go_to_next_waypoint()
{
  ++current_waypoint_index_;

  if (current_waypoint_index_ >= path_.size())
  {
    ROS_WARN("Tug %d has arrived", id_);
    --current_waypoint_index_;
    tugboat_control::ClearWaypoint clear; clear.orderID = order_id_; clear.tugID = id_;
    arrival_pub.publish(clear);
    is_already_on_hold = false;
    return false;
  }
  //heading towards goal
  else if (!SIMULATION && current_waypoint_index_ == path_.size() - 1)
  {
    ROS_WARN("Trying to find route around ship if it is on the wrong side");

    call_path_around_ship_service(path_[current_waypoint_index_-1],
                                  path_[path_.size() - 1], path_);
    current_waypoint_index_ = 1;
    publish_current_waypoint();
    return true;
  }
  else
  {
    ROS_WARN("Tug %d arrived at a waypoint", id_);

    if (is_waypoint_available(path_[current_waypoint_index_]))
```

```
        {
          publish_current_waypoint();
          is_already_on_hold = false;
          return true;
        }
        else
        {
          --current_waypoint_index_;
          if (!is_already_on_hold)
          {
            ROS_WARN("Tug %d on hold", id_);
            is_already_on_hold = true;
          }
          return false;
        }
      }
    }
  }
```

C.1.3 test

munkres_speed_test.cpp

```
#include "coordination/tug_assign_paths.hpp"
#include "geometry/tug_boat.hpp"
#include "geometry/tug_environment.hpp"
#include "search/tug_shortest_path.hpp"

#include <ros/package.h>
#include <ros/ros.h>

#include <chrono>

int main(int argc, char **argv)
{
  ClipperLib::Paths solution;
  double epsilon = 0.001;

  Tug::Environment tug_env("ex1tug.txt",
                           1.0,
                           epsilon);

  std::vector<Tug::Point> start_points;
  start_points.push_back(Tug::Point(60, 60, tug_env));
  start_points.push_back(Tug::Point(40, 40, tug_env));
  start_points.push_back(Tug::Point(250, 200, tug_env));
  start_points.push_back(Tug::Point(10, 10, tug_env));
  start_points.push_back(Tug::Point(50, 50, tug_env));
  start_points.push_back(Tug::Point(56, 30, tug_env));
```

```
start_points.push_back(Tug::Point(15, 15, tug_env));
start_points.push_back(Tug::Point(280, 200, tug_env));
start_points.push_back(Tug::Point(330 ,320, tug_env));
start_points.push_back(Tug::Point(5, 318, tug_env));

std::vector<Tug::Point> finish_points;
finish_points.push_back(Tug::Point(318, 320, tug_env));
finish_points.push_back(Tug::Point(40, 240, tug_env));
finish_points.push_back(Tug::Point(349, 1, tug_env));
finish_points.push_back(Tug::Point(20, 20, tug_env));
finish_points.push_back(Tug::Point(308, 322, tug_env));
finish_points.push_back(Tug::Point(325 , 340, tug_env));
finish_points.push_back(Tug::Point(250, 150, tug_env));
finish_points.push_back(Tug::Point(310, 324, tug_env));
finish_points.push_back(Tug::Point(170, 200, tug_env));
finish_points.push_back(Tug::Point(50, 290, tug_env));

std::vector<Tug::Boat> tugs;

for (int i = 0; i < start_points.size(); ++i)
{
  Tug::Boat tug(7.0, start_points[i], &tug_env);
  tug.set_id(i+1);
  tug.set_top_speed(1);
  tugs.push_back(tug);
}

Tug::Assign_paths assigner;
auto t1 = std::chrono::high_resolution_clock::now();
bool ok = assigner.assign(tugs, finish_points, tug_env);
```

```
  auto t2 = std::chrono::high_resolution_clock::now();

  if (!ok) return -1;

  ROS_INFO("test for %lu tugs function took %ld milliseconds", tugs.size(),
   std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1).count() );

  std::vector<Tug::Polyline> shortest_paths;
  for (int i = 0; i < tugs.size(); ++i)
  {
    shortest_paths.push_back(tugs[i].get_path());
  }
  tug_env.save_environment_as_svg("speed_test.svg", shortest_paths);

  return 0;
}
```

test_shortest_path.cpp

```cpp
#include "geometry/tug_boat.hpp"
#include "geometry/tug_environment.hpp"
#include "search/tug_shortest_path.hpp"

#include <ros/package.h>
#include <ros/ros.h>

int main(int argc, char **argv)
{
  ClipperLib::Paths solution;
  double epsilon = 0.01;

  Tug::Environment tug_env(
    "env100.txt",
     1.0,
     epsilon);
  tug_env.add_constant_safety_margin(5);
  Tug::Point start(10, 570, tug_env);
  Tug::Point finish(450, 3, tug_env);

  Tug::Shortest_path sp_node(tug_env);
  Tug::Polyline shortest_path;
  bool ok = sp_node.calculate_shortest_path(start, finish, shortest_path, tug_env);

  tug_env.save_environment_as_svg("dense_env_w_safety.svg", shortest_path);


  return 0;
```

test_speed.cpp

```
#include "geometry/tug_boat.hpp"
#include "geometry/tug_environment.hpp"
#include "search/tug_shortest_path.hpp"

#include <chrono>
#include <ros/package.h>
#include <ros/ros.h>


    int x_min_; int y_min_;
    int x_max_; int y_max_;
  void path_to_hole(const ClipperLib::Path &path, VisiLibity::Polygon &hole)
  {
    for(int i = 0; i < path.size(); i++)
    {
      int x = (int)path[i].X;
      int y = (int)path[i].Y;
      hole.push_back(VisiLibity::Point(x,y));
    }
  }
  void find_max_and_min_in_path(const ClipperLib::Path &path, char coordinate, int &max_val, int &min_val)
  {
    if (!(coordinate == 'X' || coordinate == 'Y'))
    {
      return;
    }
    max_val = std::numeric_limits<int>::min();
    min_val = std::numeric_limits<int>::max();
```

```
    int current;
    for (int i = 0; i < path.size(); ++i)
    {
      if (coordinate == 'X')
      {
        current = path[i].X;
      }
      else
      {
        current = path[i].Y;
      }

      if (current < min_val)
      {
        min_val = current;
      }
      else if (current > max_val)
      {
        max_val = current;
      }
    }
  }

  void set_outer_boundary(const ClipperLib::Path &outer_boundary, VisiLibity::Environment &environment)
  {
    find_max_and_min_in_path(outer_boundary, 'X', x_max_, x_min_);
    find_max_and_min_in_path(outer_boundary, 'Y', y_max_, y_min_);

    VisiLibity::Polygon outer_boundary_polygon;
```

```
    path_to_hole(outer_boundary, outer_boundary_polygon);
    environment.set_outer_boundary(outer_boundary_polygon);
  }

void convert_to_visilibity_environment(const ClipperLib::Paths &paths, VisiLibity::Environment &environment)
  {
    if (paths.size() == 0)
    {
      return;
    }

    find_max_and_min_in_path(paths[0], 'X', x_max_, x_min_);
    find_max_and_min_in_path(paths[0], 'Y', y_max_, y_min_);

    int x_min_cur, x_max_cur, y_min_cur, y_max_cur;

    for (int i = 1; i < paths.size(); ++i)
    {
      find_max_and_min_in_path(paths[i], 'X', x_max_cur, x_min_cur);
      if (x_max_cur >= x_max_)
      {
        x_max_ = x_max_cur +1 ;
      }
      if (x_min_cur <= x_min_)
      {
        x_min_ = x_min_cur -1 ;
      }
      find_max_and_min_in_path(paths[i], 'Y', y_max_cur, y_min_cur);
      if (y_max_cur >= y_max_)
      {
```

```
      y_max_ = y_max_cur + 1 ;
    }
    if (y_min_cur <= y_min_)
    {
      y_min_ = y_min_cur -1 ;
    }
  }
  ClipperLib::Path path_temp;
  path_temp.push_back(ClipperLib::IntPoint(x_min_, y_min_));
  path_temp.push_back(ClipperLib::IntPoint(x_max_, y_min_));
  path_temp.push_back(ClipperLib::IntPoint(x_max_, y_max_));
  path_temp.push_back(ClipperLib::IntPoint(x_min_, y_max_));

  set_outer_boundary(path_temp, environment);

  for (int i = 1; i < paths.size(); ++i)
  {
    VisiLibity::Polygon hole;
    path_to_hole(paths[i], hole);
    environment.add_hole(hole);
  }
}

bool load_from_file(ClipperLib::Paths &ppg, const std::string& filename)
{
  //file format assumes:
  //  1. path coordinates (x,y) are comma separated (+/- spaces) and
  //  each coordinate is on a separate line
  //  2. each path is separated by one or more blank lines
  ppg.clear();
```

```
    std::ifstream ifs(filename);
    if (!ifs) return false;
    std::string line;
    ClipperLib::Path pg;
    while (std::getline(ifs, line))
    {
      std::stringstream ss(line);
      double X = 0.0, Y = 0.0;
      if (!(ss >> X))
      {
        //ie blank lines => flag start of next polygon
        if (pg.size() > 0) ppg.push_back(pg);
        pg.clear();
        continue;
      }
      char c = ss.peek();
      while (c == ' ') {ss.read(&c, 1); c = ss.peek();} //gobble spaces before comma
      if (c == ',') {ss.read(&c, 1); c = ss.peek();} //gobble comma
      while (c == ' ') {ss.read(&c, 1); c = ss.peek();} //gobble spaces after comma
      if (!(ss >> Y)) break; //oops!
      pg.push_back(ClipperLib::IntPoint((ClipperLib::cInt)(X),(ClipperLib::cInt)(Y)));
    }
    if (pg.size() > 0) ppg.push_back(pg);
    ifs.close();

    return true;
  }


void run_method1(std::string &filename, const Tug::Point &start,
```

```
  const Tug::Point &goal, int no_nodes, double epsilon)
{
  ClipperLib::Paths paths;
  VisiLibity::Environment env;

  load_from_file(paths, filename);
  convert_to_visilibity_environment(paths, env);
  VisiLibity::Point start_vis(start.x(), start.y());
  VisiLibity::Point goal_vis(goal.x(), goal.y());

  auto begin = std::chrono::high_resolution_clock::now();
    VisiLibity::Polyline sol = env.shortest_path(start_vis, goal_vis, epsilon);
  auto end = std::chrono::high_resolution_clock::now();
  auto ms = std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count();
  std::cout << "Method 1, " << no_nodes << " nodes: " << ms << " microseconds. "<< std::endl;
}

Tug::Polyline run_method2(Tug::Environment &env, const Tug::Point &start,
  const Tug::Point &goal, int no_nodes, double epsilon)
{
  Tug::Polyline shortest_path;
  auto begin = std::chrono::high_resolution_clock::now();
    Tug::Shortest_path shortest_path_node(env, start, goal, shortest_path);
  auto end = std::chrono::high_resolution_clock::now();
  auto ms = std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count();
  std::cout << "Method 2, " << no_nodes << " nodes: " << ms << " microseconds. "<< std::endl;
  return shortest_path;
}

void run_method3(Tug::Environment &env, const Tug::Point &start, const Tug::Point &goal,
```

```
 int no_nodes, double epsilon)
{
  Tug::Polyline shortest_path;
  Tug::Shortest_path shortest_path_node(env);

  auto begin = std::chrono::high_resolution_clock::now();
  shortest_path_node.calculate_shortest_path(start, goal, shortest_path, env);
  auto end = std::chrono::high_resolution_clock::now();
  auto ms = std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count();
  std::cout << "Method 3, " << no_nodes << " nodes: " << ms << " microseconds. "<< std::endl;
}

int main(int argc, char **argv)
{
  double epsilon = 0.01;

  std::string filename10 = "env10.txt";
  Tug::Environment env10 = Tug::Environment(filename10, 1, epsilon);

  Tug::Point start10(145, 492, env10);
  Tug::Point goal10(684, 97, env10);
  run_method1(filename10,start10,goal10,10,epsilon);
  Tug::Polyline sp10 =
  run_method2(env10,start10,goal10,10,epsilon);
  run_method3(env10,start10,goal10,10,epsilon);

  std::string filename20 = "env20.txt";
  Tug::Environment env20 = Tug::Environment(filename20, 1, epsilon);

  Tug::Point start20(145, 492, env20);
```

```
Tug::Point goal20(684, 97, env20);

run_method1(filename20,start20,goal20,20,epsilon);
Tug::Polyline sp20 =
run_method2(env20,start20,goal20,20,epsilon);
run_method3(env20,start20,goal20,20,epsilon);

std::string filename50 = "env50.txt";
Tug::Environment env50 = Tug::Environment(filename50, 1, epsilon);

Tug::Point start50(10, 570, env50);
Tug::Point goal50(868, 3, env50);

run_method1(filename50,start50,goal50,50,epsilon);
Tug::Polyline sp50 =
run_method2(env50,start50,goal50,50,epsilon);
run_method3(env50,start50,goal50,50,epsilon);

std::string filename100 = "env100.txt";
Tug::Environment env100 = Tug::Environment(filename100, 1, epsilon);

Tug::Point start100(10, 570, env100);
Tug::Point goal100(868, 3, env100);

run_method1(filename100,start100,goal100,100,epsilon);
Tug::Polyline sp100 =
run_method2(env100,start100,goal100,100,epsilon);
run_method3(env100,start100,goal100,100,epsilon);
```

```
    std::string filename200 = "env200.txt";
    Tug::Environment env200 = Tug::Environment(filename200, 1, epsilon);

    Tug::Point start200(10, 570, env200);
    Tug::Point goal200(868, 3, env200);

    run_method1(filename200,start200,goal200,200,epsilon);
    Tug::Polyline sp200 =
    run_method2(env200,start200,goal200,200,epsilon);
    run_method3(env200,start200,goal200,200,epsilon);


    env10.save_environment_as_svg("env10.svg", sp10);
    env20.save_environment_as_svg("env20.svg", sp20);
    env50.save_environment_as_svg("env50.svg", sp50);
    env100.save_environment_as_svg("env100.svg", sp100);
    env200.save_environment_as_svg("env200.svg", sp200);

}
```

C.2 Geometry

Contents

- SVG_builder.hpp
- tug_boat.hpp
- tug_environment.hpp
- tug_point.hpp
- tug_polyline.hpp
- tug_ship.hpp
- SVG_builder.cpp
- tug_boat.cpp
- tug_environment.cpp
- tug_point.cpp
- test.cpp

C.2.1 include

SVG_builder.hpp

```
#ifndef SVG_BUILDER_HPP
#define SVG_BUILDER_HPP

#include "external/clipper.hpp"
#include "tug_polyline.hpp"

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

class SVGBuilder
{
  std::string ColorToHtml(unsigned clr);
  //------------------------------------------------------------------------

  float GetAlphaAsFrac(unsigned clr);

  class StyleInfo
```

```
{
  public:
  ClipperLib::PolyFillType pft;
  unsigned brushClr;
  unsigned penClr;
  double penWidth;
  bool showCoords;

  StyleInfo()
  {
    pft = ClipperLib::pftNonZero;
    brushClr = 0xFFFFFFCC;
    penClr = 0xFF000000;
    penWidth = 2.0;
    showCoords = false;
  }
};

class PolyInfo
{
  public:
    ClipperLib::Paths paths;
    StyleInfo si;

    PolyInfo(ClipperLib::Paths paths, StyleInfo style)
    {
        this->paths = paths;
        this->si = style;
    }
};
```

```
  typedef std::vector<PolyInfo> PolyInfoList;

private:
  PolyInfoList polyInfos;
  std::vector<Tug::Polyline> polylines;


public:
  StyleInfo style;

  void AddPaths(ClipperLib::Paths& poly);
  void AddPolyline(const Tug::Polyline& polyline);

  bool SaveToFile(const std::string& filename, double scale, int margin);
}; //SVGBuilder

#endif
```

tug_boat.hpp

```
#ifndef TUG_BOAT_H
#define TUG_BOAT_H

#include "tug_environment.hpp"
#include "tug_point.hpp"
#include "tug_polyline.hpp"

#include <memory>

namespace Tug
{
  class Boat
  {
  public:
    Point get_position();
    void update_position(Point &position);
    void set_id(int id){id_ = id;};
    int id() const {return id_;};
    void set_path(const Polyline &path);
    Polyline get_path(){return path_;};

  private:
    Point position_ = Point(-1,-1,-1);
    int id_=-1;
    Polyline path_;
  };
}
```

```
#endif //TUG_BOAT_H
```

tug_environment.hpp

```
#ifndef TUG_ENVIRONMENT_H
#define TUG_ENVIRONMENT_H

#include "external/clipper.hpp"
#include "external/visilibity.hpp"
#include "SVG_builder.hpp"
#include "tug_point.hpp"
#include "tug_polyline.hpp"

#include <map>
#include <vector>

namespace Tug
{
  class Environment
  {
    public:
      friend class Shortest_path;
      friend class All_pairs_shortest_path;

      Environment(const std::string& filename, double scale, double epsilon);
      Environment(){};

      const VisiLibity::Environment &visilibity_environment() const;
      void add_constant_safety_margin(double margin); //, ClipperLib::Paths &solution);
      bool load_from_file(ClipperLib::Paths &ppg, const std::string& filename, double scale);
      //Polyline shortest_path(const Point &start, const Point &finish); //, double epsilon);
```

```
    void save_environment_as_svg(const std::string filename);
    void save_environment_as_svg(const std::string filename, const Polyline &shortest_path);
    void save_environment_as_svg(const std::string filename, const std::vector<Polyline> &shortest_paths);

    bool has_safety_margin(){return environment_has_safety_margin;};
    unsigned n() const;
    const Point &operator () (unsigned k) const;

    std::vector<Point> points();

    void get_boundaries(int &x_min, int &x_max, int &y_min, int &y_max) const;
    bool point_is_within_outer_boundary(const Tug::Point point);
    bool point_is_on_outer_boundary(const VisiLibity::Point &point) const;

    int find_id(const VisiLibity::Point &point) const;

    std::map<int, Point>::const_iterator const_begin() const;
    std::map<int, Point>::const_iterator const_end() const;

    void print_coordinates_and_id() const;
    void mark_points_within_range(float range);

  private:
    int id_counter_ = 0;
    std::map<std::pair<double,double>,int> coordinate_to_id;
    bool environment_has_safety_margin = false;
    int x_min_, x_max_, y_min_, y_max_; //of outer boundary
    double epsilon_;

    VisiLibity::Environment visilibity_environment_;
```

```
        VisiLibity::Environment visilibity_environment_with_safety_margin_;

        //obstacles
        ClipperLib::Paths paths_;
        ClipperLib::Paths paths_with_safety_margin_;

        std::map<int, Point> points_in_environment_;
        std::map<int, Point> points_in_environment_with_safety_margin_;

        std::map<int, Point>::iterator begin();
        std::map<int, Point>::iterator end();

        void clip_against_outer_boundary(ClipperLib::Paths &paths_in, ClipperLib::Paths &paths_out);
        void update_tug_point_list(const ClipperLib::Paths &paths, std::map<int, Point> &tug_points);
        void make_visibility_graphs_for_points(std::map<int, Point> &tug_points);
        void offset_polygon(VisiLibity::Polygon &polygon, int margin);
        void convert_to_visilibity_environment(const ClipperLib::Paths &paths, VisiLibity::Environment &environment);
        void reverse_path(ClipperLib::Path &path);
        void path_to_hole(const ClipperLib::Path &path, VisiLibity::Polygon &hole);
        void find_max_and_min_in_path(const ClipperLib::Path &path, char coordinate, int &max_val, int &min_val);
        void set_outer_boundary(const ClipperLib::Path &outer_boundary, VisiLibity::Environment &environment);
        void mark_points_touching_outer_boundary();
    };
}

#endif //TUG_ENVIRONMENT_H
```

tug_point.hpp

```
#ifndef TUG_POINT_H
#define TUG_POINT_H

#include "external/clipper.hpp"
#include "external/visilibity.hpp"

#include <math.h>
#include <map>

namespace Tug
{
  class Environment;

  class Point : public VisiLibity::Point
  {
  public:
    Point(const ClipperLib::IntPoint &point, int point_id=-1);
    Point(int point_id = -1) : VisiLibity::Point(), point_id_(point_id){};
    Point(double x_temp, double y_temp, const Environment &environment, int point_id=-1);
    Point(const ClipperLib::IntPoint &point, const Environment &environment, int point_id=-1);
    Point(double x_temp, double y_temp, int point_id=-1) : VisiLibity::Point(x_temp, y_temp), point_id_(point_id){};
    Point(VisiLibity::Point &point, int point_id=-1) : VisiLibity::Point(point.x(),point.y()), point_id_(point_id) {};
    Point(const Point &obj);

    Point &operator=(const Point &other);
    bool operator==(const Point &other) const;

    int id() const {return point_id_;}
```

```
    friend std::ostream& operator<<(std::ostream &out, Point const& pt);
    bool is_visible(const Tug::Point &point) const;
    std::vector<int> visible_vertices() const {return visible_vertices_;}
    void add_shortest_path_cost(const Point &pt, double cost)
      {shortest_path_costs_.insert(std::pair<int, double>(pt.id(),cost));}
    double get_cost_to(int id) const {return shortest_path_costs_.at(id);}

    VisiLibity::Visibility_Polygon visibility_polygon() const{return visibility_polygon_;};
    void create_visibility_polygon(const Environment &environment);
    bool is_on_outer_boundary = false;
    std::vector<int> visible_vertices_;

    void add_close_point(Point *pt);

  protected:
    std::map<int, double> shortest_path_costs_;
  //  std::map<int, Tug::Point> visible_vertices_;
   // std::vector<int> visible_vertices_;
    const int point_id_;
    VisiLibity::Visibility_Polygon visibility_polygon_;
    std::vector<Point*> close_points;
  };
}

#endif //TUG_POINT_H
```

tug_polyline.hpp

```
#ifndef TUG_POLYLINE_H
#define TUG_POLYLINE_H

#include "tug_point.hpp"

namespace Tug
{
  class Polyline
  {
  public:
    Polyline(){};

    Point operator [] (unsigned i) const
    { return vertices_[i]; }

    unsigned size() const
    { return vertices_.size(); }

    Point& operator [] (unsigned i)
    { return vertices_[i]; }

    bool operator==(const Polyline &p2)
    {
      if (this->size() != p2.size())
      {
        return false;
      }
      for (int i = 0; i < this->size(); ++i)
```

```
    {
      if (this->vertices_[i] != p2[i])
      {
        return false;
      }
    }
    return true;
  }

  void clear()
  { vertices_.clear(); polyline_edited_since_last_length_calculation = true;}

  void push_back(const Point& point_temp)
  { //Point point_copy(point_temp);
    //vertices_.push_back(point_copy);
    vertices_.push_back(point_temp);
    polyline_edited_since_last_length_calculation = true;}

  void push_front( Point point)
  {reverse(); push_back(point); reverse();}

  void pop_back()
  { vertices_.pop_back(); polyline_edited_since_last_length_calculation = true;}

  Point back(){return vertices_.back();}

  void reverse()
  {
    std::vector<int> ids;
    std::vector<Point> pts;
```

```
  //Environement environment = pts.en
  for (int i = 0; i < vertices_.size(); ++i)
  {
    ids.push_back(vertices_[i].id());
    pts.push_back(vertices_[i]);
  }
  vertices_.clear();
  for (int i = pts.size()-1; i >= 0; --i)
  {
    vertices_.push_back(Point(pts[i].x(), pts[i].y(), ids[i]));
  }
//  std::reverse( std::begin(vertices_) , std::end(vertices_ ) );
}

double length()
{
  if (polyline_edited_since_last_length_calculation)
  {
    if (vertices_.size()==0 || vertices_.size()==1)
    {
      return 0;
    }
    double sum = 0;

    for (int i = 0; i < vertices_.size()-1; ++i)
    {
      sum += sqrt(pow(vertices_[i].x() - vertices_[i+1].x(), 2) + pow(vertices_[i].y() - vertices_[i+1].y(), 2));
    }
    length_ = sum;
    polyline_edited_since_last_length_calculation = false;
```

```
      return sum;
    }
    else
    {
      //It has been calculated before
      return length_;
    }
  };

  bool operator==(const Polyline &p2) const
  {
    if (this->size() != p2.size())
    {
      return false;
    }
    for (int i = 0; i < this->size(); ++i)
    {
      if (this->vertices_[i] != p2[i])
      {
        return false;
      }
    }
    return true;
  }

private:
  std::vector<Point> vertices_;
  double length_;
  bool polyline_edited_since_last_length_calculation = true;
};
```

```
}

#endif //TUG_POLYLINE_H
```

tug_ship.hpp

```
#ifndef TUG_SHIP_H
#define TUG_SHIP_H

#include "tug_point.hpp"
#include "tug_polyline.hpp"
#include "tug_environment.hpp"
//#include "tug_switching_point_decision.hpp"

namespace Tug
{
  class Ship : public Boat
  {
  public:
    Ship(double length, double width, Environment *env) : length_(length), width_(width);

  private:
    double length;
    double width;
  };
}

#endif //TUG_SHIP_H
```

C.2.2 src

SVG_builder.cpp

```
/*****************************************************************************
*                                                                            *
* Author    :  Angus Johnson                                                 *
* Version   :  6.4.0                                                         *
* Date      :  2 July 2015                                                   *
* Website   :  http://www.angusj.com                                         *
*                                                                            *
* Modified by Rebecca Cox (13 july 2017)                                     *
*                                                                            *
* License:                                                                   *
* Use, modification & distribution is subject to Boost Software License Ver 1. *
* http://www.boost.org/LICENSE_1_0.txt                                       *
*                                                                            *
* Attributions:                                                              *
* The code in this library is an extension of Bala Vatti's clipping algorithm: *
* "A generic solution to polygon clipping"                                   *
* Communications of the ACM, Vol 35, Issue 7 (July 1992) pp 56-63.           *
* http://portal.acm.org/citation.cfm?id=129906                               *
*                                                                            *
* Computer graphics and geometric modeling: implementation and algorithms    *
* By Max K. Agoston                                                          *
* Springer; 1 edition (January 4, 2005)                                      *
* http://books.google.com/books?q=vatti+clipping+agoston                     *
*                                                                            *
* See also:                                                                  *
* "Polygon Offsetting by Computing Winding Numbers"                          *
```

```
* Paper no. DETC2005-85513 pp. 565-575                                      *
* ASME 2005 International Design Engineering Technical Conferences          *
* and Computers and Information in Engineering Conference (IDETC/CIE2005)   *
* September 24-28, 2005 , Long Beach, California, USA                       *
* http://www.me.berkeley.edu/~mcmains/pubs/DAC05OffsetPolygon.pdf           *
*                                                                           *
*****************************************************************************/

/*****************************************************************************
*                                                                           *
* This is a translation of the Delphi Clipper library and the naming style  *
* used has retained a Delphi flavour.                                       *
*                                                                           *
*****************************************************************************/

#include "SVG_builder.hpp"


  const std::string svg_xml_start [] =
  {"<?xml version=\"1.0\" standalone=\"no\"?>\n"
    "<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 1.0//EN\"\n"
    "\"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd\">\n\n"
    "<svg width=\"",
    "\" height=\"",
    "\" viewBox=\"0 0 ",
    "\" version=\"1.0\" xmlns=\"http://www.w3.org/2000/svg\">\n\n"
  };
const std::string poly_end [] =
  {"\"\n style=\"fill:",
    "; fill-opacity:",
```

```
    "; fill-rule:",
    "; stroke:",
    "; stroke-opacity:",
    "; stroke-width:",
    ";\"/>\n\n"
};


std::string SVGBuilder::ColorToHtml(unsigned clr)
{
  std::stringstream ss;
  ss << '#' << std::hex << std::setfill('0') << std::setw(6) << (clr & 0xFFFFFF);
  return ss.str();
}

float SVGBuilder::GetAlphaAsFrac(unsigned clr)
{
  return ((float)(clr >> 24) / 255);
}

void SVGBuilder::AddPaths(ClipperLib::Paths& poly)
{
  if (poly.size() == 0) return;
  polyInfos.push_back(PolyInfo(poly, style));
}

void SVGBuilder::AddPolyline(const Tug::Polyline& polylinein)
{
  polylines.push_back(polylinein);
}
```

```
bool SVGBuilder::SaveToFile(const std::string& filename, double scale, int margin)
{
  //calculate the bounding rect ...
  PolyInfoList::size_type i = 0;
  ClipperLib::Paths::size_type j;
  while (i < polyInfos.size())
  {
    j = 0;
    while (j < polyInfos[i].paths.size() &&
      polyInfos[i].paths[j].size() == 0) j++;
    if (j < polyInfos[i].paths.size()) break;
    i++;
  }
  if (i == polyInfos.size()) return false;

  ClipperLib::IntRect rec;
  rec.left = polyInfos[i].paths[j][0].X;
  rec.right = rec.left;
  rec.top = polyInfos[i].paths[j][0].Y;
  rec.bottom = rec.top;
  for ( ; i < polyInfos.size(); ++i)
    for (ClipperLib::Paths::size_type j = 0; j < polyInfos[i].paths.size(); ++j)
      for (ClipperLib::Path::size_type k = 0; k < polyInfos[i].paths[j].size(); ++k)
      {
        ClipperLib::IntPoint ip = polyInfos[i].paths[j][k];
        if (ip.X < rec.left) rec.left = ip.X;
        else if (ip.X > rec.right) rec.right = ip.X;
        if (ip.Y < rec.top) rec.top = ip.Y;
        else if (ip.Y > rec.bottom) rec.bottom = ip.Y;
```

```
        }

    if (scale == 0) scale = 1.0;
    if (margin < 0) margin = 0;
    rec.left = (ClipperLib::cInt)((double)rec.left * scale);
    rec.top = (ClipperLib::cInt)((double)rec.top * scale);
    rec.right = (ClipperLib::cInt)((double)rec.right * scale);
    rec.bottom = (ClipperLib::cInt)((double)rec.bottom * scale);
    ClipperLib::cInt offsetX = -rec.left + margin;
    ClipperLib::cInt offsetY = -rec.top + margin;

    std::ofstream file;
    file.open(filename);
    if (!file.is_open()) return false;
    file.setf(std::ios::fixed);
    file.precision(0);
    file << svg_xml_start[0] <<
      ((rec.right - rec.left) + margin*2) << "px" << svg_xml_start[1] <<
      ((rec.bottom - rec.top) + margin*2) << "px" << svg_xml_start[2] <<
      ((rec.right - rec.left) + margin*2) << " " <<
      ((rec.bottom - rec.top) + margin*2) << svg_xml_start[3];
    setlocale(LC_NUMERIC, "C");
    file.precision(2);

  for (PolyInfoList::size_type i = 0; i < polyInfos.size(); ++i)
  {
      file << " <path d=\"";
    for (ClipperLib::Paths::size_type j = 0; j < polyInfos[i].paths.size(); ++j)
      {
        if (polyInfos[i].paths[j].size() < 3) continue;
```

```
      file << " M " << ((double)polyInfos[i].paths[j][0].X * scale + offsetX) <<
        " " << ((double)polyInfos[i].paths[j][0].Y * scale + offsetY);
      for (ClipperLib::Path::size_type k = 1; k < polyInfos[i].paths[j].size(); ++k)
      {
        ClipperLib::IntPoint ip = polyInfos[i].paths[j][k];
        double x = (double)ip.X * scale;
        double y = (double)ip.Y * scale;
        file << " L " << (x + offsetX) << " " << (y + offsetY);
      }
      file << " z";
  }
    file << poly_end[0] << ColorToHtml(polyInfos[i].si.brushClr) <<
  poly_end[1] << GetAlphaAsFrac(polyInfos[i].si.brushClr) <<
      poly_end[2] <<
      (polyInfos[i].si.pft == ClipperLib::pftEvenOdd ? "evenodd" : "nonzero") <<
      poly_end[3] << ColorToHtml(polyInfos[i].si.penClr) <<
  poly_end[4] << GetAlphaAsFrac(polyInfos[i].si.penClr) <<
      poly_end[5] << polyInfos[i].si.penWidth << poly_end[6];

      if (polyInfos[i].si.showCoords)
      {
    file << "<g font-family=\"Verdana\" font-size=\"11\" fill=\"black\">\n\n";
    for (ClipperLib::Paths::size_type j = 0; j < polyInfos[i].paths.size(); ++j)
    {
      if (polyInfos[i].paths[j].size() < 3) continue;
      for (ClipperLib::Path::size_type k = 0; k < polyInfos[i].paths[j].size(); ++k)
      {
        ClipperLib::IntPoint ip = polyInfos[i].paths[j][k];
        file << "<text x=\"" << (int)(ip.X * scale + offsetX) <<
        "\" y=\"" << (int)(ip.Y * scale + offsetY) << "\">" <<
```

```
          ip.X << "," << ip.Y << "</text>\n";
          file << "\n";
        }
      }
      file << "</g>\n";
        }
    }

    for (int i = 0; i < polylines.size(); i++)
    {
      if (polylines[i].size() > 0)
      {
        file << "<polyline points=\"";
        for (int j = 0; j < polylines[i].size(); ++j)
        {
          file << (polylines[i][j].x()*scale + offsetX) << "," <<
          (polylines[i][j].y()*scale + offsetY) << " ";
        }
        file << "\"\n";
        file << "style=\"fill:none;stroke:black;stroke-width:3\" />\n\n";
      }
    }
    file << "</svg>\n";
    file.close();
    setlocale(LC_NUMERIC, "");
    return true;
  }
 //SVGBuilder
```

tug_boat.cpp

```
#include "tug_boat.hpp"

#include <ros/ros.h>
#include <stdexcept>

namespace Tug
{
  Point Boat::get_position()
  {
    return position_;
  }

  void Boat::update_position(Point &position)
  {
    position_ = position;
  }

  void Boat::set_path(const Polyline &path)
  {
    path_.clear();
    path_ = path;
  }
}
```

tug_environment.cpp

```cpp
#include "tug_environment.hpp"

#include <limits>
#include <sstream>


namespace Tug
{
  Environment::Environment(const std::string& filename, double scale, double epsilon)
  {
    epsilon_ = epsilon;
    ClipperLib::Paths temp_paths;
    bool ok = load_from_file(temp_paths, filename, scale);
    clip_against_outer_boundary(temp_paths, paths_);

    if(!ok)
    {
      std::cout << "Could not read environment file" << std::endl;
    }
    convert_to_visilibity_environment(paths_, visilibity_environment_);

    if (!visilibity_environment_.is_valid(epsilon))
    {
      std::cout << "Environment is not valid" << std::endl;
    }

    update_tug_point_list(paths_, points_in_environment_);
  }
```

```
std::map<int, Point>::iterator Environment::begin()
{
  if (environment_has_safety_margin)
  {
    return points_in_environment_with_safety_margin_.begin();
  }
  else
  {
    return points_in_environment_.begin();
  }
}

std::map<int, Point>::iterator Environment::end()
{
  if (environment_has_safety_margin)
  {
    return points_in_environment_with_safety_margin_.end();
  }
  else
  {
    return points_in_environment_.end();
  }
}

std::map<int, Point>::const_iterator Environment::const_begin() const
{
  if (environment_has_safety_margin)
  {
    return points_in_environment_with_safety_margin_.begin();
```

```
  }
  else
  {
    return points_in_environment_.begin();
  }
}

std::map<int, Point>::const_iterator Environment::const_end() const
{
  if (environment_has_safety_margin)
  {
    return points_in_environment_with_safety_margin_.end();
  }
  else
  {
    return points_in_environment_.end();
  }
}

std::vector<Point> Environment::points()
{
  std::vector<Point> dummy_points;
  for (std::map<int,Point>::iterator pt = begin(); pt != end(); ++pt)
  {
    dummy_points.push_back(pt->second);
  }
  return dummy_points;
}

const Point &Environment::operator() (unsigned k) const
```

```
{
  if (environment_has_safety_margin)
  {
    return points_in_environment_with_safety_margin_.at(k);
  }
  else
  {
    return points_in_environment_.at(k);
  }
}

unsigned Environment::n() const
{
  if (environment_has_safety_margin)
  {
    return points_in_environment_with_safety_margin_.size();
  }
  else
  {
    return points_in_environment_.size();
  }
}

const VisiLibity::Environment &Environment::visilibity_environment() const
{
  if (environment_has_safety_margin)
  {
    return visilibity_environment_with_safety_margin_;
  }
  else
```

```
    {
      return visilibity_environment_;
    }
  }

  void Environment::clip_against_outer_boundary(ClipperLib::Paths &paths_in,
                                                ClipperLib::Paths &paths_out)
  {
    ClipperLib::Clipper clipper;
    clipper.AddPath(paths_in[0], ClipperLib::ptClip, true);

    for (int i = 1; i < paths_in.size(); ++i)
    {
      clipper.AddPath(paths_in[i], ClipperLib::ptSubject, true);
    }
    ClipperLib::Paths temp_paths;
    clipper.Execute(ClipperLib::ctIntersection, temp_paths, ClipperLib::pftEvenOdd);
    paths_out.push_back(paths_in[0]);

    for (int i = 0; i < temp_paths.size(); ++i)
    {
      reverse_path(temp_paths[i]);
      paths_out.push_back(temp_paths[i]);
    }
  }

  void Environment::add_constant_safety_margin(double margin)
  {
    ClipperLib::ClipperOffset co;
    //Add safety margin
```

```
    for (int i = 1; i < paths_.size(); ++i)
    {
      co.AddPath(paths_[i], ClipperLib::jtMiter, ClipperLib::etClosedPolygon);
    }
    ClipperLib::Paths tempPaths;
    co.Execute(tempPaths, margin);

    //clip against outer boundary
    ClipperLib::Clipper clipper;
    clipper.AddPath(paths_[0], ClipperLib::ptClip, true);
    for (int i = 0; i < tempPaths.size(); ++i)
    {
      clipper.AddPath(tempPaths[i], ClipperLib::ptSubject, true);
    }
    ClipperLib::Paths tempPaths2;
    clipper.Execute(ClipperLib::ctIntersection, tempPaths2, ClipperLib::pftEvenOdd);

    //Outer boundary
    paths_with_safety_margin_.push_back(paths_[0]);

    for (int i = 0; i < tempPaths2.size(); ++i)
    {
      reverse_path(tempPaths2[i]);
      paths_with_safety_margin_.push_back(tempPaths2[i]);
    }

    convert_to_visilibity_environment(paths_with_safety_margin_,
                                      visilibity_environment_with_safety_margin_);

    visilibity_environment_with_safety_margin_.is_valid(epsilon_);
```

```
  environment_has_safety_margin = true;
  update_tug_point_list(paths_with_safety_margin_,
                        points_in_environment_with_safety_margin_);
}

bool Environment::point_is_within_outer_boundary(const Tug::Point point)
{
  if (point.x() >= x_max_-1 || point.x() <= x_min_+1 ||
      point.y() <= y_min_+1 || point.y() >= y_max_-1)
  {
    return false;
  }
  return true;
}

void Environment::reverse_path(ClipperLib::Path &path)
{
  std::reverse(path.begin(), path.end());
}

void Environment::path_to_hole(const ClipperLib::Path &path, VisiLibity::Polygon &hole)
{
  for(int i = 0; i < path.size(); i++)
  {
    int x = (int)path[i].X;
    int y = (int)path[i].Y;
    hole.push_back(VisiLibity::Point(x,y));
  }
}
```

```
void Environment::convert_to_visilibity_environment(const ClipperLib::Paths &paths,
                                                    VisiLibity::Environment &environment)
{
  if (paths.size() == 0)
  {
    return;
  }
  find_max_and_min_in_path(paths[0], 'X', x_max_, x_min_);
  find_max_and_min_in_path(paths[0], 'Y', y_max_, y_min_);

  int x_min_cur, x_max_cur, y_min_cur, y_max_cur;

  for (int i = 1; i < paths.size(); ++i)
  {
    find_max_and_min_in_path(paths[i], 'X', x_max_cur, x_min_cur);
    if (x_max_cur >= x_max_)
    {
      x_max_ = x_max_cur +1 ;
    }
    if (x_min_cur <= x_min_)
    {
      x_min_ = x_min_cur -1 ;
    }
    find_max_and_min_in_path(paths[i], 'Y', y_max_cur, y_min_cur);
    if (y_max_cur >= y_max_)
    {
      y_max_ = y_max_cur + 1 ;
    }
    if (y_min_cur <= y_min_)
```

```
      {
        y_min_ = y_min_cur -1 ;
      }
    }
    ClipperLib::Path path_temp;
    path_temp.push_back(ClipperLib::IntPoint(x_min_, y_min_));
    path_temp.push_back(ClipperLib::IntPoint(x_max_, y_min_));
    path_temp.push_back(ClipperLib::IntPoint(x_max_, y_max_));
    path_temp.push_back(ClipperLib::IntPoint(x_min_, y_max_));

    set_outer_boundary(path_temp, environment);

    for (int i = 1; i < paths.size(); ++i)
    {
      VisiLibity::Polygon hole;
      path_to_hole(paths[i], hole);
      environment.add_hole(hole);
    }
  }

  void Environment::update_tug_point_list(const ClipperLib::Paths &paths,
                                          std::map<int, Point> &tug_points)
  {
    tug_points.clear();

    for (int i = 1; i < paths.size(); ++i)
    {
      for (int j = 0; j < paths[i].size(); ++j)
      {
        int id = ++id_counter_;
```

```
      tug_points.insert(std::pair<int,Point>(id, Point(paths[i][j], *this, id)));
      coordinate_to_id.insert(std::pair<std::pair<double,double>,int>
                              (std::make_pair(paths[i][j].X, paths[i][j].Y), id));
    }
  }
  mark_points_touching_outer_boundary();
  make_visibility_graphs_for_points(tug_points);
}
void Environment::print_coordinates_and_id() const
{
  for (auto i = coordinate_to_id.begin(); i != coordinate_to_id.end(); ++i)
  {
    std::cout << i->second << ": " << i->first.first << ", "
              << i->first.second << std::endl;
  }
}

void Environment::make_visibility_graphs_for_points(std::map<int, Point> &tug_points)
{
  for (std::map<int, Point>::iterator i = tug_points.begin(); i != tug_points.end(); ++i)
  {
    i->second.create_visibility_polygon(*this);
  }
}

/*Polyline Environment::shortest_path(const Point &start, const Point &finish) //, double epsilon)
{
  Polyline shortest_path;

  Shortest_path((*this), start, finish, shortest_path);
```

```
  return shortest_path;
}*/

void Environment::set_outer_boundary(const ClipperLib::Path &outer_boundary,
                                     VisiLibity::Environment &environment)
{
  find_max_and_min_in_path(outer_boundary, 'X', x_max_, x_min_);
  find_max_and_min_in_path(outer_boundary, 'Y', y_max_, y_min_);

  VisiLibity::Polygon outer_boundary_polygon;
  path_to_hole(outer_boundary, outer_boundary_polygon);
  environment.set_outer_boundary(outer_boundary_polygon);
}

void Environment::mark_points_touching_outer_boundary()
{

  for (std::map<int,Point>::iterator pt = begin(); pt != end(); ++pt)
  {
    if (environment_has_safety_margin and point_is_on_outer_boundary(pt->second))
    {
      pt->second.is_on_outer_boundary = true;
    }
    else if(!environment_has_safety_margin and point_is_on_outer_boundary(pt->second))
    {
      pt->second.is_on_outer_boundary = true;
    }
  }
}
```

```
bool Environment::point_is_on_outer_boundary(const VisiLibity::Point &point) const
{
  //outer boundary is 1 unit smaller
  if (point.x() == x_max_-1 || point.x() == x_min_+1 ||
      point.y() == y_min_+1 || point.y() == y_max_-1)
  {
    return true;
  }
  return false;
}

bool Environment::load_from_file(ClipperLib::Paths &ppg,
                                 const std::string& filename,
                                 double scale)
{
  //file format assumes:
  //  1. path coordinates (x,y) are comma separated (+/- spaces) and
  //  each coordinate is on a separate line
  //  2. each path is separated by one or more blank lines
  ppg.clear();
  std::ifstream ifs(filename);
  if (!ifs) return false;
  std::string line;
  ClipperLib::Path pg;
  while (std::getline(ifs, line))
  {
    std::stringstream ss(line);
    double X = 0.0, Y = 0.0;
    if (!(ss >> X))
    {
```

```
      //ie blank lines => flag start of next polygon
      if (pg.size() > 0) ppg.push_back(pg);
      pg.clear();
      continue;
    }
    char c = ss.peek();
    while (c == ' ') {ss.read(&c, 1); c = ss.peek();} //gobble spaces before comma
    if (c == ',') {ss.read(&c, 1); c = ss.peek();} //gobble comma
    while (c == ' ') {ss.read(&c, 1); c = ss.peek();} //gobble spaces after comma
    if (!(ss >> Y)) break; //oops!
    pg.push_back(ClipperLib::IntPoint((ClipperLib::cInt)(X * scale),
                 (ClipperLib::cInt)(Y * scale)));
  }
  if (pg.size() > 0) ppg.push_back(pg);
  ifs.close();

  return true;
}

void Environment::find_max_and_min_in_path(const ClipperLib::Path &path,
                                           char coordinate,
                                           int &max_val,
                                           int &min_val)
{
  if (!(coordinate == 'X' || coordinate == 'Y'))
  {
    return;
  }
  max_val = std::numeric_limits<int>::min();
  min_val = std::numeric_limits<int>::max();
```

```
    int current;
    for (int i = 0; i < path.size(); ++i)
    {
      if (coordinate == 'X')
      {
        current = path[i].X;
      }
      else
      {
        current = path[i].Y;
      }

      if (current < min_val)
      {
        min_val = current;
      }
      else if (current > max_val)
      {
        max_val = current;
      }
    }
  }

  int Environment::find_id(const VisiLibity::Point &point) const
  {
    try
    {
      return coordinate_to_id.at(std::make_pair(point.x(),point.y()));
    }
```

```
  catch(...)
  {
    return -1;
  }

}

void Environment::get_boundaries(int &x_min_out, int &x_max_out,
                                 int &y_min_out, int &y_max_out) const
{
  x_min_out = x_min_+1;
  x_max_out = x_max_-1;
  y_min_out = y_min_+1;
  y_max_out = y_max_-1;
}

float eucledian_distance(const Point &point1, const Point &point2)
{
  return sqrt(pow(point1.x() - point2.x(), 2) + pow(point1.y() - point2.y(), 2));
}

void Environment::mark_points_within_range(float range)
{
  for (std::map<int,Point>::iterator i = begin(); i != end(); ++i)
  {
    for (std::map<int,Point>::iterator j = begin(); j != end(); ++j)
    {
      if (i != j)
      {
        if (eucledian_distance(i->second, j->second) < range)
```

```
          {
            i->second.add_close_point(&j->second);
          }
        }
      }
    }
  }

  void Environment::save_environment_as_svg(const std::string filename)
  {
    Polyline dummy;
    save_environment_as_svg(filename, dummy);
  }

  void Environment::save_environment_as_svg(const std::string filename,
                                            const Polyline &shortest_path)
  {
    std::vector<Polyline> shortest_paths;
    shortest_paths.push_back(shortest_path);
    save_environment_as_svg(filename, shortest_paths);
  }

  void Environment::save_environment_as_svg(const std::string filename,
                                            const std::vector<Polyline> &shortest_paths)
  {
    SVGBuilder svg;
    svg.style.brushClr = 0x129C0000;
    svg.style.penClr = 0xCCFFA07A;
    svg.style.pft = ClipperLib::pftEvenOdd;
```

```
    if (environment_has_safety_margin)
    {
      svg.AddPaths(paths_with_safety_margin_);
    }
    svg.AddPaths(paths_);

    for (int i = 0; i < shortest_paths.size(); ++i)
    {
      if (shortest_paths[i].size()>0)
      {
        svg.AddPolyline(shortest_paths[i]);
      }
    }
    svg.SaveToFile(filename, 1, 0);
  }
}
```

tug_point.cpp

```
#include "tug_point.hpp"
#include "tug_environment.hpp"

namespace Tug
{
  Point::Point(const ClipperLib::IntPoint &point,  int point_id)  : point_id_(point_id)
  {
    set_x((double)point.X);
    set_y((double)point.Y);

  }

  Point::Point(double x_temp, double y_temp, const Environment &environment,  int point_id)  : point_id_(point_id)
  {
    x_ = x_temp;
    y_ = y_temp;
    //visibility_polygon_ = VisiLibity::Visibility_Polygon(*this, environment.visilibity_environment(), 0.001);
    create_visibility_polygon(environment);

  }

  Point::Point(const ClipperLib::IntPoint &point, const Environment &environment, int point_id) : point_id_(point_id)
  {
    set_x((double)point.X);
    set_y((double)point.Y);
    //create_visibility_polygon(environment);
  }
```

```
Point::Point(const Point &obj) : point_id_(obj.point_id_)
{
  x_ = obj.x_;
  y_ = obj.y_;
  is_on_outer_boundary = obj.is_on_outer_boundary;
  visibility_polygon_ = obj.visibility_polygon_;
  visible_vertices_ = obj.visible_vertices_;
}

Point &Point::operator=(const Point &other)
{
  if(&other == this)
    return *this;

  this->x_ = other.x();
  this->y_ = other.y();
  this->is_on_outer_boundary = other.is_on_outer_boundary;
  this->visibility_polygon_ = other.visibility_polygon();
  this->visible_vertices_ = other.visible_vertices();
  return *this;
}

bool Point::operator==(const Point &other) const
{
  if (this->x() == other.x() && this->y() == other.y())
  {
    return true;
  }
  else return false;
}
```

```
bool Point::is_visible(const Tug::Point &point) const
{
  return point.in(visibility_polygon_, 0.01);
}

void Point::create_visibility_polygon(const Environment &environment)
{
  visibility_polygon_.clear();
  visible_vertices_.clear();

  visibility_polygon_ = VisiLibity::Visibility_Polygon(*this,
                                    environment.visilibity_environment(), 0.001);

  for (auto i = environment.const_begin(); i != environment.const_end(); ++i)
  {
    if (i->second.in(visibility_polygon_, 0.001) and !i->second.is_on_outer_boundary)
    {
      visible_vertices_.push_back(i->first);
    }
  }

}

void Point::add_close_point(Point *pt)
{
  if (pt != this)
  {
    close_points.push_back(pt);
  }
```

```
  }


  std::ostream& operator<<(std::ostream &out, Point const &pt)
  {
    out << "(" << pt.x() << ", " << pt.y() << ")";
    return out;
  }
}
```

C.2.3 test

test.cpp

```
#include "tug_environment.hpp"
#include <gtest/gtest.h>

//TUG_POINT start
TEST(ConstructorTest, XAndYSetCorrectly)
{
    Tug::Point pt(1.0, 2.0);
    ASSERT_EQ(1.0, pt.x());
    ASSERT_EQ(2.0, pt.y());
}
//TUG_POINT end

//TUG_ENVIRONMENT start
TEST(TugEnvironmentTest, PointsOnBoundaryMarkedCorrectly)
{
  Tug::Environment tug_environment(
    "test_environment.txt", 1.0, 0.01);

  tug_environment.add_constant_safety_margin(50);

  //These are manually checked that they are on boundary
  bool on_boundary[] = {1,1,0,0,0,0,1,1,1,0,0,0,0,1,1,0,0};
  int a = 0;
  for (std::map<int, Tug::Point>::const_iterator pt = tug_environment.const_begin();
                                                 pt != tug_environment.const_end();
                                                 ++pt)
```

```
  {
    if (on_boundary[a])
    {
      EXPECT_TRUE(pt->second.is_on_outer_boundary);
    }
    else
    {
      EXPECT_FALSE(pt->second.is_on_outer_boundary);
    }
    a++;
  }
}

TEST(TugEnvironmentTest, AllPointsWithinOuterBoundary)
{
  Tug::Environment tug_environment(
        "test_environment_out_of_boundary.txt",
         1.0, 0.01);

  Tug::Point pt0(350, 260,tug_environment);
  Tug::Point pt1(320, 260,tug_environment);
  Tug::Point pt2(320, 350,tug_environment);
  Tug::Point pt3(350, 350,tug_environment);

  EXPECT_EQ(tug_environment(1), pt0);
  EXPECT_EQ(tug_environment(2), pt1);
  EXPECT_EQ(tug_environment(3), pt2);
  EXPECT_EQ(tug_environment(4), pt3);
  int a =1;
  EXPECT_EQ(a,1);
```

```
}
//TUG_ENVIRONMENT end


int main(int argc, char **argv)
{
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

C.3 Search

Contents

- shortest_path_node.hpp
- tug_a_star_search.hpp
- tug_all_pairs_shortest_path.hpp
- tug_route_around_ship.hpp
- tug_shortest_path.hpp
- tug_a_star_search.cpp
- tug_all_pairs_shortest_path.cpp
- tug_route_around_ship.cpp
- tug_shortest_path.cp
- test.cpp

C.3.1 include

shortest_path_node.hpp

```
/**
 * \file visilibity.cpp
 * \author Karl J. Obermeyer
 * \date March 20, 2008
 * \Modified by Rebecca Cox June 13, 2017
\remarks
VisiLibity:  A Floating-Point Visibility Algorithms Library,
Copyright (C) 2008  Karl J. Obermeyer (karl.obermeyer [ at ] gmail.com)

This file is part of VisiLibity.

VisiLibity is free software: you can redistribute it and/or modify it under
the terms of the GNU Lesser General Public License as published by the
Free Software Foundation, either version 3 of the License, or (at your
option) any later version.

VisiLibity is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public
License for more details.

You should have received a copy of the GNU Lesser General Public
License along with VisiLibity.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef SHORTEST_PATH_NODE_H
```

```
#define SHORTEST_PATH_NODE_H

namespace Tug
{
  class Shortest_Path_Node
    {
    public:
      //flattened index of corresponding Environment vertex
      //convention vertex_index = n() => corresponds to start Point
      //vertex_index = n() + 1 => corresponds to finish Point
      unsigned vertex_index;
      //pointer to self in search tree.
      std::list<Shortest_Path_Node>::iterator search_tree_location;
      //pointer to parent in search tree.
      std::list<Shortest_Path_Node>::iterator parent_search_tree_location;
      //Geodesic distance from start Point.
      double cost_to_come;
      //Euclidean distance to finish Point.
      double estimated_cost_to_go;
      //std::vector<Shortest_Path_Node> expand();
      bool operator < (const Shortest_Path_Node& spn2) const
      {
        double f1 = this->cost_to_come + this->estimated_cost_to_go;
        double f2 = spn2.cost_to_come + spn2.estimated_cost_to_go;
        if( f1 < f2 )
          return true;
        else if( f2 < f1 )
          return false;
        else if( this->vertex_index < spn2.vertex_index )
          return true;
```

```
      else if( this->vertex_index > spn2.vertex_index )
        return false;
      else if( &(*(this->parent_search_tree_location))
         < &(*(spn2.parent_search_tree_location)) )
        return true;
      else
        return false;
    }
    // print member data for debugging
    void print() const
    {
std::cout << "         vertex_index = " << vertex_index << std::endl
    << "parent's vertex_index = "
    << parent_search_tree_location->vertex_index
    << std::endl
    << "         cost_to_come = " << cost_to_come << std::endl
    << " estimated_cost_to_go = "
    << estimated_cost_to_go << std::endl;
    }
  };
}

#endif //SHORTEST_PATH_NODE_H
```

tug_a_star_search.hpp

```
#ifndef TUG_A_STAR_SEARCH_H
#define TUG_A_STAR_SEARCH_H

#include "geometry/tug_environment.hpp"
#include "shortest_path_node.hpp"

namespace Tug
{
  class A_star_search
  {
  public:
    A_star_search(const Point &start,
                  const Point &finish,
                  const std::vector<Point> &points,
                  Polyline &shortest_path,
                  double epsilon);
    ~A_star_search(){};
  private:
    Polyline best_first_search(const Point &start,
                               const Point &finish,
                               const std::vector<Point> &points_in_environment);
    double epsilon_ = 0;
    double heurestic(const Point &point1, const Point &point2);
    double eucledian_distance(const Point &point1, const Point &point2);
    bool   trivial_case(const Point &start,
                        const Point &finish,
                        Polyline &shortest_path_output);
    void attach_child(Shortest_Path_Node *child, Shortest_Path_Node *current_node,
```

```
                        std::vector<Shortest_Path_Node> &children,
                        const VisiLibity::Environment &environment,
                        const Point &finish,
                        int index);
    void reconstruct_path(Polyline &shortest_path_output,
                          Shortest_Path_Node &current_node,
                          const Point &start,
                          const Point &finish,
                          const std::vector<Point> &points_in_environment);

  };
}

#endif //TUG_A_STAR_SEARCH_H
```

tug_all_pairs_shortest_path.hpp

```
#ifndef TUG_ALL_PAIRS_SHORTEST_PATH_H
#define TUG_ALL_PAIRS_SHORTEST_PATH_H

#include "tug_a_star_search.hpp"
#include "vector"
#include "map"

namespace Tug
{
        class All_pairs_shortest_path
        {
        public:
                All_pairs_shortest_path( Environment &env);
    void write_to_file( std::vector<std::vector<int>> &apsp);

    std::vector<std::vector<int>> get_apsp_matrix() const {return apsp_;};
    std::map<std::pair<int,int>, int> get_apsp_matrix_2() const {return apsp2_;};
    std::map<std::pair<int,int>, double> get_apsp_costs() const {return apsp_costs_;};

        private:
                std::vector<std::vector<Point>> optimal_paths;
                double epsilon_ = 0.001;
    void find_optimal_path_from_all_points_2(Environment &env, std::map<std::pair<int,int>, int> &apsp);
    std::vector<std::vector<int>> find_optimal_path_from_all_points(Environment &env);
    std::vector<std::vector<int>> apsp_;
    std::map<std::pair<int,int>, int> apsp2_;
    std::map<std::pair<int,int>, double> apsp_costs_;
        };
```

```
}
#endif
```

tug_route_around_ship.hpp

```
#ifndef TUG_ROUTE_AROUND_SHIP
#define TUG_ROUTE_AROUND_SHIP

#include "geometry/tug_point.hpp"
#include "geometry/tug_polyline.hpp"

#include <Eigen/Dense>

namespace Tug
{
class Route_around_ship
  {
  public:
    Route_around_ship(double orientation, double width, double length);
    Route_around_ship(){};
    void move(const Point &mid_pt, double orientation);
    Polyline best_route(Point start, Point finish);
    Point get_ship_position(){return position_;}
    std::vector<double> ship_corners();
  private:
    void rotate_ship(double angle, Eigen::Matrix<double,2,4> &ship_mat_);
    void calculate_corners(const Point &mid_pt, double orientation,
                           double width, double length,
                           Eigen::Matrix<double,2,4> &ship_mat_);
    int orientation(const Point &p,const Point &q,const Point &r);
    bool do_cross(const Point &p1, const Point &q1, const Point &p2, const Point &q2);
    double dist(const Point &point1, const Point &point2);
    int min_element_index(double list[4]);
```

```
    bool ship_placed_ = false;
    Eigen::Matrix<double,2,4> ship_mat_;
    double ship_corners_[4][2];
    Point position_;
    double orientation_;
  };
}

#endif //TUG_ROUTE_AROUND_SHIP
```

tug_shortest_path.hpp

```
#ifndef TUG_SHORTEST_PATH_H
#define TUG_SHORTEST_PATH_H

#include "geometry/tug_environment.hpp"
#include "geometry/tug_point.hpp"
#include "tug_a_star_search.hpp"
#include "tug_all_pairs_shortest_path.hpp"

#include <limits>
#include <map>
#include <math.h>

namespace Tug
{
  class Shortest_path
  {
  public:
    Shortest_path(Environment &environment, const Point &start,
                  const Point &finish, Polyline &shortest_path);
    Shortest_path(const std::string &all_pairs_shortest_path, Environment &environment);
    Shortest_path(Environment &environment);

    bool calculate_shortest_path(const Point &start,
                                 const Point &end,
                                 Polyline &shortest_path,
                                 Environment &environment);
  private:
    std::vector<std::vector<int>> apsp_;
```

```
    std::map<std::pair<int,int>, int> apsp2_;
    bool read_file(const std::string &filename);
    bool extract_shortest_path(int start_id, int finish_id,
                               Polyline &shortest_path, Environment &environment);
    bool calculate_shortest_path_outside_safety_margin(const Point &start,
                                                       const Point &end,
                                                       Polyline &shortest_path,
                                                       Environment &environment);
    void set_waypoints(Polyline &shortest_path);
    bool start_and_end_points_are_valid(const Point &start, const Point &finish,
                                        const Tug::Environment &environment);
    int point_within_safety_margin(const Point &point, const Tug::Environment &environment);
    Point point_closest_to_line_segment(const Point &point,
                                        const VisiLibity::Line_Segment &line);
    Point calculate_point_on_boundary(const Point &point,
                                      const VisiLibity::Polygon &hole,
                                      const Tug::Environment &env);
    std::map<std::pair<int,int>, double> apsp_costs_;
  };
}

#endif //TUG_SHORTEST_PATH_H
```

C.3.2 src

tug_a_star_search.cpp

```
/**
 * \file visilibity.cpp
 * \author Karl J. Obermeyer
```

```
 * \date March 20, 2008
   \Modified by Rebecca Cox June 13 2017
 *
\remarks
VisiLibity:  A Floating-Point Visibility Algorithms Library,
Copyright (C) 2008  Karl J. Obermeyer (karl.obermeyer [ at ] gmail.com)

This file is part of VisiLibity.

VisiLibity is free software: you can redistribute it and/or modify it under
the terms of the GNU Lesser General Public License as published by the
Free Software Foundation, either version 3 of the License, or (at your
option) any later version.

VisiLibity is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the GNU Lesser General Public
License for more details.

You should have received a copy of the GNU Lesser General Public
License along with VisiLibity.  If not, see <http://www.gnu.org/licenses/>.
*/

#include "tug_a_star_search.hpp"
#include <cmath>

namespace Tug
{
  A_star_search::A_star_search(const Point &start,
                               const Point &finish,
```

```
                              const std::vector<Point> &points,
                              Polyline &shortest_path,
                              double epsilon)
{
  epsilon_ = epsilon;
  shortest_path = best_first_search(start,
                                    finish,
                                    points);
}

double A_star_search::heurestic(const Point &point1, const Point &point2)
{
  return eucledian_distance(point1, point2);
}

double A_star_search::eucledian_distance(const Point &point1, const Point &point2)
{
  return sqrt(pow(point1.x() - point2.x(), 2) + pow(point1.y() - point2.y(), 2));
}


bool A_star_search::trivial_case(const Point &start,
                                 const Point &finish,
                                 Polyline &shortest_path_output)
{
  if( heurestic(start,finish) <= epsilon_ )
  {
    shortest_path_output.push_back(start);
    return true;
  }
```

```
  else if( start.is_visible(finish) )
  {
    shortest_path_output.push_back(start);
    shortest_path_output.push_back(finish);
    return true;
  }
  return false;
}

Polyline A_star_search::best_first_search(const Point &start,
                                          const Point &finish,
                                          const std::vector<Point>
                                                 &points_in_environment)
{

  Polyline shortest_path_output;

  if(trivial_case(start, finish, shortest_path_output))
  {
    return shortest_path_output;
  }
  //Connect start and finish Points to the visibility graph
  bool *start_visible;  //start row of visibility graph
  bool *finish_visible; //finish row of visibility graph
  start_visible = new bool[points_in_environment.size()];
  finish_visible = new bool[points_in_environment.size()];

  for(unsigned k=0; k<points_in_environment.size(); k++)
  {
    if( !points_in_environment[k].is_on_outer_boundary &&
```

```
        //points_in_environment[k].in( start_visibility_polygon , epsilon_ )
        start.is_visible(points_in_environment[k]))
    {
      start_visible[k] = true;
    }
    else
    {
      start_visible[k] = false;
    }
    if( !points_in_environment[k].is_on_outer_boundary &&
        //points_in_environment[k].in( finish_visibility_polygon , epsilon_ )
        finish.is_visible(points_in_environment[k]))
    {
      finish_visible[k] = true;
    }
    else
    {
      finish_visible[k] = false;
    }
  }

  //Initialize search tree of visited nodes
  std::list<Shortest_Path_Node> T;
  //:WARNING:
  //If T is a vector it is crucial to make T large enough that it will not be resized.
  //  If T were resized, any iterators pointing to its contents would be invalidated,
  // thus causing the program to fail.
  //T.reserve( points_in_environment.size() + 3 );

  //Initialize priority queue of unexpanded nodes
```

```
std::set<Shortest_Path_Node> Q;
//Construct initial node
Shortest_Path_Node current_node;
//convention vertex_index == points_in_environment.size() => corresponds to start Point
//vertex_index == points_in_environment.size() + 1 => corresponds to finish Point
current_node.vertex_index = points_in_environment.size();
current_node.cost_to_come = 0;
current_node.estimated_cost_to_go = heurestic(start, finish );
//Put in T and on Q
T.push_back( current_node );
T.begin()->search_tree_location = T.begin();
current_node.search_tree_location = T.begin();
T.begin()->parent_search_tree_location = T.begin();
current_node.parent_search_tree_location = T.begin();
Q.insert( current_node );

//Initialize temporary variables
Shortest_Path_Node child;
//children of current_node
std::vector<Shortest_Path_Node> children;
//flags
bool solution_found = false;
bool child_already_visited = false;

//-----------Begin Main Loop-----------
while( !Q.empty() )
{
  //Pop top element off Q onto current_node
  current_node = *Q.begin();
  Q.erase( Q.begin() );
```

```
        //Check for goal state (if current node corresponds to finish)
        if( current_node.vertex_index == points_in_environment.size() + 1 )
        {
          solution_found = true;
          break;
        }
        //Expand current_node (compute children)
        children.clear();
        //if current_node corresponds to start
        if( current_node.vertex_index == points_in_environment.size() )
        {
          //loop over environment vertices
          for(unsigned i=0; i < points_in_environment.size(); i++)
          {
            if( start_visible[i] )
            {
              //attach_child(&child, &current_node, children, environment, finish, i);
              child.vertex_index = i;
              child.parent_search_tree_location = current_node.search_tree_location;
              child.cost_to_come = heurestic( start , points_in_environment[i] );
              child.estimated_cost_to_go = heurestic( points_in_environment[i] , finish );
              children.push_back( child );
            }
          }
        }
        //else current_node corresponds to a vertex of the environment
        else
        {
          //check which environment vertices are visible
          for(unsigned i=0; i < points_in_environment.size(); i++)
```

```
{
  if( current_node.vertex_index != i )
  {
    if( points_in_environment[current_node.vertex_index].is_visible(
      points_in_environment[i]) && !points_in_environment[i].is_on_outer_boundary)
    {
      //attach_child(&child, &current_node, children, environment, finish, i);

      child.vertex_index = i;
      child.parent_search_tree_location = current_node.search_tree_location;

      child.cost_to_come = current_node.cost_to_come
        + heurestic( points_in_environment[current_node.vertex_index],
                     points_in_environment[i] );

      child.estimated_cost_to_go = heurestic( points_in_environment[i], finish );

      children.push_back( child );
    }
  }
}
//check if finish is visible
if( finish_visible[ current_node.vertex_index ] )
{
  child.vertex_index = points_in_environment.size() + 1; //finish point
  child.parent_search_tree_location = current_node.search_tree_location;

  child.cost_to_come = current_node.cost_to_come
    + heurestic( points_in_environment[current_node.vertex_index] , finish );
  child.estimated_cost_to_go = 0;
```

```
      children.push_back( child );
    }

  }
  //Process children
  for( std::vector<Shortest_Path_Node>::iterator children_itr =
                                                  children.begin();
                                                  children_itr != children.end();
                                                  children_itr++ )
  {
    child_already_visited = false;

    //Check if child state has already been visited (by looking in search tree T)
    for( std::list<Shortest_Path_Node>::iterator T_itr = T.begin();
                                                 T_itr != T.end();
                                                T_itr++ )
    {
      if( children_itr->vertex_index == T_itr->vertex_index )
      {
        children_itr->search_tree_location = T_itr;
        child_already_visited = true;
        break;
      }
    }

    if( !child_already_visited )
    {
      //Add child to search tree T
      T.push_back( *children_itr );
      (--T.end())->search_tree_location = --T.end();
```

```
      children_itr->search_tree_location = --T.end();
      Q.insert( *children_itr );
    }
    else if( children_itr->search_tree_location->cost_to_come >
                                    children_itr->cost_to_come )
    {
      //redirect parent pointer in search tree
      children_itr->search_tree_location->parent_search_tree_location
        = children_itr->parent_search_tree_location;
      //and update cost data
      children_itr->search_tree_location->cost_to_come = children_itr->cost_to_come;
      //update Q
      for(std::set<Shortest_Path_Node>::iterator Q_itr = Q.begin();
                                                  Q_itr!= Q.end();
                                                    Q_itr++)
      {
        if( children_itr->vertex_index == Q_itr->vertex_index )
        {
          Q.erase( Q_itr );
          break;
        }
      }
      Q.insert( *children_itr );
    }

    if( !child_already_visited )
    {
      Q.insert( *children_itr );
    }
  }
```

```
    }
    //-----------End Main Loop-----------

    //Recover solution
    if( solution_found )
    {
      reconstruct_path(shortest_path_output, current_node, start,
                       finish, points_in_environment);
    }
    //free memory
    delete [] start_visible;
    delete [] finish_visible;
    return shortest_path_output;
  }

  void A_star_search::reconstruct_path(Polyline &shortest_path_output,
                                       Shortest_Path_Node &current_node,
                                       const Point &start,
                                       const Point &finish,
                                       const std::vector<Point> &points_in_environment)
{
      shortest_path_output.push_back( finish );
      std::list<Shortest_Path_Node>::iterator backtrace_itr =
                    current_node.parent_search_tree_location;

      const Point *waypoint;

      while( true )
      {
        if( backtrace_itr->vertex_index < points_in_environment.size() )
```

```
        {
          waypoint = &points_in_environment[ backtrace_itr->vertex_index ];
        }
        else if( backtrace_itr->vertex_index == points_in_environment.size() )
        {
          waypoint = &start;
        }
        //Add vertex if not redundant
        if( shortest_path_output.size() > 0
            and heurestic( shortest_path_output[ shortest_path_output.size()- 1 ],
              *waypoint ) > epsilon_ )
        {
         shortest_path_output.push_back(*waypoint);
        }
        if( backtrace_itr->cost_to_come == 0 )
        {
          break;
        }
        backtrace_itr = backtrace_itr->parent_search_tree_location;
      }
      shortest_path_output.reverse();
  }
}
```

tug_all_pairs_shortest_path.cpp

```
#include "tug_all_pairs_shortest_path.hpp"
#include "fstream"
#include "limits"

namespace Tug
{
  All_pairs_shortest_path::All_pairs_shortest_path( Environment &env)
  {
    apsp_ = find_optimal_path_from_all_points(env);
    find_optimal_path_from_all_points_2(env, apsp2_);
    //write_to_file(apsp_);
  }

  void All_pairs_shortest_path::write_to_file( std::vector<std::vector<int>> &apsp)
  {
    std::ofstream file_output;
    file_output.open ("all_pairs_shortest_path.txt");
    for (int i = 0; i < apsp.size(); i++)
    {
      for (int j = 0; j < apsp[i].size(); ++j)
      {
        file_output << apsp[i][j] << "  ";
      }
      file_output << "\n";
    }
    std::cout << "All pairs shortest path written to file\n";
    file_output.close();
  }
```

```
std::vector<std::vector<int>>
All_pairs_shortest_path::find_optimal_path_from_all_points( Environment &env)
{
  std::vector<std::vector<int>> optimal_vertex(env.n());
  for (int i = 0; i < env.n(); ++i)
  {
    optimal_vertex[i] = std::vector<int>(env.n());
  }
  Polyline shortest_path_temp;

  int a = 0;
    for (std::map<int,Point>::iterator i = env.begin();
                                       i != env.end();
                                       ++i)
    {
      int b = 0;

      for (std::map<int,Point>::iterator j = env.begin();
                                         j != env.end();
                                         ++j)
      {
        if (a==b)
        {
          optimal_vertex[a][b] = 0;
        }
        else
        {
          A_star_search(i->second, //start point
                        j->second, //end point
```

```
                        env.points(),
                        shortest_path_temp,
                        epsilon_);
          if (shortest_path_temp.size() > 0)
          {
            optimal_vertex[a][b] = shortest_path_temp[1].id();
          }
          else
          {
            optimal_vertex[a][b] = -1;
          }
        }
        b++;
      }
      a++;
    }
    return optimal_vertex;

}

void All_pairs_shortest_path::find_optimal_path_from_all_points_2(
                              Environment &env,
                              std::map<std::pair<int,int>, int> &apsp)
{
  // map<pair<from, to>, go_via>>
  Polyline shortest_path_temp;
  apsp.clear();

    for (std::map<int,Point>::iterator i = env.begin(); i != env.end(); ++i)
    {
```

```
          for (std::map<int,Point>::iterator j = env.begin(); j != env.end(); ++j)
          {
            std::pair<int,int> pair_of_points(i->first, j->first);
            if (i->first == j->first)
            {
              apsp.insert(std::pair<std::pair<int,int>,int>(pair_of_points, 0));
              apsp_costs_.insert(std::pair<std::pair<int,int>,double>(pair_of_points, 0.0));
            }
            else
            {
              A_star_search(i->second, //start point
                            j->second, //end point
                            env.points(),
                            shortest_path_temp,
                            epsilon_);
              if (shortest_path_temp.size() > 0)
              {
                apsp.insert(std::pair<std::pair<int,int>,int>(pair_of_points, shortest_path_temp[1].id()));
                apsp_costs_.insert(std::pair<std::pair<int,int>,double>(pair_of_points, shortest_path_temp.length()));
              }
              else
              {
                apsp.insert(std::pair<std::pair<int,int>,int>(pair_of_points, -1));
                apsp_costs_.insert(std::pair<std::pair<int,int>,double>(pair_of_points, std::numeric_limits<double>::max(
              }
            }
          }
        }
      }
  }
```

tug_route_around_ship.cpp

```
#include <tug_route_around_ship.hpp>
namespace Tug
{
  Route_around_ship::Route_around_ship(double orientation, double width, double length)
  {
    orientation_ = 0;
    position_ = Point(-1000,-1000,-1);
    ship_mat_ = Eigen::Matrix<double,2,4>(2,4);

    calculate_corners(position_, orientation, width, length, ship_mat_);
    rotate_ship(orientation, ship_mat_);
  }

  std::vector<double> Route_around_ship::ship_corners()
  {
    std::vector<double> temp;
    for (int i = 0; i < 4; ++i)
    {
      for (int j = 0; j < 2; ++j)
      {
        temp.push_back(ship_mat_(j,i));
      }
    }
    return temp;
  }


  void Route_around_ship::move(const Point &mid_pt, double orientation)
```

```
{
  Eigen::Translation2d transl(mid_pt.x() - position_.x(), mid_pt.y() - position_.y());
  Eigen::Affine2d af(transl);
  position_ = mid_pt;
  ship_mat_ = af*ship_mat_;
  rotate_ship(orientation, ship_mat_);
}


void Route_around_ship::rotate_ship(double angle, Eigen::Matrix<double,2,4> &ship_mat)
{
  Eigen::Translation<double, 2> trans1(-position_.x(), -position_.y());
  Eigen::Rotation2D<double> rot(angle - orientation_); //TODO: Riktig retnining??
  Eigen::Translation<double, 2> trans2(position_.x(), position_.y());

  ship_mat = trans2*rot*trans1*ship_mat;
  orientation_ = angle;

}

void Route_around_ship::calculate_corners(const Point &mid_pt,
                                          double orientation,
                                          double width,
                                          double length,
                                          Eigen::Matrix<double,2,4> &ship_mat)
{
  double x = mid_pt.x();
  double y = mid_pt.y();
  double half_length = length/2.0;
  double half_width = width/2.0;
```

```
    ship_mat(0, 0) = x - half_length;
    ship_mat(1, 0) = y - half_width;

    ship_mat(0, 1) = x - half_length;
    ship_mat(1, 1) = y + half_width;

    ship_mat(0, 2) = x + half_length;
    ship_mat(1, 2) = y + half_width;

    ship_mat(0, 3) = x + half_length;
    ship_mat(1, 3) = y - half_width;

}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int Route_around_ship::orientation(const Point &p, const Point &q, const Point &r)
{
    // See http://www.geeksforgeeks.org/orientation-3-ordered-points/
    // for details of below formula.
    int val = (q.y() - p.y()) * (r.x() - q.x()) -
              (q.x() - p.x()) * (r.y() - q.y());

    if (val == 0) return 0;  // colinear

    return (val > 0)? 1: 2; // clock or counterclock wise
```

```
}

// The main function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool Route_around_ship::do_cross(const Point &p1, const Point &q1,
                                 const Point &p2, const Point &q2)
{
  //Intersectiong endpoints will never give crossing lines,
  //exept when they are all the same
  if (p1 == p2 || p1 == q2 || q1 == p2 || q1 == q2)
  {
    return false;
  }
  int o1 = orientation(p1, q1, p2);
  int o2 = orientation(p1, q1, q2);
  int o3 = orientation(p2, q2, p1);
  int o4 = orientation(p2, q2, q1);

  if (o1 != o2 && o3 != o4)
  {
    return true;
  }
  return false;
}

double Route_around_ship::dist(const Point &point1, const Point &point2)
{
  return sqrt(pow(point1.x() - point2.x(), 2) + pow(point1.y() - point2.y(), 2));
}
```

```
int Route_around_ship::min_element_index(double list[4])
{
  int current_min_index = 0;
  double current_min = list[0];
  for (int i = 1; i < 4; ++i)
  {
    if (list[i] < current_min)
    {
      current_min_index = i;
      current_min = list[i];
    }
  }
  return current_min_index;
}

Polyline Route_around_ship::best_route(Point start, Point finish)
{
  Polyline route;
  Point ship[5];
  for (int i = 0; i < 4; ++i)
  {
    ship[i] = Point(ship_mat_(0,i), ship_mat_(1,i), -1);
  }
  ship[4] = Point(ship_mat_(0,0), ship_mat_(1,0), -1);

  bool intersection[4];
  for (int i = 0; i < 4; ++i)
  {
    intersection[i] = do_cross(ship[i], ship[i+1], start, finish);
  }
```

```
int num_intersections = 0;
for (int i = 0; i < 4; ++i)
{
  if (intersection[i])
  {
    num_intersections++;
  }
}
if (num_intersections < 2)
{
  return route;
}
else if(intersection[0] && intersection[2])
{
  double alternatives[4];
  alternatives[0] = dist(start, ship[0]) + dist(ship[0], ship[3]) +
                    dist(ship[3], finish);
  alternatives[1] = dist(start, ship[3]) + dist(ship[3], ship[0]) +
                    dist(ship[0], finish);
  alternatives[2] = dist(start, ship[1]) + dist(ship[1], ship[2]) +
                    dist(ship[2], finish);
  alternatives[3] = dist(start, ship[2]) + dist(ship[2], ship[1]) +
                    dist(ship[1], finish);

  int best = min_element_index(alternatives);

  if(best == 0){
    route.push_back(ship[0]);
    route.push_back(ship[3]);
```

```
    }
    else if(best == 1){
      route.push_back(ship[3]);
      route.push_back(ship[0]);
    }
    else if(best == 2)
    {
      route.push_back(ship[1]);
      route.push_back(ship[2]);
    }
    else if(best == 3)
    {
      route.push_back(ship[2]);
      route.push_back(ship[1]);
    }
  }
  else if(intersection[1] && intersection[3])
  {
    double alternatives[4];
    alternatives[0] = dist(start, ship[0]) + dist(ship[0], ship[1]) +
                      dist(ship[1], finish);
    alternatives[1] = dist(start, ship[1]) + dist(ship[1], ship[0]) +
                      dist(ship[0], finish);
    alternatives[2] = dist(start, ship[3]) + dist(ship[3], ship[2]) +
                      dist(ship[2], finish);
    alternatives[3] = dist(start, ship[2]) + dist(ship[2], ship[3]) +
                      dist(ship[3], finish);

    int best = min_element_index(alternatives);
```

```
      if(best == 0)
      {
        route.push_back(ship[0]);
        route.push_back(ship[1]);
      }
      else if(best == 1)
      {
        route.push_back(ship[1]);
        route.push_back(ship[0]);
      }
      else if(best == 2)
      {
        route.push_back(ship[3]);
        route.push_back(ship[2]);
      }
      else if(best == 3)
      {
        route.push_back(ship[2]);
        route.push_back(ship[3]);
      }
    }
    else
    {
      if (intersection[0] && intersection[1])
      {
        route.push_back(ship[1]);
      }
      else if(intersection[1] && intersection[2])
      {
        route.push_back(ship[2]);
```

```
      }
      else if(intersection[2] && intersection[3])
      {
        route.push_back(ship[3]);
      }
      else if(intersection[3] && intersection[0])
      {
        route.push_back(ship[0]);
      }
    }
    std::cout << "Route intersects ship. " << std::endl;
    return route;
  }
}
```

tug_shortest_path.cpp

```
#include "tug_shortest_path.hpp"
#include "limits"
#include "memory"

namespace Tug
{
  Shortest_path::Shortest_path(Tug::Environment &environment, const Point &start,
                               const Point &end, Polyline &shortest_path)
  {
    Point second_to_last_point;
    Point second_point;

    if(!start_and_end_points_are_valid(start, end, environment))
    {
      return;
    }

    std::shared_ptr<Point> start_point_to_a_star;
    std::shared_ptr<Point> end_point_to_a_star;

    int index_start = point_within_safety_margin(start,environment);

    if (index_start > 0)
    {
      std::cout << "Start point within safety margin of obstacle " <<
                   index_start<< std::endl;
      second_point = calculate_point_on_boundary(start,
        environment.visilibity_environment_with_safety_margin_[index_start],
```

```
                                                                environment);
}

int index_end = point_within_safety_margin(end,environment);
if (index_end > 0)
{
  std::cout << "end point within safety margin of obstacle " << index_end << std::endl;
  second_to_last_point = calculate_point_on_boundary(end,
    environment.visilibity_environment_with_safety_margin_[index_end], environment);
  std::cout << second_to_last_point << std::endl;
}

double epsilon = 0.01;
if (index_start > 0)
{
  start_point_to_a_star = std::make_shared<Point>(second_point);
}
else
{
  start_point_to_a_star = std::make_shared<Point>(start);

}
if (index_end > 0)
{
  end_point_to_a_star = std::make_shared<Point>(second_to_last_point);
}
else
{
  end_point_to_a_star = std::make_shared<Point>(end);
}
```

```
 A_star_search(*start_point_to_a_star,
               *end_point_to_a_star,
               environment.points(),
               shortest_path,
               epsilon);

  if (shortest_path.size() > 0 and shortest_path[shortest_path.size()-1] != end)
  {
    shortest_path.push_back(end);
  }

  if (shortest_path.size() > 0 and shortest_path[0] != start)
  {
    shortest_path.reverse();
    shortest_path.push_back(start);
    shortest_path.reverse();
  }
}

Shortest_path::Shortest_path(Environment &environment)
{
  All_pairs_shortest_path all_pairs_shortest_path(environment);
  apsp_ = all_pairs_shortest_path.get_apsp_matrix();
  apsp2_ = all_pairs_shortest_path.get_apsp_matrix_2();
  apsp_costs_ = all_pairs_shortest_path.get_apsp_costs();
}


Shortest_path::Shortest_path(const std::string &all_pairs_shortest_path,
```

```
                                Environment &environment)
{
  std::cout << "Reading file: " << all_pairs_shortest_path << std::endl;

  if (!read_file(all_pairs_shortest_path))
  {
    std::cout << "Could not read file: " << all_pairs_shortest_path << std::endl;
  }
}

bool Shortest_path::calculate_shortest_path(const Point &start,
                                            const Point &end,
                                            Polyline &shortest_path,
                                            Environment &environment)
{
  if(!start_and_end_points_are_valid(start, end, environment))
  {
    return false;
  }

  std::shared_ptr<Point> start_point_outside_safety_margin;
  std::shared_ptr<Point> end_point_outside_margin;


  Point second_to_last_point;
  Point second_point;

  int index_start = point_within_safety_margin(start,environment);

  if (index_start > 0)
```

```
{
  second_point = calculate_point_on_boundary(start,
    environment.visilibity_environment_with_safety_margin_[index_start], environment); //todo: maybe one off
}

int index_end = point_within_safety_margin(end,environment);
if (index_end > 0)
{
  second_to_last_point = calculate_point_on_boundary(end,
  environment.visilibity_environment_with_safety_margin_[index_end], environment);
  std::cout << second_to_last_point << std::endl;
}

double epsilon = 0.001;

if (index_start > 0)
{
  start_point_outside_safety_margin = std::make_shared<Point>(second_point);
}
else
{
  start_point_outside_safety_margin = std::make_shared<Point>(start);
}
if (index_end > 0)
{
  end_point_outside_margin = std::make_shared<Point>(second_to_last_point);
}
else
{
  end_point_outside_margin = std::make_shared<Point>(end);
```

```
    }

    bool ok = calculate_shortest_path_outside_safety_margin(
                                                *start_point_outside_safety_margin,
                                                *end_point_outside_margin,
                                                shortest_path,
                                                environment);
    if (shortest_path.size() > 0 and shortest_path[shortest_path.size()-1] != end)
    {
      shortest_path.push_back(end);
    }

    if (shortest_path.size() > 0 and shortest_path[0] != start)
    {
      shortest_path.reverse();
      shortest_path.push_back(start);
      shortest_path.reverse();
    }
    return ok;
  }

  double cost(const Point &pt1, const Point &pt2)
  {
    return sqrt(pow(pt1.x() - pt2.x(), 2) + pow(pt1.y() - pt2.y(), 2));
  }

  bool Shortest_path::calculate_shortest_path_outside_safety_margin(
                                                    const Point &start,
                                                    const Point &end,
                                                    Polyline &shortest_path,
```

```
                                                    Environment &environment)
{
  bool valid_path = true;

  //VisiLibity::Visibility_Polygon start_vp = start.visibility_polygon();
  shortest_path.clear();
  std::vector<int> visible_vertices_start = start.visible_vertices();
  std::vector<int> visible_vertices_end = end.visible_vertices();

  if (start.is_visible(end))
  {
    shortest_path.push_back(start);
    shortest_path.push_back(end);
    return true;
  }

  std::vector<std::pair<int, double>> costs_from_start;
  std::vector<std::pair<int, double>> costs_from_end;

  //calculate cost from start and end points to all points in their
  //respectively visibility polygon
  for (int i = 0; i < visible_vertices_start.size(); ++i)
  {
    int id = visible_vertices_start[i];
    costs_from_start.push_back(std::pair<int,double>(id, cost(start, environment(id))));
  }
  for (int i = 0; i < visible_vertices_end.size(); ++i)
  {
    int id = visible_vertices_end[i];
    costs_from_end.push_back(std::pair<int,double>(id, cost(end, environment(id))));
```

```
    }

    double lowest_cost = std::numeric_limits<double>::max();
    int best_start = -1;
    int best_end = -1;
    //Loop through all points seen by start
    for (int i = 0; i < costs_from_start.size(); ++i)
    {
      int start_id = costs_from_start[i].first;
      //Find shortest path from current point seen from start to all points seen by end
      for (int j = 0; j < costs_from_end.size(); ++j)
      {
        int end_id = costs_from_end[j].first;

        Polyline shortest_path_internal;

        bool path_found = extract_shortest_path(start_id, end_id, shortest_path_internal,
                                                environment);

        if (path_found)
        {
          double current_cost = costs_from_start[i].second
                              + apsp_costs_.at(std::make_pair(start_id, end_id))
                              + costs_from_end[j].second;

          if (current_cost < lowest_cost)
          {
            best_start = start_id;
            best_end = end_id;
            lowest_cost = current_cost;
```

```
        }
      }
    }
  }

  //If a path is found
  if (best_start > -1 and best_end > -1)
  {
    shortest_path.push_back(start);
    Polyline shortest_path_internal;
    extract_shortest_path(best_start, best_end, shortest_path_internal, environment);

    for (int i = 0; i < shortest_path_internal.size(); ++i)
    {
      shortest_path.push_back(shortest_path_internal[i]);
    }
    shortest_path.push_back(end);
    return true;
  }
  return false;
}

bool Shortest_path::extract_shortest_path(int start_id,
                                          int finish_id,
                                          Polyline &shortest_path,
                                          Environment &environment)
{
  shortest_path.clear();
  shortest_path.push_back(environment(start_id));
```

```
    //First vertex to visit. If it is equal to start_id, than
    //there are not more points to traverse
    int next_vertex = apsp2_[std::make_pair(start_id, finish_id)];

    if (next_vertex == -1)
    {
      shortest_path.clear();
      return false;
    }
    int prev_vertex = 0;

    while (next_vertex != 0) //prev_vertex)
    {
      shortest_path.push_back(environment(next_vertex));

      prev_vertex = next_vertex;
      next_vertex = apsp2_[std::make_pair(prev_vertex, finish_id)];

      if (next_vertex == -1)
      {
        shortest_path.clear();
        return false;
      }
    }
    return true;
  }

  bool Shortest_path::read_file(const std::string &filename)
  {
    std::ifstream ifs(filename);
```

```
  if (!ifs) return false;
  std::string line;
  apsp_.clear();

  while (std::getline(ifs, line))
  {
    apsp_.push_back(std::vector<int>());
    std::stringstream ss(line);
    while (1)
    {
      char c = ss.peek();
      if (c == ' ')
      {
        ss.read(&c, 2);
      }
      int id;
      if(!(ss >> id))
      {
        break;
      }
      apsp_.back().push_back(id);
    }
  }

  ifs.close();

  for (int i = 0; i < apsp_.size(); ++i)
  {
    for (int j = 0; j < apsp_[i].size(); ++j)
    {
```

```
      std::cout << apsp_[i][j] << "   ";
    }
    std::cout << std::endl;
  }
  return true;
}

bool Shortest_path::start_and_end_points_are_valid(const Point &start,
                                                   const Point &finish,
                                                   const Tug::Environment &environment)
{
  ClipperLib::IntPoint start_clipperlib(round(start.x()), round(start.y()));
  ClipperLib::IntPoint finish_clipperlib(round(finish.x()), round(finish.y()));

  for (int i = 1; i < environment.paths_.size(); ++i) //path_[0] is outer boundary
  {
    if (ClipperLib::PointInPolygon(start_clipperlib, environment.paths_[i])>0)
    {
      return false;
    }
    if (ClipperLib::PointInPolygon(finish_clipperlib, environment.paths_[i])>0)
    {
      return false;
    }
  }
  return true;
}

int Shortest_path::point_within_safety_margin(const Point &point,
                                              const Tug::Environment &environment)
```

```
{
  if (environment.paths_with_safety_margin_.size() == 0)
  {
    return 0;
  }
  ClipperLib::IntPoint point_clipperlib(round(point.x()), round(point.y()));
  for (int i = 1; i < environment.paths_with_safety_margin_.size(); ++i)
  {
    if (ClipperLib::PointInPolygon(point_clipperlib,
                          environment.paths_with_safety_margin_[i])>0)
    {
      return i;
    }
  }
  return 0;
}

Point Shortest_path::calculate_point_on_boundary(const Point &point,
                                                 const VisiLibity::Polygon &hole,
                                                 const Tug::Environment &env)
{
  Point current_shortest;
  Point current;
  double shortest_distance = std::numeric_limits<double>::max();

  if (!( env.point_is_on_outer_boundary(hole[hole.n()-1]) and
         env.point_is_on_outer_boundary(hole[0]) ))
  {
    VisiLibity::Line_Segment line(hole[hole.n()-1], hole[0], 0);
    current_shortest = point_closest_to_line_segment(point, line);
```

```
      shortest_distance = VisiLibity::distance(point, current_shortest);
    }

    for (int i = 0; i < hole.n()-1; ++i)
    {
      if (!( env.point_is_on_outer_boundary(hole[i]) and
             env.point_is_on_outer_boundary(hole[i+1]) ))
      {
        VisiLibity::Line_Segment line(hole[i], hole[i+1], 0);
        current = point_closest_to_line_segment(point, line);
        double dist = distance(point, current);
        if (dist < shortest_distance)
        {
          //std::cout << "thug life: " << dist << std::endl;
          current_shortest = current;
          shortest_distance = dist;
        }
      }

    }
   Point return_pt(current_shortest.x(), current_shortest.y(), env);

    return return_pt;
  }

  Point Shortest_path::point_closest_to_line_segment(const Point &point,
                                                     const VisiLibity::Line_Segment &line)
  {
    VisiLibity::Point pt = point.projection_onto(line);
    return Point(pt);
```

C.3.3 test

test.cpp

```
// tests.cpp
#include "geometry/tug_environment.hpp"
#include "tug_shortest_path.hpp"
#include "tug_route_around_ship.hpp"
#include <gtest/gtest.h>
#include <math.h>


//SHORTEST_PATH start
TEST(ShortestPathTest, NoSafetyMargin)
{
  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  Tug::Polyline shortest_path_test;

  Tug::Point start(30, 180, tug_environment);
  Tug::Point finish(190, 87, tug_environment);

  Tug::Shortest_path shortest_path_class(tug_environment, start, finish,
                                         shortest_path_test);

  VisiLibity::Polyline shortest_path_solution;
  shortest_path_solution.push_back(VisiLibity::Point(30,180));
  shortest_path_solution.push_back(VisiLibity::Point(120,190));
  shortest_path_solution.push_back(VisiLibity::Point(170,170));
  shortest_path_solution.push_back(VisiLibity::Point(190,87));
```

```
  ASSERT_EQ(shortest_path_test.size(), shortest_path_solution.size());

  for (int i = 0; i < shortest_path_solution.size(); ++i)
  {
    ASSERT_EQ(shortest_path_test[i].x(), shortest_path_solution[i].x());
    ASSERT_EQ(shortest_path_test[i].y(), shortest_path_solution[i].y());
  }
}

TEST(ShortestPathTest, StartAndFinishWithinSafetyMargin)
{
  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  tug_environment.add_constant_safety_margin(44);

  Tug::Polyline shortest_path_test;
  Tug::Point start(30, 180, tug_environment);
  Tug::Point finish(347, 230, tug_environment);

  Tug::Shortest_path shortest_path_class(tug_environment, start, finish,
                                         shortest_path_test);

  Tug::Polyline shortest_path_solution;
  shortest_path_solution.push_back(Tug::Point(30,180, tug_environment));
  shortest_path_solution.push_back(Tug::Point(26,180, tug_environment));
  shortest_path_solution.push_back(Tug::Point(26,200, tug_environment));
  shortest_path_solution.push_back(Tug::Point(120, 237, tug_environment));
  shortest_path_solution.push_back(Tug::Point(176, 216, tug_environment));
  shortest_path_solution.push_back(Tug::Point(347, 216, tug_environment));
```

```
  shortest_path_solution.push_back(Tug::Point(347, 230, tug_environment));

  ASSERT_EQ(shortest_path_test.size(), shortest_path_solution.size());

  for (int i = 0; i < shortest_path_solution.size(); ++i)
  {
    ASSERT_EQ(shortest_path_test[i].x(), shortest_path_solution[i].x());
    ASSERT_EQ(shortest_path_test[i].y(), shortest_path_solution[i].y());
  }

}

TEST(ShortestPathTest, NoValidPath)
{
  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  tug_environment.add_constant_safety_margin(54);

  Tug::Polyline shortest_path_test;
  Tug::Point start(30, 180, tug_environment);
  Tug::Point finish(347, 180, tug_environment);

  Tug::Shortest_path shortest_path_class(tug_environment, start, finish,
                                         shortest_path_test);

  ASSERT_EQ(shortest_path_test.size(), 0);
}

TEST(ShortestPathTest, TrivialCase)
{
```

```
  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  Tug::Polyline shortest_path_test;
  Tug::Point start(30, 180, tug_environment);
  Tug::Point finish(20, 180, tug_environment);

  Tug::Shortest_path shortest_path_class(tug_environment, start, finish,
                                         shortest_path_test);

  ASSERT_EQ(shortest_path_test[0].x(), start.x());
  ASSERT_EQ(shortest_path_test[0].y(), start.y());
  ASSERT_EQ(shortest_path_test[1].x(), finish.x());
  ASSERT_EQ(shortest_path_test[1].y(), finish.y());

}

TEST(ShortestPathTest, StartPointInsideObstacle)
{
  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  Tug::Polyline shortest_path_test;
  Tug::Point start(90, 150, tug_environment);
  Tug::Point finish(347, 180, tug_environment);

  Tug::Shortest_path shortest_path_class(tug_environment, start, finish,
                                         shortest_path_test);

  ASSERT_EQ(shortest_path_test.size(), 0);
}
```

```
TEST(ShortestPathTest, AllPairsShortestPathNoSafetyMargin)
{
  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  Tug::Polyline shortest_path_test;

  Tug::Point start(30, 180, tug_environment);
  Tug::Point finish(190, 87, tug_environment);

  Tug::Shortest_path shortest_path_class(tug_environment);
  shortest_path_class.calculate_shortest_path(start, finish,
                                              shortest_path_test,
                                              tug_environment);

  Tug::Polyline shortest_path_solution;
  shortest_path_solution.push_back(Tug::Point(30,180,tug_environment));
  shortest_path_solution.push_back(Tug::Point(120,190, tug_environment));
  shortest_path_solution.push_back(Tug::Point(170,170, tug_environment));
  shortest_path_solution.push_back(Tug::Point(190,87, tug_environment));

  ASSERT_LE(shortest_path_test.length(), shortest_path_solution.length());

}

TEST(ShortestPathTest, AllPairsShortestPathNoValidPath)
{
  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  tug_environment.add_constant_safety_margin(54);
```

```
  Tug::Polyline shortest_path_test;

  Tug::Point start(2, 7, tug_environment);
  Tug::Point finish(340, 3, tug_environment);

  Tug::Shortest_path shortest_path_class(tug_environment);

  bool ok = shortest_path_class.calculate_shortest_path(start, finish,
                                  shortest_path_test, tug_environment);

  ASSERT_EQ(shortest_path_test.size(), 0);
}
//SHORTEST_PATH end

TEST(RouteAroundShipTest, correctPlacement)
{

  Tug::Environment tug_environment("test_environment.txt", 1.0, 0.01);

  Tug::Point mid_pt(10, 10, tug_environment);
  Tug::Route_around_ship route_around_ship(M_PI/2, 2, 4);

  Tug::Point start(14, 10, tug_environment);
  Tug::Point finish(7, 9, tug_environment);

  Tug::Polyline solution = route_around_ship.best_route(start, finish, tug_environment);

  std::cout << "solution: ";
  for (int i = 0; i < solution.size(); ++i)
  {
```

```
    std::cout << solution[i] << std::endl;
  }
}

int main(int argc, char **argv)
{
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

C.4 Coordination

Contents

- tug_assign_paths.hpp
- tug_assign_paths.cpp

C.4.1 include

tug_assign_paths.hpp

```
#ifndef TUG_ASSIGN_PATHS
#define TUG_ASSIGN_PATHS

#include "external/munkres.h"
#include "geometry/tug_boat.hpp"
#include "geometry/tug_environment.hpp"
#include "search/tug_shortest_path.hpp"

namespace Tug
{
  class Assign_paths
  {
  public:
    Assign_paths(Environment environment)
    {
      shortest_path_node_ptr = std::make_shared<Shortest_path>(environment);
    }

    bool assign_on_combined_shortest_path(std::map<int, Boat> &tugs,
                                          const std::vector<Point> &poal_pts,
                                          Environment &environment);
    bool assign_on_combined_shortest_path(std::vector<Boat> &tugs,
                                          std::map<int, Point> &poal_pts,
                                          Environment &environment);
  private:
    double euclidean_distance(const Point &point1, const Point &point2);
```

```
    bool assign(std::vector<Boat> &tugs,
                const std::vector<Point> &poal_pts,
                Environment &environment);
    std::shared_ptr<Shortest_path> shortest_path_node_ptr;

  };
}

#endif //TUG_ASSIGN_PATHS
```

C.4.2 src

tug_assign_paths.cpp

```
/*
 *   Copyright (c) 2007 John Weaver
 *   Copyright (c) 2015 Miroslav Krajicek
 *   Copyright (c) 2017 Rebecca Cox
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#include "tug_assign_paths.hpp"
#include "ros/ros.h"
namespace Tug
{
//OBS: This function removes tugs from vector if they are not used.
  bool Assign_paths::assign(std::vector<Boat> &tugs,
```

```
                          const std::vector<Point> &goal_pts,
                          Environment &environment)
{
  int nrows = goal_pts.size();
  int ncols = tugs.size();

  if (nrows > ncols) {std::cout << "More goals than tugs" << std::endl;return false;}

  if (goal_pts.size() == 1 && tugs.size()>0)
  {
    Boat *closest = &tugs[0];
    double closest_distance = euclidean_distance(tugs[0].get_position(),
    goal_pts[0]);
    for (std::vector<Boat>::iterator tug = tugs.begin(); tug != tugs.end(); ++tug)
    {
      if (euclidean_distance(tug->get_position(), goal_pts[0]) < closest_distance)
      {
        closest = &*tug;
      }
    }

    Polyline sp;
    shortest_path_node_ptr->calculate_shortest_path(closest->get_position(),
                                                    goal_pts[0],
                                                    sp,
                                                    environment);
    closest->set_path(sp);
    return true;
  }
```

```
if (nrows < ncols)
{
  std::cout << "More tugs than goals. Choosing closest tugs" << std::endl;

  double sum_x =0; double sum_y=0;
  for (int i = 0; i < ncols; ++i)
  {
    sum_x += goal_pts[i].x();
    sum_y += goal_pts[i].y();
  }
  Point mid_point(sum_x/ncols, sum_y/ncols);

  double distances[nrows];

  for (int i = 0; i < nrows; ++i)
  {
    distances[i] = euclidean_distance(tugs[i].get_position(), mid_point);
  }

  int n_tugs_to_remove = ncols-nrows;
  std::vector<int> removable;

  for(int i = 0; i < n_tugs_to_remove; i++)
  {
    double *max_dist = std::max_element(distances, distances+nrows);
    int max_index = std::distance(distances, max_dist);
    distances[max_index] = 0;
    removable.push_back(max_index);
  }
  for (int i = removable.size()-1; i>=0; --i)
```

```
  {
    tugs.erase(tugs.begin()+removable[i]);
  }
  ncols = nrows;
}

Matrix<double> cost_mat(nrows, ncols);

Polyline sp_temp;
for (int i = 0; i < nrows; ++i)
{
  for (int j = 0; j < ncols; ++j)
  {
    shortest_path_node_ptr->calculate_shortest_path(tugs[i].get_position(),
                                                    goal_pts[j],
                                                    sp_temp,
                                                    environment);
    cost_mat(i,j) = sp_temp.length();
  }
}

Munkres<double> m;
m.solve(cost_mat);

for ( int row = 0 ; row < nrows ; row++ )
{
  int rowcount = 0;
  for ( int col = 0 ; col < ncols ; col++ )
  {
    if ( cost_mat(row,col) == 0 )
```

```
      rowcount++;
    }
    if ( rowcount != 1 )
    {
      std::cerr << "Row " << row << " has " << rowcount <<
      " columns that have been matched." << std::endl;
      return false;
    }
  }

  for ( int col = 0 ; col < ncols ; col++ )
  {
    int colcount = 0;
    for ( int row = 0 ; row < nrows ; row++ )
    {
      if ( cost_mat(row,col) == 0 )
      {
        colcount++;
      }
    }

    if ( colcount != 1 )
    {
      std::cerr << "Column " << col << " has " << colcount <<
      " rows that have been matched." << std::endl;
      return false;
    }
  }
    // Display solved matrix.
  for ( int row = 0 ; row < nrows ; row++ )
```

```
    {
      bool tug_planned = false;
      for ( int col = 0 ; col < ncols ; col++ )
      {
        std::cout.width(2);
        std::cout << cost_mat(row,col) << ",";
        if (cost_mat(row,col) == 0)
        {
          Polyline path;
          shortest_path_node_ptr->calculate_shortest_path(tugs[row].get_position(),
                                                          goal_pts[col],
                                                          path,
                                                          environment);

          tugs[row].set_path(path);
        }
      }
      std::cout << std::endl;
    }
    std::cout << std::endl;

    return true;
  }

  bool Assign_paths::assign_on_combined_shortest_path(std::map<int, Boat> &tugs,
                                                      const std::vector<Point> &goal_pts,
                                                      Environment &environment)
  {
    std::vector<Boat> temp_tugs;
    for (std::map<int, Boat>::iterator i = tugs.begin(); i != tugs.end(); ++i)
    {
```

```
      temp_tugs.push_back(i->second);
    }
    bool ok = assign(temp_tugs, goal_pts, environment);

    for (std::vector<Boat>::iterator i = temp_tugs.begin(); i != temp_tugs.end(); ++i)
    {
     tugs[i->id()].set_path(i->get_path());
    }

    return ok;
  }

  bool Assign_paths::assign_on_combined_shortest_path(std::vector<Boat> &tugs,
                                                      std::map<int, Point> &goal_pts,
                                                      Environment &environment)
  {
    std::vector<Point> end_points;
    for (std::map<int,Point>::iterator i = goal_pts.begin(); i != goal_pts.end(); ++i)
    {
      end_points.push_back(i->second);
    }
    assign(tugs, end_points, environment);
  }

  double Assign_paths::euclidean_distance(const Point &point1, const Point &point2)
  {
    return sqrt(pow(point1.x() - point2.x(), 2) + pow(point1.y() - point2.y(), 2));
  }
}
```

C.5 Tug Gazebo

Contents

- goal_point_generator.cpp
- tug_location_publisher.cpp
- tug_startup_publisher.cpp
- tug_state_publisher.cpp
- waypTugs_publisher.cpp

C.5.1 src

goal_point_generator.cpp

```
#include "geometry/tug_environment.hpp"
#include "tugboat_control/ClearWaypoint.h"
#include "tugboat_control/Waypoint.h"

#include <ros/package.h>
#include <ros/ros.h>
#include <string>

namespace
{
  std::map<int, tugboat_control::Waypoint> waypoints;
}

void callback_clear_waypoint(const tugboat_control::ClearWaypoint::ConstPtr& msg)
{
  try
  {
    waypoints.erase(msg->orderID);
  }
  catch(const std::out_of_range &oor){}
}


int main(int argc, char **argv)
{
```

```
ros::init(argc, argv, "goal_points_generator");
ros::NodeHandle n;

ros::Publisher pub_goal = n.advertise<tugboat_control::Waypoint>("waypointRequest", 20);
ros::Subscriber sub_arrived = n.subscribe("clearWaypoint", 20, callback_clear_waypoint);

tugboat_control::Waypoint pt1;
pt1.ID = 11;
pt1.x = 2;
pt1.y = 1.625;

tugboat_control::Waypoint pt2;
pt2.ID = 22;
pt2.x = 2.5;
pt2.y = 1.75;

tugboat_control::Waypoint pt3;
pt3.ID = 33;
pt3.x = 2;
pt3.y = 1.25;

tugboat_control::Waypoint pt4;
pt4.ID = 44;
pt4.x = 2.375;
pt4.y = 1.125;

waypoints.insert(std::pair<int, tugboat_control::Waypoint>(pt1.ID, pt1));
waypoints.insert(std::pair<int, tugboat_control::Waypoint>(pt2.ID, pt2));
waypoints.insert(std::pair<int, tugboat_control::Waypoint>(pt3.ID, pt3));
waypoints.insert(std::pair<int, tugboat_control::Waypoint>(pt4.ID, pt4));
```

```
  ros::Rate loop_rate(2);
  while (ros::ok())
  {
    for (std::map<int, tugboat_control::Waypoint>::iterator pt = waypoints.begin();
                                                            pt != waypoints.end();
                                                            ++pt)
    {
      pub_goal.publish(pt->second);
    }
    loop_rate.sleep();
    ros::spinOnce();
  }


  ros::spin();
  return 0;
}
```

tug_location_publisher.cpp

```
#include "master/tug_constants.hpp"
#include "tugboat_control/BoatPose.h"
#include "tugboat_control/Waypoint.h"

#include <gazebo_msgs/GetModelState.h>
#include <ros/package.h>
#include <ros/ros.h>
#include <string>

ros::ServiceClient pose_srv;
ros::Publisher pub;
std::vector<std::string> tug_names_;
std::map<int,std::string> tugs_;

void update_tug_pose()
{
  gazebo_msgs::GetModelState getmodelstate;

  for (std::map<int, std::string>::iterator tug = tugs_.begin();
                                            tug != tugs_.end();
                                            ++tug)
  {
    getmodelstate.request.model_name = tug->second.c_str();
    getmodelstate.request.relative_entity_name = "world";
    try
    {
      pose_srv.call(getmodelstate);
      tugboat_control::BoatPose pose_msg;
```

```
      pose_msg.ID = tug->first;
      pose_msg.x = getmodelstate.response.pose.position.x/SCALE;
      pose_msg.y = getmodelstate.response.pose.position.y/SCALE;
      pub.publish(pose_msg);
    }
    catch(...)
    {
      ROS_ERROR("Cannot find pose info about %s", tug->second.c_str());
    }
  }
}


int main(int argc, char **argv)
{
  ros::init(argc, argv, "tug_location_publisher");
  int i = 1;
  while (i < argc)
  {
    std::string name = argv[i];
    int tug_id = *argv[++i] - '0';
    ROS_INFO("Tug %s with id %d, ready to publish pose", name.c_str(), tug_id);
    tugs_.insert(std::pair<int, std::string>(tug_id, name));
    i++;
  }

  ros::NodeHandle node;
  pub = node.advertise<tugboat_control::BoatPose>("pose", 20);
  pose_srv = node.serviceClient<gazebo_msgs::GetModelState>("/gazebo/get_model_state");
```

```
  ros::Rate loop_rate(12);
  while (ros::ok())
  {
    update_tug_pose();
    loop_rate.sleep();
  }
  return 0;
}
```

tug_startup_publisher.cpp

```
#include "ros/package.h"
#include <ros/ros.h>
#include "std_msgs/UInt8.h"


int main(int argc, char **argv)
{
  ros::init(argc, argv, "tug_startup_publisher");
  ros::NodeHandle n;

  ros::Publisher pub_goal = n.advertise<std_msgs::UInt8>("startup", 20);

  std_msgs::UInt8 msg1;
  msg1.data = 1;
  std_msgs::UInt8 msg2;
  msg2.data = 2;
  std_msgs::UInt8 msg3;
  msg3.data = 3;
    std_msgs::UInt8 msg4;
  msg4.data = 4;
    std_msgs::UInt8 msg5;
  msg5.data = 5;
    std_msgs::UInt8 msg6;
  msg6.data = 6;

  ros::Rate loop_rate(0.5);
  while (ros::ok())
  {
```

```
    pub_goal.publish(msg1);
    pub_goal.publish(msg2);
    pub_goal.publish(msg3);
    pub_goal.publish(msg4);
    pub_goal.publish(msg5);
    pub_goal.publish(msg6);

    loop_rate.sleep();
    ros::spinOnce();
  }


  return 0;
}
```

tug_state_publisher.cpp

```
#include "tugboat_control/Waypoint.h"

#include <gazebo_msgs/GetModelState.h>
#include <gazebo_msgs/ModelState.h>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/Vector3.h>
#include <math.h>
#include <ros/ros.h>
#include <sstream>

ros::Publisher state_pub;
geometry_msgs::Pose tug_pose;
ros::ServiceClient pose_srv;
std::string tug_name;
int tug_id;
geometry_msgs::Pose current_goal;

double get_distance(geometry_msgs::Pose p1, geometry_msgs::Pose p2)
{
  return sqrt(pow((p2.position.x-p1.position.x),2) + pow((p2.position.y-p1.position.y),2));
}

void update_tug_pose()
{
  gazebo_msgs::GetModelState getmodelstate;

  getmodelstate.request.model_name = tug_name;
```

```
  pose_srv.call(getmodelstate);

  tug_pose.position.x = getmodelstate.response.pose.position.x;
  tug_pose.position.y = getmodelstate.response.pose.position.y;
}

void stop_moving()
{
  geometry_msgs::Twist vel_msg;
  gazebo_msgs::ModelState state_msg;
  state_msg.model_name = tug_name;
  state_msg.reference_frame = "world";

  vel_msg.linear.x = 0;
  vel_msg.linear.y = 0;
  update_tug_pose();
  state_msg.twist = vel_msg;
  state_msg.pose = tug_pose;
  state_pub.publish(state_msg);
}

void callback_waypoint(const tugboat_control::Waypoint::ConstPtr &msg)
{
  if (msg->ID == tug_id)
  {
    geometry_msgs::Twist vel_msg;
    gazebo_msgs::ModelState state_msg;
    state_msg.model_name = tug_name;
    state_msg.reference_frame = "world";
    double speed = msg->v;
```

```
    double distance_tolerance = 0.5;

    current_goal.position.x = msg->x;
    current_goal.position.y = msg->y;

    update_tug_pose();

    ros::Rate loop_rate(4);

    while(get_distance(tug_pose, current_goal) > distance_tolerance)
    {
      double speed_x = (current_goal.position.x - tug_pose.position.x);
      double speed_y = (current_goal.position.y - tug_pose.position.y);
      double norm_const = sqrt(pow(speed_x,2)+pow(speed_y,2));

      vel_msg.linear.x = speed*speed_x/norm_const;
      vel_msg.linear.y = speed*speed_y/norm_const;
      vel_msg.linear.z = 0;

      state_msg.twist = vel_msg;
      state_msg.pose = tug_pose;

      state_pub.publish(state_msg);
      ros::spinOnce(); //check if current_goal is updated
      loop_rate.sleep();
      ros::spinOnce(); //check if current_goal is updated
      update_tug_pose();
    }

    stop_moving();
```

```
  }
}


int main(int argc, char** argv){
  ros::init(argc, argv, "tug_state_publisher");

  ros::NodeHandle node;
  if (argc != 3){ROS_ERROR("need tug name and ID as argument"); return -1;};
  tug_name = argv[1];

  try
  {
    tug_id = *argv[2] - '0';
  }
  catch(...)
  {
    ROS_ERROR("need tug ID argument to be an integer"); return -1;
  }

  //current_goal.position.x=0; current_goal.position.y=0;

        srand(time(0));

  pose_srv = node.serviceClient<gazebo_msgs::GetModelState>("/gazebo/get_model_state");
 // update_tug_pose();
 // sleep(5);
  ros::Subscriber sub_goal = node.subscribe<tugboat_control::Waypoint>("waypoint", 20, callback_waypoint);

  state_pub = node.advertise<gazebo_msgs::ModelState>("/gazebo/set_model_state", 100);
```

```
  ros::spin();

  return 0;
};
```

waypTugs_publisher.cpp

```
#include "tugboat_control/ClearWaypoint.h"

#include <ros/package.h>
#include <ros/ros.h>
#include <std_msgs/UInt8MultiArray.h>

std_msgs::UInt8MultiArray waypTugs_msg;


void callback_clearWaypoint(const tugboat_control::ClearWaypoint::ConstPtr &msg)
{
  int id = msg->tugID;
  std_msgs::UInt8MultiArray temp = waypTugs_msg;
  waypTugs_msg.data.clear();
  for (int i = 0; i < temp.data.size(); ++i)
  {
    if (temp.data[i] != id)
    {
      waypTugs_msg.data.push_back(temp.data[i]);
    }
  }
}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "waypTugs");
```

```
  ros::NodeHandle n;
  ros::Publisher pub = n.advertise<std_msgs::UInt8MultiArray>("waypTugs", 20);

  ros::Subscriber sub = n.subscribe("clearWaypoint", 20, callback_clearWaypoint);

  waypTugs_msg.data.push_back(1);
  waypTugs_msg.data.push_back(2);
  waypTugs_msg.data.push_back(3);
  waypTugs_msg.data.push_back(4);
  waypTugs_msg.data.push_back(5);
  waypTugs_msg.data.push_back(6);


  ros::Rate loop_rate(1);
  while (ros::ok())
  {
    pub.publish(waypTugs_msg);
    loop_rate.sleep();
    ros::spinOnce();
  }

  ros::spin();
  return 0;
}
```