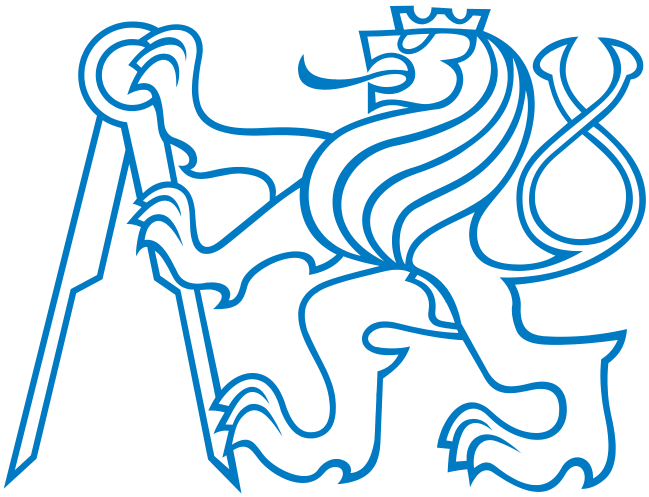
Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Cybernetics



# BACHELOR THESIS

## Submarine Behaviour Model for Monte Carlo Simulations

Tomáš Dlask

Supervisor: Ing. Ondřej Hrstka

May 2016

**Author statement**

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 16th May 2016 .....

signature

**Czech Technical University in Prague**
**Faculty of Electrical Engineering**

**Department of Cybernetics**

# BACHELOR PROJECT ASSIGNMENT

**Student:** Tomáš D l a s k

**Study programme:** Open Informatics

**Specialisation:** Computer and Information Science

**Title of Bachelor Project:** Submarine Behaviour Model for Monte Carlo Simulations

**Guidelines:**

1. Study the submarine and anti-submarine warfare domain.
2. Create environment and risk aware submarine trajectory planner.
3. Develop and integrate submarine behaviour model into Bandit simulator.
4. Create user interface for simulation.

**Bibliography/Sources:**

[1] Mishra, Mahesh K., et al. "Decision support software for Anti-Submarine warfare mission planning within a dynamic environmental context." Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on. IEEE, 2014.
[2] Vaněk, Ondřej, et al. "Agent-based model of maritime traffic in piracy-affected waters." Transportation research part C: emerging technologies 36 (2013): 157-176.
[3] Hrstka, Ondřej, et al. "BANDIT (Behavioral Agents for Drug Interdiction) - Phase 1 Report"

**Bachelor Project Supervisor:** Ing. Ondřej Hrstka

**Valid until:** the end of the summer semester of academic year 2016/2017

L.S.

prof. Dr. Ing. Jan Kybic
**Head of Department**

prof. Ing. Pavel Ripka, CSc.
**Dean**

Prague, January 11, 2016

*Abstrakt*

Cílem této práce je vytvoření plánovače tras pro ponorky, který umí zvažovat riziko zvolené cesty a je schopen fungovat v prostředí, které se mění v čase. K dosáhnutí tohoto cíle jsme vlastní metodou diskretizovali stavový prostor (v tomto případě oceán) na jednotlivé buňky, v rámci kterých jsou vlastnosti prostředí homogenní. Abychom byli schopni počítat se změnami v prostředí, navrhli jsme vzorkovací metodu Gradually Extending RRT* na základě algoritmu RRT*. U této metody jsme prokázali, že je nejlepší z několika zvažovaných alternativ. GERRT* jsme testovali v několika prostředích s diskrétním časovým paradigmatem a pokaždé jsme obdrželi uspokojivé výsledky. V neposlední řadě jsme vytvořili model chování ponorky, který je zasazen do simulačního prostředí BANDIT. Součástí práce je také implementace navržených algoritmů v jazyce Java.

Klíčová slova: plánování, RRT*, grid, simulace, vzorkování

*Abstract*

The aim of this work is to develop a risk-aware planning method for submarines functioning in dynamic environments with time-variant conditions. To achieve this goal, a gridding method is introduced that divides a state space (in this case an ocean) into a set of cells with locally homogeneous conditions. To capture the changes in the environment, the Gradually Extending RRT* algorithm is introduced as a modification of RRT*. This new sampling-based planning method was chosen as the best from multiple considered alternatives. The GERRT* provided fair results in the tested scenarios with discrete time paradigm. A behaviour model of a submarine driven by the algorithm was also created to be employed in the BANDIT simulation interface. The implementation of the proposed methods in the Java programming language is enclosed.

Keywords: planning, RRT*, grid, simulation, sampling

*Acknowledgements*

I would like to thank to my supervisor Ing. Ondřej Hrstka for his guidance and useful advice concerning the thesis. I also wish to express my sincere gratitude to my family and friends for their unceasing support throughout my studies.

# Contents

# List of Figures

# List of Tables

# Used notation

| Symbol | Meaning |
|---:|---|
| $a, A, \ldots z, Z, \epsilon, \Delta$ | real numbers, unless stated otherwise |
| $\lvert a \rvert$ | the absolute value of $a$ |
| $\mathbb{R}$ | the set of real numbers |
| $\mathbb{R}^D$ | the set of $D$-dimensional real vectors |
| $\mathbb{R}^{\geq 0}$ | the set of non-negative real numbers |
| $[a, b]$ | closed interval of real numbers, includes both $a$ and $b$ |
| $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots, \mathcal{Z}$ | sets |
| $\mathcal{A} \subset \mathcal{B}$ | $\mathcal{A}$ is a subset of $\mathcal{B}$ |
| $\mathcal{A} \times \mathcal{B}$ | Cartesian product of sets $\mathcal{A}$ and $\mathcal{B}$ |
| $\mathcal{A} - \mathcal{B}$ | a set of all elements from $\mathcal{A}$ that are not in $\mathcal{B}$ |
| $2^{\mathcal{A}}$ | system of all subsets of $\mathcal{A}$ |
| f: $\mathcal{A} \mapsto \mathcal{B}$ | the function f has domain $\mathcal{A}$ and codomain $\mathcal{B}$ |
| $\exp(a)$ | $e^a$, where $e$ is the Euler's number |
| $a \propto b$ | the value of $a$ is directly proportional to $b$ |
| $\mathbf{a}, \mathbf{A}, \ldots, \mathbf{z}, \mathbf{Z}, \mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3$ | vectors |
| $\mathbf{a}_i$ | $i$-th component of vector $\mathbf{a}$ |
| $\mathbf{a} \times \mathbf{b}$ | cross product of vectors $\mathbf{a}$ and $\mathbf{b}$ |
| $\mathbf{a} \cdot \mathbf{b}$ | dot product of vectors $\mathbf{a}$ and $\mathbf{b}$ |
| $\lVert \mathbf{a} \rVert_2$ | Euclidean norm of the vector $\mathbf{a}$ |
| $\mathsf{A}, \mathsf{B}, \ldots, \mathsf{Z}$ | graphs |

# Used abbreviations

| Abbreviation | Full name |
|---|---|
| ABC | Artificial Bee Colony |
| ACTUV | ASW Continuous Trailing Unmanned Vessel |
| ASW | Antisubmarine Warfare |
| BANDIT | Behavioral Agents for Drug Interdiction |
| BGERRT* | Balanced Gradually Extending RRT* |
| BRRT*TD | Balanced RRT* with Time as Dimension |
| BSM | Behaviour State Machine |
| DARPA | Defense Advanced Research Projects Agency |
| FSM | Finite-state Machine |
| GE | Gradually Extending (refers to both GERRT* and BGERRT*) |
| GERRT* | Gradually Extending RRT* (unbalanced) |
| GPS | Global Positioning System |
| hFSM | Hierarchical Finite State Machine |
| LOFAR | Low Frequency Array |
| OSCAR | Ocean Surface Current Analyses Real-time |
| USS | United States Ship |
| USV | Unmanned Surface Vessel |
| UUV | Unmanned Underwater Vehicle |
| PRM | Probabilistic Roadmap |
| RD | Risk of Detection |
| RRT | Rapidly-Exploring Random Tree |
| RRT*TD | RRT* with Time as Dimension (unbalanced) |
| Sonar | Sound Navigation and Ranging |
| SOSUS | Sound Surveillance System |
| SURTASS | Surveillance Towed Array Sensor System |
| TD | Time as Dimension (refers to both RRT*TD and BRRT*TD) |
| TW | Time Window |

# Thesis overview

Detection of submarines has become a challenging task in the military branch during the last decades due to the constantly improving stealth technologies. On the other hand, the ways of detection are getting broader with the increase in computational power that is able to process fuzzy information received from detectors with limited abilities. Monitoring vast areas of sea is an expensive procedure and that is why we would like to predict areas where a submarine might be located in order to focus the available assets there. In this thesis we are thus dealing with path planning under this specific domain.

First, we discuss the development of the roles of submarines until the present-day and also debate on their capabilities in the Chapter 1. Additionally, both the traditional and the state-of-the-art methods of detection are listed with a brief commentary on their functioning and efficacy.

Next, four sampling-based methods of path planning in static environment are introduced in the Chapter 2. We mention Rapidly-Exploring Random Trees and the asymptotically optimal improvement of this method, RRT*, as invented by Karaman et al. (2011). After that, we describe an algorithm solving a similar task - path planning for a submarine that wants to avoid detection. Li et al. (2014) uses the Artificial Bee Colony algorithm to plan the path. The fourth introduced method is the Probabilistic Roadmap planning. Lastly, a concise summary of the simulation parameters and characteristics is stated.

In the Chapter 3, a gridding method that produces sampling of the ocean both horizontally and vertically is presented along with the characteristics of the cells that make up the grid. Then, the task and constraints are formalized by introducing a cost function on the basis of the previously defined grid. Eventually, we bring forward four alternatives of the modified RRT* algorithm that are capable of planning in a dynamic environment.

Chapter 4 deals with the actual implementation of the BANDIT simulation framework and the integration of the submarine model. The many inputs of the gridding method and path planning process are listed in this chapter with a brief discussion on their meaning, importance and recommended values. Lastly, the formats of the output are shown.

The performance of the previously presented algorithms is compared in the Chapter 5. The methods are confronted in the terms of quality of the solution, duration of computation and stability of the results. The time complexity of the best performing method is then further analysed and we test it on few scenarios.

# Chapter 1

# Domain background

In this chapter, the purpose and the functions of submarines are described in 1.1, followed by the appraisal of their abilities in 1.2. Then, both usual and uncommon methods of detection are explained in 1.3 and their importance is discussed in the course of history with possible predictions for future development in 1.3.1. The general source for this chapter is mainly Clark (2015).

## 1.1 Objectives

Submarines have played an important role in warfare since World War I. Their main purpose in wartime was to sink the enemy ships, especially targeting convoys in the Second World War. Later, submarines became capable of attacking each other too. Except that, their roles nowadays also include reconnaissance, surveillance, performing blockades, deployment of special forces, or protection of other ships. Clark (2015) estimates that in the future, submarines will not be used as direct participants, but will only serve as mother ships deploying unmanned underwater vessels (UUVs) that would perform assignments in hostile waters and could replace the manned vessels to some extent.

Outside of the military branch, submarines can also be used by civilians in oceanography or tourism. In addition, so-called narco submarines are used by smugglers to transport drugs.

## 1.2 Capabilities

In the first half of the 20th century, the diesel electric submarines were only capable of staying 1-2 days below surface. For instance, in the operation Sandblast, USS Triton circumnavigated the globe in 83 days, but had to ventilate and specify its position with an sextant every day, according to Lundquist (2013). Because the whole hull did not need to surface, the voyage of USS Triton is considered to be the first vessel to accomplish a round-the-world tour submerged. The position of the submarine during the day was estimated using dead reckoning navigation; that means by calculating the direction and speed of movement and also taking the influence of currents into account.

Figure 1.1: French submarine Téméraire, picture taken from Alabordache (2005).

With the development of nuclear power and the ability to extract oxygen from seawater, state-of-the-art submarines are capable of staying submerged almost indefinitely. However, there are still reasons for surfacing, for example limited food supplies for the crew or navigation. Speaking about navigation, GPS does not receive signal underwater, therefore accelerometers are used in submarines to estimate their position. But they exhibit inaccurate behaviour and after one day of voyage, the difference of calculated and actual position may grow up to 1 kilometre. This issue could be solved by a newly introduced quantum positioning system that is approximately 1000 times more accurate than currently used accelerometers. After testing, the system could not only be used in submarines, but also in other environments where GPS signal can not be received or would be critical to lose, according to Marks (2014).

The maximal speed of modern submarines is usually over 30 knots (55 kilometres per hour) and increases when the submarine dives due to the fact that its shape is optimized to reduce the drag in the submerged state. State-of-the-art submarines have test depth in the range between 200 and 400 metres.

## 1.3 Detection

As explained in Wren and May (1997), the detection methods can be divided into acoustic and non-acoustic. The most known acoustic methods are passive and active sonar; with the active sonar, a vessel or buoy emits a signal and measures whether it will hear an echo. The passive sonar only listens to the sources of noise in its surroundings.

Magnetic anomaly detection is an example of a non-acoustic method. It detects the changes in magnetic field caused by the hull of a submarine. Next, the movement of a vessel creates waves on the surface, such as the Bernoulli hump, Kelvin waves or internal waves. These phenomena are futher described in Wren and May (1997) and also represent the non-acoustic methods of detection. As also mentined in Jenkins (2012), the turbulences that the movement generates may cause the bioluminescent living organisms in sea to emit light and indirectly disclose the position of a boat.

Additionally, the whereabouts of a nuclear submarine can be revealed due to the heat that is generated by its reactor. It produces warmer water that rises to the surface and could be uncovered by infra red sensors. Similarly, a diesel submarine can be detected by its hot exhaust fumes if the vessel is snorkeling[1].

### 1.3.1 Historical development and views to the future

In both World Wars, the main methods of antisubmarine warfare (ASW) were visual detection, radar detection, locating radio transmission, or de-cyphering its content. The passive sonar could not successfully detect submerged vessels that quietly ran using their electric motors. However, this changed with the nuclear submarines that produced more noise from their machinery. Therefore, in the second half of the 20th century, the Sound Surveillance System (SOSUS) was created by the United States to detect Soviet submarines. The system was implemented on the east coast of the Northern America and the western coast of Europe. Listening posts were also located in the GIUK[2] gap in order to be aware of any Soviet submarines entering the Atlantic ocean.

The system showed its abilities for example in the project Azorian in the late 1960s, when the records of SOSUS and LOFAR (Low Frequency Array) detected an implosion and estimated its location to places where the Soviets conducted search activities, as mentioned in Aid et al. (2010). After the Soviet's search activities decreased, USS Halibut was able to find a wreck of a Soviet submarine in the predicted area in three weeks.

Similar system can also be implemented as a mobile device - for instance as in Surveillance Towed Array Sensor System (SURTASS) or Sonar 2087. According to Department of the Navy; Defence Supplier Directory, in these systems, a ship uses its active sonar and also tows an array of passive sensors that are able to either measure the reflection from a submarine located below the ship or function entirely passively.

Nowadays, the newest submarines are quiet and leave only little trace, causing the passive sonar to lose its efficiency. An incident from 2009 may support this claim; a British and a French submarine rammed each other, according to Burns (2009). Due to being armed with ballistic missiles and patrolling, active sonar was not used in neither of the submarines. In addition, both boats had their hulls covered in anechoic tile that eliminated the chances of mutual detection.

However, with the increase in computational power, even the small changes in the environment that the vessels leave can reveal them when the big data approach is used in combination with various sensors. This could for example be fuzzy information obtained from a sonar or even the ripples on the ocean surface that were left by a submarine passing underneath, as mentioned in Freedberg (2015). In the future, it is likely that extensive low-frequency active sonar arrays might be an important element in ASW. Steve Walker, DARPA deputy director, claimed in Magnuson (2016); Gady (2016) that the military may focus more on diversification of its assets and not rely mainly on individual submarines.

[1] A diesel submarine is snorkeling (also known as snorting) if the only part that is above surface is a tube for air intake and exhaust.

[2] Greenland - Iceland - United Kingdom

Vincent (2016) informs that an ASW Continuous Trail Unmanned Vessel (ACTUV) was christened in April 2016 and its main purpose is tracking the enemy submarines. The project also produced some civilian uses, for example pushing the quality of autonomous behaviour of vessels further.

# Chapter 2

# Related work

In this chapter, four stochastic methods are introduced, they serve as planning methods that can be used in environments with obstacles. We start with the RRT in the Section 2.1.1 and its improvement RRT* in 2.1.2. Next, the Artificial Bee Colony algorithm is introduced in 2.1.3 and the last mentioned method is the Probabilistic Roadmap planning in 2.1.4. Eventually, an introduction to the field of simulations is presented in 2.2.

## 2.1 Planning algorithms

In this context, the task of a planning algorithm is to find a usually optimal trajectory from a start point to a target. We could not utilize A* (described in Hart et al. (1968)) or another deterministic algorithm that finds the optimal path, because it requires substantial amount of cost function computations. That would considerably slow the algorithm down in our case due to the computationally demanding cost function evaluation. Additionally, we would like to use a stochastic planner that provides non-deterministic results in order to use them as an input for Monte Carlo simulations. Both these conditions are satisfied by the presented algorithms, RRT, RRT* and PRM, that are based on random sampling, do not require to determine the cost function too many times and are usable in randomized simulations.

### 2.1.1 Rapidly-Exploring Random Trees

RRT, as introduced in LaValle (1998) and further elaborated in LaValle and Kuffner (2001), is a sampling-based planning algorithm that addresses kinodynamic or non-holonomic constraints in a multi-dimensional space. Since the planning in our simulations will be done from a distant point of view on the scale of kilometres, not considering individual steering and accelerating abilities of vessels, there will not be any dynamic constraints. The only constraints will be limited speed and maximal time that a vessel is able to spend below surface, these are discussed in 3.3.1.

In the Algorithm 1, a tree is built in a given state space $\mathcal{S}$ starting with the initial vertex $p_{\text{init}} \in \mathcal{S}$ as its root. Then, the iterative process of extending the tree is applied.

First, the function get_random_state() is called. It returns a random vertex $p_{\text{random}}$ from the state space uniformly. Second, the function find_nearest($p_{\text{random}}$) is used to find the vertex $p_{\text{near}}$ in the tree such that it is the closest one to the $p_{\text{random}}$. After that, the function steer($p_{\text{near}}$, $p_{\text{random}}$, $\epsilon$) returns a vertex $p_{\text{new}}$ that is obtained by movement from $p_{\text{near}}$ in direction towards $p_{\text{random}}$ for given distance or time period $\epsilon$. Finally, if the edge between $p_{\text{new}}$ and $p_{\text{near}}$ is feasible and does not violate any constraints, the vertex $p_{\text{new}}$ is added into the tree using add_vertex($p_{\text{new}}$) function and is set as a follower of $p_{\text{near}}$ by set_parent($p_{\text{new}}$,$p_{\text{near}}$). The whole process is repeated until the tree $\mathsf{T}$ has enough vertices. Other stopping conditions can also be used, for example finding a solution or exceeding the computational time.

**Algorithm 1**: RRT($N$,$p_{\text{init}}$,$\epsilon$)

```
T ← empty_tree();
T.add_vertex(p_init);
while T has less than N vertices do
    p_random ← get_random_state();
    p_near ← T.find_nearest(p_random);
    p_new ← steer(p_near, p_random, ϵ);
    if edge between p_new and p_near is feasible then
        T.add_vertex(p_new);
        T.set_parent(p_new,p_near);
    end
end
return T;
```

The result of RRT is a feasible sequence of states from $p_{\text{init}}$ to given $p_{\text{goal}}$. The algorithm might add the $p_{\text{goal}}$ into the tree during one of its iterations and thus create a feasible result. That does not need to necessarily happen, yet a solution can be created by finding a vertex in the tree that is nearby $p_{\text{goal}}$ and the edge between them is feasible.

The algorithm is biased towards exploring the parts of space that were not visited yet. The reason for this behaviour can be explained simply, but first, the term Voronoi cell needs to be defined.

Given set of points $\mathcal{P} = \{p_1, ..., p_n\}$ in a state space $\mathcal{S}$, $\mathcal{P} \subset \mathcal{S}$ and a metric d: $\mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}^{\geq 0}$, the Voronoi cell belonging to the point $p_i$ is a set

$$\mathcal{V}_i = \{p \in \mathcal{S} \,|\, \mathrm{d}(p, p_i) \leq \mathrm{d}(p, p_j)\, \forall j \in \{1, ..., n\}\}. \tag{2.1}$$

Now, we can say that the unexplored parts of the space are usually large Voronoi cells. Therefore, the probability of choosing a point from the large cells is greater than its occurrence in the smaller cells, under the assumption of uniform sampling.

To better understand the new term, a state space in the form of a square is divided by RRT with various amount of sampled nodes in the Figure 2.1. In the upper sub-figures, the trees that represent the current state of exploration are depicted. The divisions of the space into Voronoi cells corresponding to the trees are shown in the bottom part of the figure.

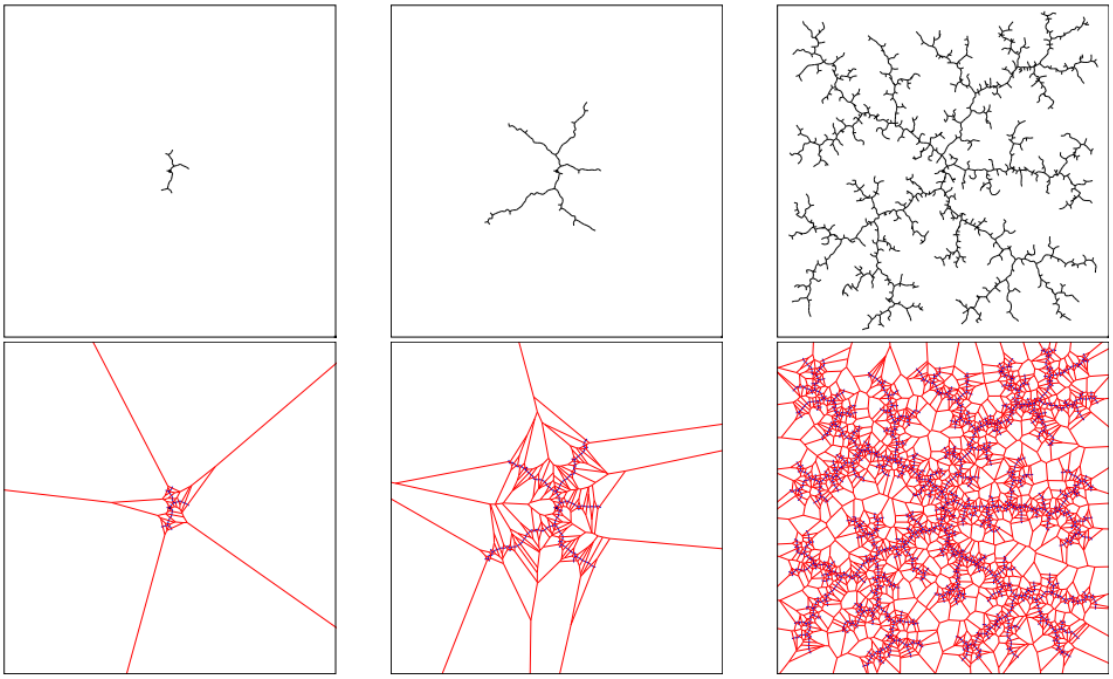


Figure 2.1: Division of a state space into Voronoii cells by RRT, taken from LaValle and Kuffner (2001).

Now, it can be stated that the areas in the state space that were not sampled yet will pose as large Voronoi cells. Thus, in random sampling, the probability of placing a new node into these areas will be larger than into those that have larger density of already sampled points. This will lead to the division of the large Voronoi cells and exploration of new states. This is why RRT is sometimes denoted as a Monte Carlo method.

Another characteristics of this algorithm is its probabilistic completeness, i.e. the probability of finding a feasible solution approaches one as the number of sampled vertices approaches infinity if a feasible solution exists.

However, the optimality of the solution is not guaranteed and the algorithm will most likely find non-optimal solution, as mentioned in Karaman and Frazzoli (2010).

### 2.1.2 RRT*

According to Karaman et al. (2011), RRT* maintains the tree structure of RRT, but unlike RRT, it also iteratively optimizes the tree. That results in asymptotic optimality of the algorithm as its main advantage over RRT.

In RRT*, it is additionally necessary to define a cost function that will assign non-negative values to sequences of vertices from the state space $S$. The cost function as introduced in 3.2.3 can not be used directly, because RRT* does not handle changing environment in its basic version. Thus, to introduce this algorithm, a simpler version of cost function is defined as

$$\text{cost} : \mathcal{S} \times \mathcal{S} \times \cdots \times \mathcal{S} \mapsto \mathbb{R}^{\geq 0}. \tag{2.2}$$

The sequences of cells will be denoted with arrows, for example $p_1 \to p_2 \to p_3 \to p_4$ for a path from $p_1$ via $p_2$ and $p_3$ to $p_4$ in this specified order.

If the intermediate nodes are obvious, the inner parts of the sequences are replaced by dots, especially if the sequence is taken as a path in the tree from the root to a defined node. For example, we would like to know the cost of getting from the root of the tree, $p_{\text{init}}$ to a vertex $p$ in the tree. Then $\text{cost}(p_{\text{init}} \to ... \to p)$ denotes the cost of travel from the root via all intermediate edges in the tree till the $p$ vertex. On the other hand, $\text{cost}(p_{\text{init}} \to p)$ denotes the cost of the direct path between two vertices in the state space. Both approaches can be concatenated in an intelligible form, for example going along a path in the tree and then deflecting to a vertex that is not in the tree.

In addition, a distance function d is also needed to properly define the algorithm. The function returns the distance between two vertices in the state space $\mathcal{S}$ and is formally defined as

$$\text{d} : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}^{\geq 0}. \tag{2.3}$$

**Algorithm 2**: RRT*($N$,$p_{\text{init}}$,$\epsilon$, $d$)

```
T ← empty_tree();
T.add_vertex(p_init);
while T has less than N vertices do
    p_random ← get_random_state();
    p_near ← T.find_nearest(p_random);
    p_new ← steer(p_near, p_random, ϵ);
    if edge between p_new and p_near is feasible then
        p_best ← T.find_best_parent_in_vicinity(p_new,p_near,d);
        T.add_vertex(p_new);
        T.set_parent(p_new,p_best);
        T.optimize(p_new,p_best,d);
    end
end
return T;
```

The basic outline of the Algorithm 2 is based on the RRT with the meaning of the functions already described in the section 2.1.1. Moreover, two other functions are defined: find_best_parent_in_vicinity($p_{\text{new}}$,$p_{\text{near}}$,$d$) and optimize($p_{\text{new}}$,$p_{\text{best}}$,$d$).

The former one is presented in the Algorithm 3 and it searches for a predecessor of the $p_{\text{new}}$ vertex. It iterates over all the vertices in the tree that are closer to $p_{\text{new}}$ than $d$ and returns the one that will result in the smallest cost for $p_{\text{new}}$ when added to the tree. Of course, only those vertices that will result in a feasible path are considered. Using this function, the new vertex $p_{\text{new}}$ is not assigned as a follower to the closest vertex, but to the one that minimises the final cost of travel.

The latter one, optimize($p_{\text{new}}$,$p_{\text{best}}$,$d$), defined in the Algorithm 4 also iterates over all the vertices $p$ in the tree that are closer to $p_{\text{new}}$ than the given distance $d$ and then tries

whether setting $p_{\text{new}}$ as a predecessor of $p$ would lower $p$'s cost. If it does, then $p_{\text{new}}$ is set as the parent of $p$ and the original connection of $p$ to its parent is disposed.

**Algorithm 3**: $\mathsf{T}$.find_best_parent_in_vicinity($p_{\text{new}}$,$p_{\text{near}}$,$d$)

```
p_best ← p_near;
for all vertices p in T where d(p, p_new) ≤ d do
    if edge between p_new and p is not feasible then
        continue;
    end
    if cost(p_init → ... → p → p_new) <cost(p_init → ... → p_best → p_new) then
        p_best ← p;
    end
end
return p_best;
```

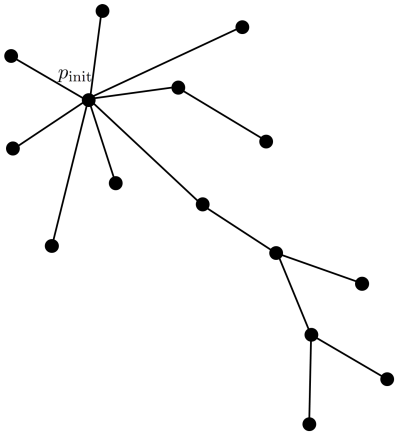
**Algorithm 4**: $\mathsf{T}$.optimize($p_{\text{new}}$,$p_{\text{best}}$,$d$)

```
for all vertices p in T where d(p, p_new) ≤ d and p ≠ p_best do
    if edge between p_new and p is not feasible then
        continue;
    end
    if cost(p_init → ... → p_new → p) <cost(p_init → ... → p) then
        T.set_parent(p,p_new);
    end
end
```

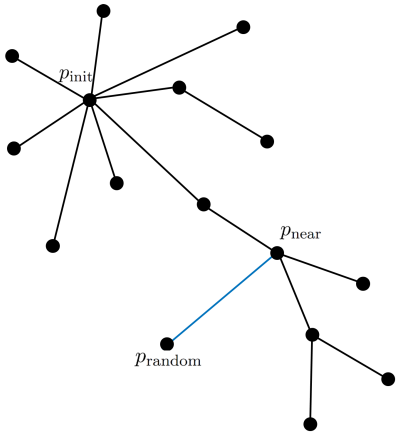
To make the algorithm more clear, an example of one iteration that adds a node to the tree is shown in the Figure 2.2. The tree that enters the iteration is shown in the Figure 2.2a. The node $p_{\text{random}}$ is randomly chosen from the state space and the node $p_{\text{near}}$ that is the nearest to $p_{\text{random}}$ from all the nodes in the tree is found, as shown in the Figure 2.2b.

Illustrated in the Figure 2.2c, the steer function produces the node $p_{\text{new}}$ that is in the distance $\epsilon$ from $p_{\text{near}}$ in the direction towards $p_{\text{random}}$. Next, the $p_{\text{new}}$ is connected to its parent $p_{\text{best}}$ that minimises the cumulative distance from $p_{\text{init}}$ to $p_{\text{new}}$. We only look for the candidates for $p_{\text{best}}$ in the radius $d$, as pictured in the Figure 2.2d.

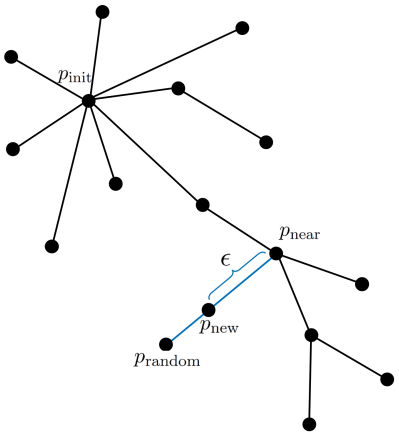
Finally, the optimizing function looks for nodes in the radius $d$ that would get their cost lowered if they were connected directly to the $p_{\text{new}}$ node instead of their current parent. In the Figure 2.2e, we can see that one node is joined as a follower of $p_{\text{new}}$, because it shortens its cumulative distance to $p_{\text{init}}$. The connection of the node to its original parent is annulled. This operation finishes the iteration and the tree that enters the next iteration is shown in the Figure 2.2f.



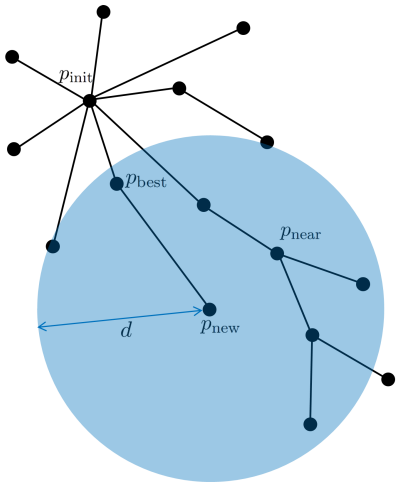

(a) The tree that will be extended.



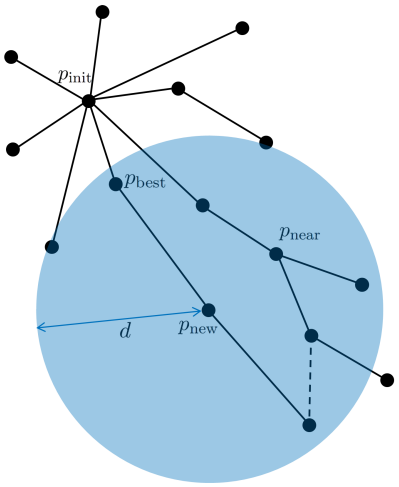

(b) Generated $p_{\text{random}}$ and found the nearest node $p_{\text{near}}$.



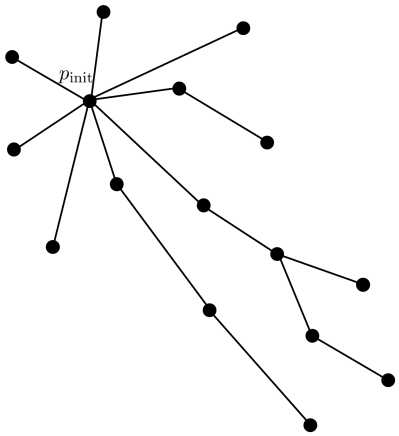

(c) The steer function produces the $p_{\text{new}}$.




(d) $p_{\text{new}}$ is connected to $p_{\text{best}}$




(e) A vertex is rewired as a successor of $p_{\text{new}}$.




(f) The tree after the iteration.

Figure 2.2: Illustration of one iteration of the RRT* algorithm.

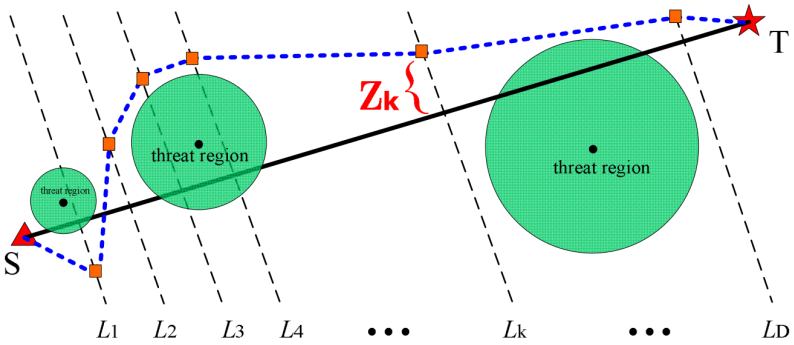



Figure 2.3: Illustration to the ABC algorithm for search-evasion planning, taken from Li et al. (2014)

### 2.1.3 Artificial Bee Colony algorithm

Li et al. (2014) proposed an evolutionary algorithm for planning search-evading paths for submarines. In the paper, the threat regions were represented as circles that a submarine should avoid. The algorithm first connects the starting point $S$ and the target point $T$ with a straight line and divides it into $D+1$ line segments by introducing lines $L_1, \ldots, L_D$ that are perpendicular to the line defined by the points $S$ and $T$. Next, the coordinate system is rotated in order to move $S$ to the origin and $T$ on the horizontal axis.

Then, a path in this coordinate system is defined as a vector $\mathbf{z} = (\mathbf{z}_1, \ldots, \mathbf{z}_D)$, where $\mathbf{z}_k$ is the vertical coordinate of the point where the path intersects the perpendicular $L_k$, as shown in the Figure 2.3.

In this method, the length of the path is taken into account and crossing through threat regions is penalized. Given these conditions, a vector $\mathbf{z}^* \in \mathbb{R}^D$ is seek that minimizes the cost function. Formally, the function that returns the costs is defined as

$$\text{cost} : \mathbb{R}^D \mapsto \mathbb{R}. \tag{2.4}$$

Informally, there are $B$ bees in the ABC algorithm that are divided into two groups, first half are the employed bees and the other half are the onlooker bees. The employed bees hold their configurations while the onlooker bees search around them and try to find better configurations. Technically, there is also a third type of bees, scout bees. An employed bee becomes a scout bee if its configuration has not improved in the predefined amount of iterations. The scout bee is assigned a random configuration by the random_initialization() function and after that, it becomes an employed bee again.

Formally, the Algorithm 5 searches for the vector $\mathbf{x}^i$ that will minimize cost($\mathbf{x}^i$). The variables $t_i$ denote the number of times the vector $\mathbf{x}^i$ was tried to be improved, but it did not. The $k$-th value of vector $\mathbf{x}^i$ is denoted as $\mathbf{x}^i_k$. The vector $\mathbf{x}^i$ represents the $i$-th employed bee, whereas $\mathbf{y}^i$ is the $i$-th onlooker bee.

In the algorithm, the values of $t_i$ are initially set to 0 and the values of $\mathbf{x}^i$ (the employed bees) are randomly initialized using the random_initialization() function presented in the Algorithm 6. The function returns a vector of dimension $D$ with limited bounds for each component of the vector. The lower bound of each dimension is in the vector

$\mathbf{min} \in \mathbb{R}^D$ and the upper bounds in the $\mathbf{max} \in \mathbb{R}^D$ vector. In detail, the vector is chosen from the set $\mathcal{V}$, as defined in the Equation 2.5. The function rand($\mathcal{M}$) returns a random value from the set $\mathcal{M}$. If the set $\mathcal{M}$ is infinitely large, then it is chosen with continuous uniform distribution. On the other hand, if the set $\mathcal{M}$ is finite, then the result is chosen with discrete uniform distribution. Due to technical reasons, the value of $\mathbf{x}^{\text{best}}$ is initialized to $\mathbf{x}^1$ and it maintains the best result found so far during the algorithm.

$$\mathcal{V} = [\mathbf{min}_1, \mathbf{max}_1] \times [\mathbf{min}_2, \mathbf{max}_2] \times \cdots \times [\mathbf{min}_D, \mathbf{max}_D] \tag{2.5}$$

After the initialization, the iterative cycle starts. In the cycle, first the employed_bee_phase() is performed. This function, which is described in the Algorithm 7, creates the configuration $\overline{\mathbf{x}}^i$ for each employed bee by randomly choosing the dimension $d$ and the other bee $k$. Then, the new configuration $\overline{\mathbf{x}}^i$ is created by randomly altering the value of $\mathbf{x}^i_d$ to one of the values in the interval $[\mathbf{x}^i_d - \delta, \mathbf{x}^i_d + \delta]$, where $\delta = |\mathbf{x}^k_d - \mathbf{x}^i_d|$. Finally, cost of $\overline{\mathbf{x}}^i$ is compared to $\mathbf{x}^i$. If the changed configuration is better, the value of $\mathbf{x}^i$ is overwritten and $t_i$ set to zero, because the corresponding value was updated. Otherwise, $t_i$ is increased by one due to the unsuccessful update.

Next, the fitness($i$) and P($i$) values are calculated for all the employed bees using the equations in the Algorithm 5. These values are used in the onlooker_bee_phase(), where onlooker bees are generated. The values of P($j$) are compared to a randomly chosen number in the interval [0, 1]. If the P($j$) is larger than the number, then the $i$-th onlooking bee with configuration $\mathbf{y}^i$ is created using the $j$-th employed bee $\mathbf{x}^j$. The vector $\mathbf{y}^i$ is generated using the vector $\mathbf{x}^j$ and changing its $k$-th component to one in the interval $[\mathbf{x}^j_k - \delta, \mathbf{x}^j_k + \delta]$, where $\delta = |\mathbf{x}^m_k - \mathbf{x}^j_k|$ and $\mathbf{x}^m$ is a randomly chosen employed bee other than $\mathbf{x}^j$. If the cost of $\mathbf{y}^i$ is then lower than $\mathbf{x}^j$, the $j$-th employed bee is redirected to the configuration $\mathbf{y}^i$ and the value $t_j$ is again reset to 0, otherwise it is increased, as in the employed_bee_phase().

After both phases, $\Omega$ is defined as a set of employed bees that have the values $t_i$ higher than a given threshold *limit*. If the set is not empty, one bee is chosen, randomly reinitialized and its $t_i$ value is set to 0.

The result of the algorithm is the best solution $\mathbf{x}^{\text{best}}$ that was found while iterating. The value is updated after each iteration by the get_best_result() function. The function goes through all the employed bees and returns the one with the lowest cost if its cost is lower than the best configuration found till now. This straightforward function is described in the Algorithm 9.

Using the algorithm that was explained above, the result can be interpreted as the vector $\mathbf{z}$ defining the path of the submarine. The ABC can also plan in real-time with changing environment. That means, the whole path does not need to be planned in advance, but only the next $N$ points will be planned, i.e. when the submarine crosses the line $L_i$, it will plan only points $\mathbf{z}_{i+1}, \ldots, \mathbf{z}_{i+N}$. After this trajectory is executed, the next parts are planned. The advantage of this process is not only the ability to plan in dynamic environment, but also dimension reduction, due to $N < D$.

**Algorithm 5**: ABC($B, limit, \mathbf{min}, \mathbf{max}$)

$t_i \leftarrow 0$ for all $i = 1, \ldots, \frac{B}{2}$;
$\mathbf{x}^i \leftarrow$ random_initialization() for all $i = 1, \ldots, \frac{B}{2}$;
$\mathbf{x}^{\text{best}} \leftarrow \mathbf{x}^1$;
**for** *predefined amount of iterations* **do**
  employed_bee_phase();
  $\text{fitness}(i) \leftarrow \begin{cases} \frac{1}{1+\text{cost}(\mathbf{x}^i)} & \text{if cost}(\mathbf{x}^i) \geq 0 \\ 1 - \text{cost}(\mathbf{x}^i) & \text{if cost}(\mathbf{x}^i) < 0 \end{cases}$ for all $i = 1, \ldots, \frac{B}{2}$;
  $\text{P}(i) \leftarrow \frac{\text{fitness}(i)}{\sum_{j=1}^{\frac{B}{2}} \text{fitness}(j)}$ for all $i = 1, \ldots, \frac{B}{2}$;
  onlooker_bee_phase();
  $\Omega \leftarrow \{i \,|\, t_i > limit\}$;
  **if** $\Omega \neq \emptyset$ **then**
    $k \leftarrow \text{rand}(\Omega)$;
    $\mathbf{x}^k \leftarrow$ random_initialization();
    $t_k \leftarrow 0$;
  **end**
  $\mathbf{x}^{\text{best}} \leftarrow$ get_best_result($\mathbf{x}^{\text{best}}$);
**end**
**return** $\mathbf{x}^{\text{best}}$;

**Algorithm 6**: random_initialization()

$\mathbf{z} \leftarrow (0, \ldots, 0) \in \mathbb{R}^D$;
**for** $i = 1, \ldots, D$ **do**
  $\mathbf{z}_i \leftarrow \mathbf{min}_i + \text{rand}([0, 1]) \cdot (\mathbf{max}_j - \mathbf{min}_j)$;
**end**
**return** $\mathbf{z}$;

**Algorithm 7**: employed_bee_phase()

**for** $i = 1, \ldots, \frac{B}{2}$ **do**
  $k \leftarrow \text{rand}(\{1, \ldots, \frac{B}{2}\} - \{i\})$;
  $d \leftarrow \text{rand}(\{1, \ldots, D\})$;
  $\overline{\mathbf{x}}^i \leftarrow \mathbf{x}^i$;
  $\overline{\mathbf{x}}^i_d \leftarrow \mathbf{x}^i_d + \text{rand}([-1, 1]) \cdot (\mathbf{x}^k_d - \mathbf{x}^i_d)$;
**end**
**for** $i = 1, \ldots, \frac{B}{2}$ **do**
  **if** $\text{cost}(\overline{\mathbf{x}}^i) < \text{cost}(\mathbf{x}^i)$ **then**
    $\mathbf{x}^i \leftarrow \overline{\mathbf{x}}^i$;
    $t_i \leftarrow 0$;
  **else**
    $t_i \leftarrow t_i + 1$;
  **end**
**end**

**Algorithm 8**: onlooker_bee_phase()

```
j ← 1;
for i = 1, ..., B/2 do
    if P(j) > rand([0, 1]) then
        m ← rand({1, ..., B/2} − {j});
        k ← rand({1, ..., D});
        y^i ← x^j;
        y^i_k ← x^j_k + rand([−1, 1]) · (x^m_k − x^j_k);
        if cost(y^i) < cost(x^j) then
            x^j ← y^i;
            t_j ← 0;
        else
            t_j ← t_j + 1;
        end
    end
    j ← j + 1;
end
```

**Algorithm 9**: get_best_result($\mathbf{x}^{\text{best}}$)

```
x^min ← x^best;
for i = 1, ..., B/2 do
    if cost(x^i) < cost(x^min) then
        x^min ← x^i;
    end
end
return x^min;
```

### 2.1.4 Probabilistic Roadmap

Like RRT or RRT*, the Probabilistic Roadmap (PRM) planning is a sampling-based method capable of running in complex environments and satisfying various conditions on the result, as stated in Saha (2006). PRM planning is also asymptotically complete.

In PRM, the state space $\mathcal{S}$ is divided into two disjunctive sets $\mathcal{S}_{\text{free}}$, where the object can be located, and $\mathcal{S}_{\text{obs}}$ that represents the obstacles in the state space. Additionally, it is needed to define the function feasible($p_1$, $p_2$) that for two given vertices $p_1, p_2 \in \mathcal{S}$ determines whether the edge joining the vertices does not go through the $\mathcal{S}_{\text{obs}}$ set and is therefore feasible.

The algorithm that searches a feasible connection between $V_{\text{start}}$ and $V_{\text{target}}$ can be divided into two phases. In the first phase, a graph $\mathsf{G}$ is created by adding $N$ vertices from $\mathcal{S}_{\text{free}}$. For this process, the function random_vertex($\mathcal{S}$) is needed. It returns a random vertex from the whole state space $\mathcal{S}$ that is further checked whether it belongs to $\mathcal{S}_{\text{free}}$. The vertices $V_{\text{start}}$ and $V_{\text{target}}$ are also added to the graph.

Next, for each vertex $V$ from the graph $\mathsf{G}$, a set $\mathcal{H}$ is created that contains the $k$ nearest neighbours of $V$ in $\mathsf{G}$ using the get_nearest_neighbours$(V, k)$ function. All the vertices $U \in \mathcal{H}$ are connected to $V$ via an edge if the edge is feasible, i.e. does not cross the $\mathcal{S}_{\text{obs}}$ area. This way, the graph is constructed.

The second phase involves only the call of plan_path$(\mathsf{G}, V_{\text{start}}, V_{\text{target}})$ function. The function returns a path in the $\mathsf{G}$ graph starting in node $V_{\text{start}}$ and ending in $V_{\text{target}}$. This can be for example the Dijkstra's algorithm.

**Algorithm 10**: PRM$(N, k, V_{\text{start}}, V_{\text{target}})$

```
G ← empty_graph();
G.add_vertex(V_start);
G.add_vertex(V_target);
while G has fewer vertices than N do
    V ← random_vertex(S);
    if V ∈ S_free then
        G.add_vertex(V);
    end
end
for all vertices V in G do
    H ← G.get_nearest_neighbours(V, k);
    for all vertices U ∈ H do
        if feasible(V, U) then
            G.add_edge(V, U);
        end
    end
end
p ← plan_path(G, V_start, V_target);
return p;
```

## 2.2 Simulations

In agent-based (also called multi-agent or agent-oriented) simulations, there are active autonomous entities (agents) that are present in the environment[1] defined by the simulation. Each agent is located in the environment, pursues its goals and interacts with other agents. The interaction may be limited, as well as the knowledge of an agent about the whole environment. The agent gains information concerning the environment through interactions that are mediated through its sensors or inter-agent communication. The interactions are usually spatially restricted so that only those agents that are within reach can exchange information or perceive each other. Other characteristics of an agent are pro-activity and reactivity, meaning that it should be able to actively respond to changes in its surroundings.

The properties of agents mentioned above are contingent on the characteristics of the environment. For example, it can dynamically change with time or due to agents' activities

[1]The agent is not considered a part of the environment, but a separate entity.

or be static. Next, the world can behave deterministically, then the outcomes of particular actions or situations are known and can be predicted, or the world may be stochastic and the results depend on random variables or unpredictable events. The last distinction is between continuous and discrete environment. In the latter one, there is only finite amount of states or locations, as opposed to the former option.

Another property of a simulation is the time advance paradigm. In systems driven by differential or difference equations, a continuous simulation is usually used. Another approach is the time-stepped simulation which resembles a timer that updates the agents and the state of the environment periodically. The last approach mentioned in Klügl (2009) is a discrete event-based simulation. In this paradigm, a queue of events is maintained and the time is skipped from one event to the next one, as the state variables do not change between events.

The term granularity of a simulation determines the level of detail in which the world is modelled. The basic distinction is between the macro and the micro simulations. In the former one, the whole environment is described by state variables as a homogeneous integral unit. On the other hand, the microscopic level consists of multiple entities with their own variables, various capabilities and behaviour models.

# Chapter 3

# Formalization

This chapter formalizes the used terms. First, the gridding method and characteristics of cells are introduced in 3.1. Then, the cost function, time windows, risk of detection and additional issues are discussed in 3.2. The whole task and its constraints are defined in 3.3. The algorithms that are used to solve the task are then presented in 3.4.

## 3.1 Grid

In planning tasks, it is beneficial to discretise the continuous real world in order to simplify computations. Thus, a grid is introduced as a set of cells that fill a given space in predefined density. In our case, the space is a volume of water.

### 3.1.1 Implementation of the grid

A popular gridding method partitions Earth's surface along meridians and parallels in fixed angular intervals. Meteorological data, such as OSCAR from ESR (2009) or ETOPO by National Geophysical Data Center (2006), are arranged in this manner. Even though this arrangement is simple, the main disadvantage is that the size of cells varies significantly with changing latitude.

To avoid this drawback, the intervals are not fixed in our implementation. Approximating the Earth as a sphere, the coordinates can be computed so that the cells have roughly the same dimensions by enlarging the zonal intervals as the position of a cell approaches one of the geographical poles. Thus, the areas covered by the cells are close to being uniform. A collateral effect of this choice is the uneven position of cells and creation of cells which may partially lie outside of the designated area, as seen on the eastern boundary of the grid in the Figure 3.1. Additionally, it is needed to not only sample the surface, but also the areas below surface. Therefore, more layers of cells at specified vertical distance are created.

In the Figure 3.1, there are outlined the areas of 409 cells in one layer. It may not seem so, but even the cells near Antarctica have approximately the same size as those near

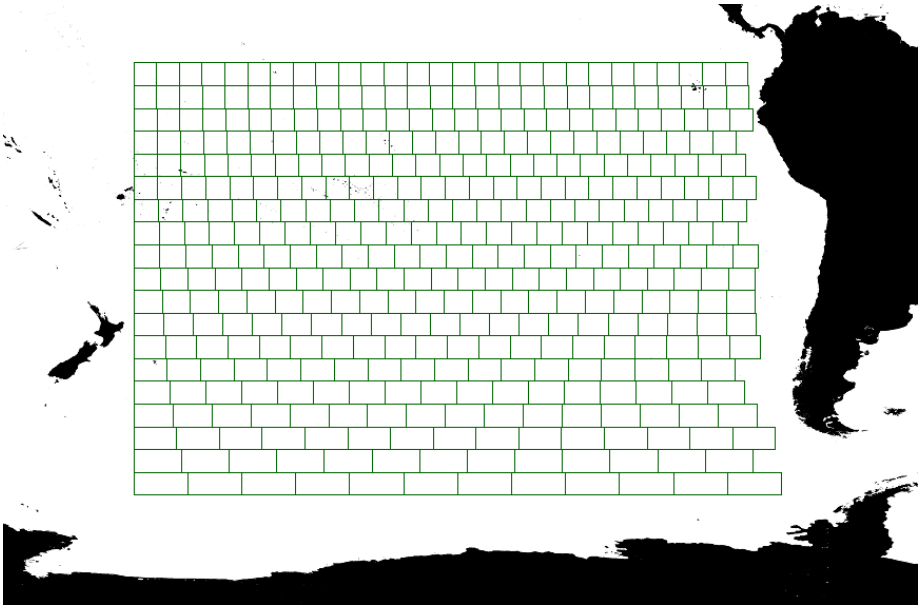


Figure 3.1: Grid in Southern Pacific (equirectangular projection).

equator. Only the distortion caused by projection from spherical surface to plane results in the sizes appearing non-uniform.

The grid is defined by the following parameters:

- the position of the area that is discretised, defined by GPS coordinates of north-western $(\varphi_{\text{North}}, \lambda_{\text{West}})$ and south-eastern corners $(\varphi_{\text{South}}, \lambda_{\text{East}})$ of the area,
- the size $s$ of the cells, usually specified in kilometres - the distance from western to eastern boundary and the distance from northern to southern boundary both equal $s$,
- the depth interval $\Delta d$, distance between layers of cells,
- depth $m$ of the deepest layer.

In the Algorithm 11 that creates the grid, the values of longitudes are checked. If $\lambda_{\text{East}} \leq \lambda_{\text{West}}$, then the grid should include also the Date Line. That means that the value of $\lambda_{\text{East}}$ is increased by 360 degrees to maintain the characteristic that the $\lambda_{\text{East}}$ is larger than $\lambda_{\text{West}}$ and thus truly eastwards. This will not effect anything, since the values of longitude are normalized when creating the actual cells. The latitudes and longitudes are in decimal degrees[1]. The value $R$ in the algorithm denotes the radius of the Earth.

If the grid should contain the North Pole, it is added separately to avoid problems in the equations. The South Pole is not an issue since there is no ocean there, but it would be implemented likewise if the need arose.

Then, two loops generate all the areas to be covered by the cells. The outer loop changes the latitude by constant value $\Delta\varphi$, this can be done due to the fact that the

[1]That means that the latitudes are normalized to the interval [-90, 90], where the negative values are found on the Southern Hemisphere, and the longitudes are real numbers from the interval [-180, 180], where the negative values are on the Western Hemisphere.

surface distance between two parallels with angular difference $\Delta\varphi$ is always equal to $s$. On the other hand, the surface distance between meridians differs with variable latitude, thus the value of $\Delta\lambda$ is calculated for every latitude. In this process, all the areas are defined by the function area$(\lambda_{\mathrm{W}}, \lambda_{\mathrm{E}}, \varphi_{\mathrm{N}}, \varphi_{\mathrm{S}})$, where $\lambda_{\mathrm{W}}$ and $\lambda_{\mathrm{E}}$ are the western and eastern longitude bounds for the area and $\varphi_{\mathrm{N}}$ and $\varphi_{\mathrm{S}}$ are the northern and southern latitude bounds.

**Algorithm 11**: create_grid$(s, \Delta d, m, \varphi_{\mathrm{North}}, \varphi_{\mathrm{South}}, \lambda_{\mathrm{West}}, \lambda_{\mathrm{East}})$

$\mathcal{G} \leftarrow \emptyset$;
**if** $\lambda_{\mathrm{East}} \leq \lambda_{\mathrm{West}}$ **then**
  $\lambda_{\mathrm{East}} \leftarrow \lambda_{\mathrm{East}} + 360°$;
**end**
**if** $\varphi_{\mathrm{North}} = 90°$ **then**
  $\mathcal{G} \leftarrow \mathcal{G} \cup$ create_cells(North Pole, $\Delta d, m$);
  $\varphi_{\mathrm{North}} \leftarrow \varphi_{\mathrm{North}} - \epsilon$;
**end**
$\varphi \leftarrow \varphi_{\mathrm{North}}$;
$\Delta\varphi \leftarrow \dfrac{360° \cdot s}{2\pi R}$;
**while** $\varphi \leq \varphi_{\mathrm{South}}$ **do**
  $\lambda \leftarrow \lambda_{\mathrm{West}}$;
  $\Delta\lambda \leftarrow \dfrac{360° \cdot s}{2\pi R \cdot \cos|\varphi|}$;
  **while** $\lambda \leq \lambda_{\mathrm{East}}$ **do**
    $A \leftarrow \mathrm{area}(\lambda, \lambda + \Delta\lambda, \varphi, \varphi - \Delta\varphi)$;
    $\mathcal{G} \leftarrow \mathcal{G} \cup$ create_cells$(A, \Delta d, m)$;
    $\lambda \leftarrow \lambda + \Delta\lambda$;
  **end**
  $\varphi \leftarrow \varphi - \Delta\varphi$;
**end**
**return** $\mathcal{G}$;

**Algorithm 12**: create_cells$(A, \Delta d, m)$

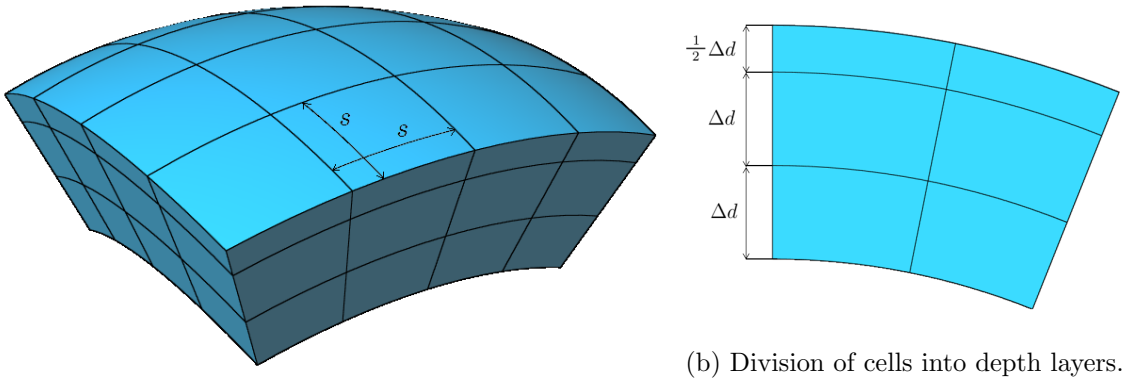
$\mathcal{S} \leftarrow \emptyset$;
$depth_{\max} \leftarrow$ get_depth_in_area$(A)$;
**if** $depth_{\max} \geq 0$ **then**
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathrm{cell}(A, 0, 0, \frac{1}{2}\Delta d)\}$;
  $d \leftarrow \frac{3}{2}\Delta d$;
  **while** $d \leq depth_{\max} \wedge d \leq m$ **do**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathrm{cell}(A, d - \frac{1}{2}\Delta d, d, d + \frac{1}{2}\Delta d)\}$;
    $d \leftarrow d + \Delta d$;
  **end**
**end**
**return** $\mathcal{S}$;

The function create_cells$(A, \Delta d, m)$ returns a set of cells in the predefined area $A$, but in different layers. The function first estimates the depth $depth_{\max}$ in the area given by the function get_depth_in_area$(A)$. If the depth is negative, it means that there is no ocean, but land. In such case, the function returns an empty set. Otherwise, it starts by adding the surface cell separately. This cell spans from depth 0 to $\frac{1}{2}\Delta d$ and has its representative depth[2] equal to 0. The notation cell$(A, d_{\min}, d_{\text{rep}}, d_{\max})$ denotes a cell in the area $A$ that covers depths from $d_{\min}$ to $d_{\max}$ and has the representative depth $d_{\text{rep}}$. The cells that are not on surface have the depth range equal to $\Delta d$, whereas the surface cells have half of it.

If we define $p = \max(depth_{\max}, m)$, there will be exactly $N = \left\lfloor \frac{p}{\Delta d} \right\rfloor + 1$ cells. Then, a cell in the $k$-th layer, $k \in 1, ..., N$, has its representative depth equal to $(k-1)\Delta d$. The upper and lower boundaries of the layers are in depths $(k - \frac{1}{2})\Delta d$. All layers except the one on the surface have their representative depth equal to their average depth.

In the following text, the grid will be denoted as $\mathcal{G}$, or $\mathcal{G}_s$ if the size of the cells needs to be accentuated.

In the Figure 3.2, the exaggerated shape of the cells is shown with dimensioned $s$ and $\Delta d$ values.




(a) The shape of cells with dimensioned $s$.

(b) Division of cells into depth layers.

Figure 3.2: Grid with cells in 3 layers.

### 3.1.2 Cells

As introduced in 3.1.1, each cell is assigned multiple values concerning its position. In a grid $\mathcal{G}$, each cell $c \in \mathcal{G}$ has its middle point $\mathbf{c}^{\text{M}} = (\varphi, \lambda)$ with latitude $\varphi \in [-90, 90]$ and longitude $\lambda \in [-180, 180]$. Analogously, the four corners of $c$ have coordinates $\mathbf{c}^1, \mathbf{c}^2, \mathbf{c}^3$ and $\mathbf{c}^4$. To make calculations easier and faster, the spherical coordinates can be transformed to the Cartesian coordinates using the mapping $\Phi : \mathbb{R}^2 \mapsto \mathbb{R}^3$ with

$$\Phi(\varphi, \lambda) = \begin{bmatrix} R \cdot \cos\varphi \sin\lambda \\ R \cdot \cos\varphi \cos\lambda \\ R \cdot \sin\varphi \end{bmatrix}, \text{where } R \text{ is the radius of the Earth.} \tag{3.1}$$

[2]The term representative depth was chosen because together with the centre of the cell, it represents the position of the cell in simulations.

Note that the depths of the cells are disregarded in these calculations. Using the mapping, the Cartesian coordinates are obtained as

$$\mathbf{C}^i = \Phi(\mathbf{c}^i), \text{for } i \in \{\mathrm{M}, 1, 2, 3, 4\}. \tag{3.2}$$

To make the notation clear, all the properties of a single cell in the grid are listed in the Table 3.1.

| Parameter | Value |
|---|---|
| $\mathbf{c}^{\mathrm{M}} \in \mathbb{R}^2$ | spherical coordinates of the middle of the cell |
| $\mathbf{c}^1, ..., \mathbf{c}^4 \in \mathbb{R}^2$ | spherical coordinates of the corners of the cell |
| $\mathbf{C}^{\mathrm{M}} \in \mathbb{R}^3$ | Cartesian coordinates of the middle of the cell |
| $\mathbf{C}^1, ..., \mathbf{C}^4 \in \mathbb{R}^3$ | Cartesian coordinates of the corners of the cell |
| $c^{\overline{\mathrm{D}}} \in \mathbb{R}$ | maximum depth of the cell |
| $c^{\underline{\mathrm{D}}} \in \mathbb{R}$ | minimum depth of the cell |
| $c^{\mathrm{D}} \in \mathbb{R}$ | representative depth of the cell |

Table 3.1: Overview of the parameters of a cell $c$.

## 3.2 Cost function

The choice of the cost function is a crucial step in optimization tasks. In this case, travelled distance, risk of detection and elapsed time will be taken into account. Unlike most cases, in this one, the cost of travel does not only depend on the starting and ending cell, but also on all the cells that are along the way. That is why an algorithm for creating the list of cells on given path is introduced.

### 3.2.1 Creating list of cells on given path

In this algorithm, we will use the fact that the geodesic between two points on a sphere is a segment of the great circle of that sphere. A great circle is an intersection of the sphere and any plane that goes through the centre of the sphere. The method is described in pseudocode in the Algorithm 13.

The input of this algorithm is a grid $\mathcal{G}_s$, a starting cell $c_S \in \mathcal{G}_s$, and an ending cell $c_E \in \mathcal{G}_s$. Then, the plane $p$ that defines the corresponding great circle goes through points $\mathbf{C}_S^{\mathrm{M}}$ and $\mathbf{C}_E^{\mathrm{M}}$. The centre of Earth, which is also on the plane $p$, has coordinates $(0, 0, 0)$. Therefore, plane $p$ can be defined by its normalised normal vector $\mathbf{n}$. The first argument of the function (i.e. the grid) can be omitted if it is clear in the context to simplify notation.

The distance of a point $\mathbf{C}_i^{\mathrm{M}}$ from the plane $p$ equals $|\mathbf{n} \cdot \mathbf{C}_i^{\mathrm{M}}|$. If the vertical shape of a cell is approximated to be a isosceles trapezoid, all the cells that are on the path from $c_S$ to $c_E$ must have the distance from their middle point to $p$ lower than $s\left(\frac{\sqrt{5}}{2} + \epsilon\right)$. The value of $\epsilon$ is added so that no cells are lost due to precision errors. If the previous condition was required only, it would be satisfied by any cell on the whole great circle around the Earth.

However, we are interested only in the segment between $c_S$ and $c_E$, thus function check$(c_i, c_S, c_E)$ is introduced. The function calculates the northernmost and southernmost latitudes $\varphi_{\text{North}}, \varphi_{\text{South}}$ along the path by the get_latitude_bounds_of_trajectory$(c_S, c_E)$ function, for example iteratively by sampling the trajectory. It also creates longitude bounds that are given directly by the middle points of $c_S$ and $c_E$.

**Algorithm 13**: list$(\mathcal{G}_s, c_S, c_E)$

$$\mathbf{n} \leftarrow \frac{\mathbf{C}_S^{\text{M}} \times \mathbf{C}_E^{\text{M}}}{\left\|\mathbf{C}_S^{\text{M}} \times \mathbf{C}_E^{\text{M}}\right\|_2};$$

$$\mathcal{X}_{\text{approx}} \leftarrow \left\{ c \in \mathcal{G}_s \middle| |\mathbf{n} \cdot \mathbf{C}_i^{\text{M}}| < s\left(\frac{\sqrt{5}}{2} + \epsilon\right) \wedge \text{check}(\mathbf{c}_i, \mathbf{c}_S, \mathbf{c}_E) \right\};$$

$$\mathcal{X}_{\text{depths}} \leftarrow \left\{ c \in \mathcal{X}_{\text{approx}} \middle| \underline{c^{\text{D}}} \leq \frac{\text{d}(c, c_S) \cdot c_S^{\text{D}} + \text{d}(c, c_E) \cdot c_E^{\text{D}}}{\text{d}(c, c_S) + \text{d}(c, c_E)} < \overline{c^{\text{D}}} \right\};$$

$$\mathcal{X}_{\text{exact}} \leftarrow \{c \in \mathcal{X}_{\text{depths}} | \max_{i=1,\dots 4} \text{sign}(\mathbf{n} \cdot \mathbf{C}^i) = 1 \wedge \min_{i=1,\dots 4} \text{sign}(\mathbf{n} \cdot \mathbf{C}^i) = -1\};$$

$\mathcal{L} \leftarrow \emptyset$;

**for** *each cell* $c$ *in* $\mathcal{X}_{\text{exact}}$ **do**

  $I_1, I_2 \leftarrow$ get_intersection_points(edges($c$),$\mathbf{n}$);

  **if** $c = c_S$ or $c = c_E$ **then**

    $\mathcal{L} \leftarrow \mathcal{L} \cup \left\{ \left( c, \frac{\text{d}(I_1, I_2)}{2} \right) \right\}$;

  **else**

    $\mathcal{L} \leftarrow \mathcal{L} \cup \{(c, \text{d}(I_1, I_2))\}$;

  **end**

**end**

**return** $\mathcal{L}$;

**Algorithm 14**: check$(c_i, c_S, c_E)$

$\varphi_{\text{North}}, \varphi_{\text{South}} \leftarrow$ get_latitude_bounds_of_trajectory$(\mathbf{c}_S^{\text{M}}, \mathbf{c}_E^{\text{M}})$;

$\lambda_{\text{East}} \leftarrow$ max_longitude$(\mathbf{c}_S^{\text{M}}, \mathbf{c}_E^{\text{M}})$;

$\lambda_{\text{West}} \leftarrow$ min_longitude$(\mathbf{c}_S^{\text{M}}, \mathbf{c}_E^{\text{M}})$;

$\varphi_i \leftarrow$ get_latitude$(\mathbf{c}_i^{\text{M}})$;

$\lambda_i \leftarrow$ get_longitude$(\mathbf{c}_i^{\text{M}})$;

**if** $\varphi_i \notin [\varphi_{\text{South}}, \varphi_{\text{North}}]$ **then**

  **return** *false*;

**end**

**if** *the shortest path between* $c_S$ *and* $c_E$ *crosses the Date Line* **then**

  **return** $\lambda_i \in [\lambda_{\text{West}}, \lambda_{\text{East}}]$;

**else**

  **return** ($\lambda_i \geq \lambda_{\text{East}}$ or $\lambda_i \leq \lambda_{\text{West}}$);

**end**

In the first step, only an approximate set $\mathcal{X}_{\text{approx}}$ is created to quickly filter the cells that are definitely not on the path.

The depth limits of cells were not reflected into the Cartesian coordinates, meaning that if a cell is present in $\mathcal{X}_{\text{approx}}$, then all the cells that only differ by depth are also

present. Filtering depth is straightforward, since it is assumed that the depth changes linearly during the voyage. Define the starting depth $c_S^D$ as the representative depth of $c_S$ and the ending depth $c_E^D$ as the representative depth of $c_E$. Also define $\mathrm{d}(c_i, c_j)$ as the spherical distance from cell $c_i$ to $c_j$. This is how the $\mathcal{X}_{\text{approx}}$ is filtered from superfluous cells into the $\mathcal{X}_{\text{depths}}$ set.

Finally, the cells can be exactly filtered from the $\mathcal{X}_{\text{depths}}$ set by determining in which half-space the corners of the cells lie. If all the corners of a cell are in the same half-space, the cell is not intersected by the plane and therefore not on the path. Otherwise, the cell is approximated to a convex quadrilateral, the edges of the cell that cross the plane $p$ are determined and the exact length of the intersection is calculated as the distance between the two intersection points. The reason why this procedure was not applied on the full set $\mathcal{G}_s$ is that there could be a lot of cells and the approximate filtering performs only one dot product per cell, unlike the exact filtering that needs four dot products.

Now, the set of cells that are along the path is known, but it is also necessary to assign the length of intersection to each cell using get_intersection_points(edges($c$),$\mathbf{n}$) function. The computation is easy, since we only need to find the intersections of the edges of the cell and the plane going through the origin point (0,0,0) defined by its normal vector $\mathbf{n}$. The edges of a cell are defined by the corners of the cell

$$\text{edges}(c) = \{(\mathbf{C}^1, \mathbf{C}^2), (\mathbf{C}^2, \mathbf{C}^3), (\mathbf{C}^3, \mathbf{C}^4), (\mathbf{C}^4, \mathbf{C}^1)\}. \tag{3.3}$$

The intersection points are then calculated as an intersection of a plane and a line segment. The corresponding distance travelled in the cell is approximated as the distance of the two intersection points. The only exceptions are the two cells $c_S$ and $c_E$. Their corresponding distances need to be halved, because the starting and ending point is in the middle of these cells, not at the boundary.

The approximations that were made across the algorithm are precise enough, even with large cells with size $s$=100 km, the error in length is lower than 3 %.

The output of this algorithm is the set $L$ consisting of the cells from $\mathcal{X}_{\text{exact}}$ and corresponding lengths $d_i$, distances travelled in cells $c_i \in \mathcal{X}_{\text{exact}}$ on the path from $c_S$ to $c_E$. As mentioned earlier, the grid argument can be omitted to simplify notation if the grid is clearly known, that is for instance

$$\text{list}(c_S, c_E) = \{(c_S, d_S), ..., (c_E, d_E)\}. \tag{3.4}$$

### 3.2.2 Time windows

Before the form of the cost function is introduced, it is important to mention that our implementation is able to handle changing environment, i.e. the risk of detection changes in time. It is modelled discretely by dividing the whole execution time into multiple time windows $0, 1, ..., E$. Each time window (TW) has the same duration.

### 3.2.3 Form of the cost function

The input of the cost function consists of two cells or a sequence of cells with specified time windows in which the individual distances are crossed. Clearly, the time windows have to be in non-decreasing order.

For instance, the cost of voyage from cell $c_1$ to $c_2$ in time window $t_1$ followed by proceeding to cell $c_3$ in time window $t_2$ will be denoted as $\text{cost}(c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} c_3)$ given the condition that $t_1 \leq t_2$. These requirements lead to the redefinition

$$\text{cost} : \mathcal{G} \times \mathcal{M} \times \mathcal{G} \times \mathcal{M} \times \cdots \times \mathcal{G} \mapsto \mathbb{R}^{\geq 0}, \text{ where } \mathcal{M} = \{0, ..., E\}. \tag{3.5}$$

The value of the cost function is determined by 3 criteria: travelled distance, risk of detection and elapsed time. The travelled distance is defined by the d function

$$\text{d} : \mathcal{G} \times \mathcal{G} \mapsto \mathbb{R}^{\geq 0} \tag{3.6}$$

that for two cells in a grid returns their distance. In detail, it is the spheric distance calculated by the Haversine formula, as explained in Korn and Korn (1967).

Next, p function represents the risk of being detected along the path in given time. The input for this function is a set of cells that are on the path, as defined in 3.2.1 and the time window in which the path is executed, formally

$$\text{p} : 2^{\mathcal{G} \times \mathbb{R}^{\geq 0}} \times \{0, 1, ..., E\} \mapsto \mathbb{R}^{\geq 0}. \tag{3.7}$$

The particular definition of this function is not straightforward and will be further investigated in the Section 3.2.4.

The last component is the time window in which the voyage is finished. This task is an instance of multi-objective optimization. To allow comparing, the weighted sum method is used, as mentioned in Grodzevich and Romanko (2006). The cost of travel is calculated as a conic combination[3] of multiple values. Weighs $w_i$ represent the importance of each component. Note that the normalization constraint $\sum_{i=1}^{3} w_i = 1$ is not necessary, because positive linear transformation does not change preferences.

With this set up, the form of the cost function can be expressed for a path between two cells $c_S$ and $c_E$ executed in the time window $t$, as

$$\text{cost}(c_S \xrightarrow{t} c_E) = w_1 \cdot \text{d}(c_S, c_E) + w_2 \cdot \text{p}(\text{list}(c_S, c_E), t) + w_3 \cdot t. \tag{3.8}$$

For longer inputs, it is needed to know the p function. The possible outcomes are presented in equations 3.14 and 3.17.

[3] Conic combination is any linear combination with non-negative coefficients

### 3.2.4 Risk of detection

To concretely define the p function, we need to assume that a function

$$\mathrm{RD} : \mathcal{G} \times \{0, ..., E\} \mapsto \mathbb{R}^{\geq 0} \tag{3.9}$$

is available. This function assigns a non-negative value representing the risk of detection to each cell in the grid in given time.

Additionally, a function

$$\mathrm{d_{max}} : \mathcal{G} \mapsto \mathbb{R}^{\geq 0} \tag{3.10}$$

is needed. It returns the length of the longest straight line segment across a cell. The cells have approximately the shape of isosceles trapezoids, so the function returns either the length of a diagonal or the length of one of its sides, depending on which value is the largest. The value is not the same for all cells, so it is necessary to define it as a function.

If an object crosses along the longest straight line segment across a cell $c \in \mathcal{G}$ in time $t$, then the risk of this object in this cell is $\mathrm{RD}(c, t)$. That does not give us any information about the risk if the object crossed the cell in length $d < \mathrm{d_{max}}(c)$. This leads to another function $\mathrm{p_m}(c, t, d)$ that returns the risk of detection if an object crosses only a distance $d$ in a cell $c$ in specified time window $t$. This function is formally defined as

$$\mathrm{p_m} : \mathcal{G} \times \{0, ..., E\} \times \mathbb{R}^{\geq 0} \mapsto \mathbb{R}^{\geq 0}. \tag{3.11}$$

There will be the following constraints on the $\mathrm{p_m}$ function:

- The value of the function grows with increasing $\mathrm{RD}(c, t)$ or $d$.
- $\forall t \in \{0, ..., E\} \, \forall c \in \mathcal{G} : \lim_{d \to \mathrm{d}^-_{\mathrm{max}}(c)} \mathrm{p_m}(c, t, d) = \mathrm{RD}(c, t)$

  This states that as the crossed distance approaches the maximal distance, the value of $\mathrm{p_m}$ converges to the value of RD.
- $\forall t \in \{0, ..., E\} \, \forall c \in \mathcal{G} : \mathrm{p_m}(c, t, 0) = 0$

  This expresses the fact that if the cell was not actually crossed (distance $d = 0$), the risk equals zero.
- $\forall t \in \{0, ..., E\} \, \forall c \in \mathcal{G} : \mathrm{RD(c,t)} = 0 \implies \forall d \in [0, \mathrm{d_{max}}(c)] : \mathrm{p_m}(c, t, d) = 0$

  If the risk in a cell equals zero, the value of $\mathrm{p_m}$ will also be zero.
- The total risk of detection on path $\{(c, d)\}$ (path with one cell) should be equal to the risk of detection on path consisting of $N$ cells $\{(c_1, \frac{d}{N}), ..., (c_N, \frac{d}{N})\}$ if $\mathrm{RD}(c_1, t) = ... = \mathrm{RD}(c_N, t) = \mathrm{RD}(c, t)$. This condition ensures that crossing through space with uniform risk of detection does not depend on the amount of cells crossed, but only on the overall distance.

In accordance with these constraints, two admissible approaches are introduced.

### Probabilistic approach

First, exact probabilistic approach is applied. It is assumed that the detections in individual cells are independent events and the risk of detection is equal to the probability of being at least once detected. The function p defined below will be referred to as detection probability. In this case, the range of RD values must be limited to interval $[0, 1]$.

$$\mathrm{p_m}(c, t, d) = 1 - (1 - \mathrm{RD}(c, t))^{\frac{d}{\mathrm{d_{max}}(c)}} \tag{3.12}$$

$$\mathrm{p}\left(\{(c_1, d_1), ..., (c_N, d_N)\}, t\right) = 1 - \prod_{i=1}^{N} (1 - \mathrm{p_m}(c_i, t, d_i)) = 1 - \prod_{i=1}^{N} (1 - \mathrm{RD}(c_i, t))^{\frac{d_i}{\mathrm{d_{max}}(c_i)}} \tag{3.13}$$

That gives the corresponding cost function:

$$\begin{aligned} &\mathrm{cost}(c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} c_3 \xrightarrow{t_3} \ldots \xrightarrow{t_{N-1}} c_N \xrightarrow{t_N} c_{N+1}) = \\ &w_1 \cdot \sum_{i=1}^{N} \mathrm{d}(c_i, c_{i+1}) + w_2 \cdot \left(1 - \prod_{i=1}^{N} (1 - \mathrm{p}(\mathrm{list}(c_i, c_{i+1}), t_i))\right) + w_3 \cdot t_N \end{aligned} \tag{3.14}$$

### Linear approach

Another approach is not to compute the exact probabilities, but only linearly scale and add the risk values, creating a detection score.

$$\mathrm{p_m}(c, t, d) = \frac{d}{\mathrm{d_{max}}(c)} \cdot \mathrm{RD}(c, t) \tag{3.15}$$

$$\mathrm{p}\left(\{(c_1, d_1), ..., (c_N, d_N)\}, t\right) = \sum_{i=1}^{N} \mathrm{p_m}(c_i, t, d_i) = \sum_{i=1}^{N} \frac{d_i}{\mathrm{d_{max}}(c_i)} \cdot \mathrm{RD}(c_i, t) \tag{3.16}$$

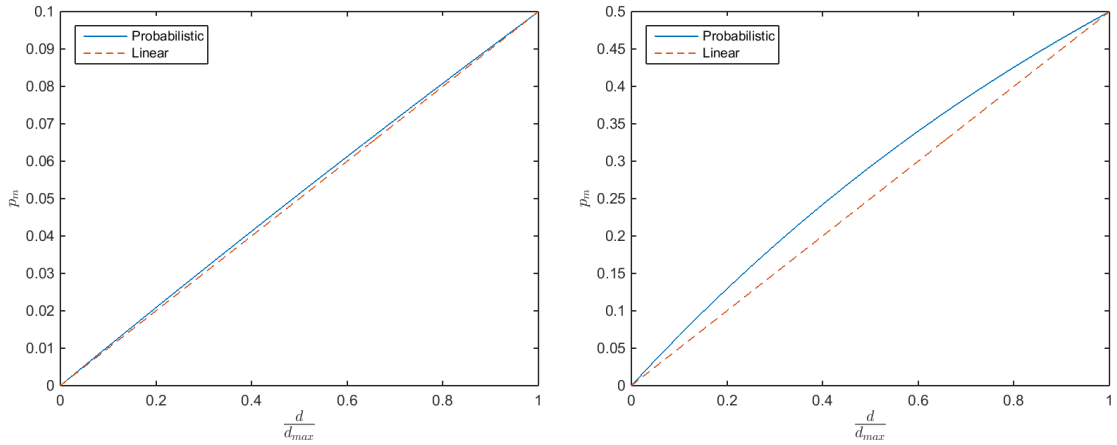
This approach results in the following form of the cost function:

$$\begin{aligned} &\mathrm{cost}(c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} c_3 \xrightarrow{t_3} \ldots \xrightarrow{t_{N-1}} c_N \xrightarrow{t_N} c_{N+1}) = \\ &w_1 \cdot \sum_{i=1}^{N} \mathrm{d}(c_i, c_{i+1}) + w_2 \cdot \sum_{i=1}^{N} \mathrm{p}(\mathrm{list}(c_i, c_{i+1}), t_i) + w_3 \cdot t_N \end{aligned} \tag{3.17}$$

### Comparison of approaches

In this section, the values taken from the linear approach will be referred to as detection score and the values from probabilistic approach as detection probability.

For small values of $\mathrm{RD}(c, t)$, the values of $\mathrm{p_m}$ will be almost similar in both approaches. However, with higher values of $\mathrm{RD}(c, t)$, the difference increases, as shown in the Figure 3.3.




(a) $\mathrm{RD}(c,t) = 0.1$ (b) $\mathrm{RD}(c,t) = 0.5$

Figure 3.3: Comparison of the suggested $\mathrm{p_m}$ functions.

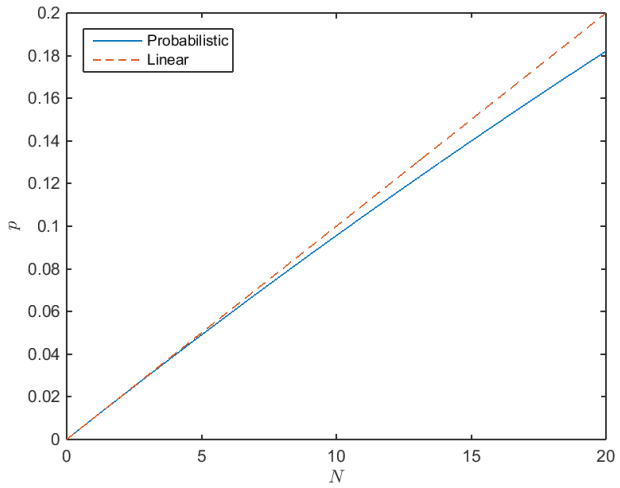
More important are the values of p, as these are used in the cost function. The probability of detection has 1 as its upper bound, therefore if the probabilities of detection in individual cells are large, the value of p will converge to 1 with increasing length of path and then, the risk of detection will not affect the cost function, as it remains constant. This is a huge disadvantage of the probabilistic approach. In the same conditions, detection score will still increase, because it is not bounded and even for large $\mathrm{RD}(c,t)$, it can still be scaled with the travelled distance. This phenomenon is shown in the Figure 3.4, where the values of $\mathrm{p}(\{(c_1, d_{\max}), ..., (c_N, d_{\max})\})$ are plotted for variable $N$ and constant $\mathrm{RD}(c_1,t) = ... = \mathrm{RD}(c_N,t) = \mathrm{RD}(c,t)$.

All in all, with small RD values, the results are about the same and with larger RD values, the detection probability may converge to 1 quickly and then become a constant in the cost function. This is why the detection score is used in our implementation.

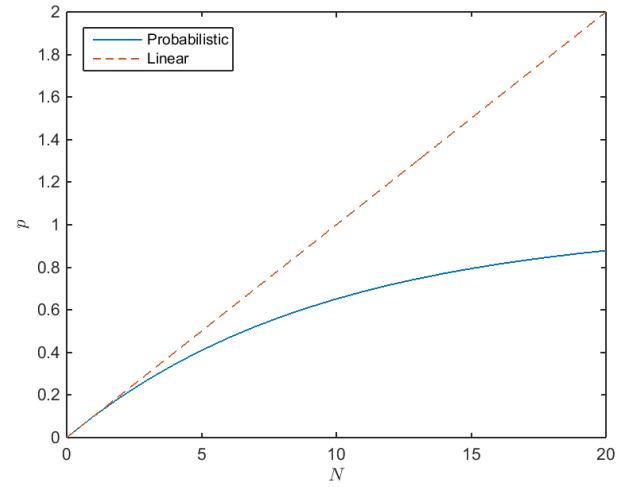
In addition, the detection score can be further generalised. There is no need to estimate the risk of detection $\mathrm{RD}(c,t)$ as a probability in the $[0,1]$ range. It is sufficient to do it only proportionally on the set of real non-negative numbers. For example, the risk in the cell $c_1$ is $k$-times larger than in $c_2$. Then, $\mathrm{RD}(c_1,t) = k \cdot v$ and $\mathrm{RD}(c_2,t) = v$ are chosen correctly for any positive $v$. But the $w_2$ weight, as used in the Equation 3.17, needs to be chosen properly with respect to the actual RD values. Lastly, with linear approach comes another advantage, the values can not only represent detection risk, but also any other inconvenient influences that may deter an object from entering a space.

## 3.3 Problem definition

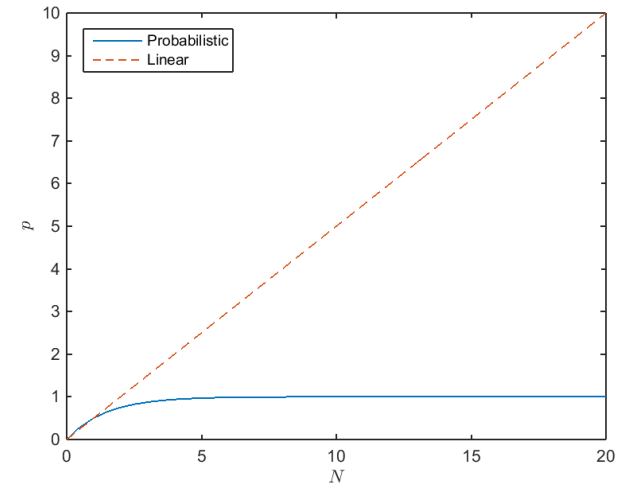
Given the defined terms above, the whole task can be formalized as looking for a sequence of cells $c_1, \ldots, c_{N+1}$ and time windows $t_1, \ldots, t_N$ such that the value of $\mathrm{cost}(c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} \ldots \xrightarrow{t_N} c_{N+1})$ is minimised, $t_1 \leq t_2 \leq \cdots \leq t_N$ and the constraints on movement are satisfied.




(a) $\mathrm{RD}(c,t) = 0.01$




(b) $\mathrm{RD}(c,t) = 0.1$




(c) $\mathrm{RD}(c,t) = 0.5$

Figure 3.4: Comparison of the suggested functions p for variable $N$.

### 3.3.1 Constraints on movement

Constraints originate from the section 1.2. The first limitation is caused by the inability of the submarine to stay submerged indefinitely. As mentioned in 3.2.2, the time is discretised into time windows, therefore the maximum submerged time is also defined by the amount of time windows that the submarine can spend below surface. That results in the condition 3.18, where the maximum time below surface is denoted as $t_{\text{submerged}}$. The condition expresses the fact that for all subsequences of the path in which the submarine is submerged continuously the overall time spent below surface may not be longer than $t_{\text{submerged}}$. If $t_{\text{submerged}} = 0$, then the vessel can move only on surface.

$$\begin{aligned}&\forall \text{ subsequences } c_i \xrightarrow{t_i} \ldots \xrightarrow{t_{j-1}} c_j \text{ of the original path } c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} \ldots \xrightarrow{t_N} c_{N+1} : \\ &c_i^D, \ldots, c_j^D > 0 \implies t_{j-1} - t_i < t_{\text{submerged}}\end{aligned} \tag{3.18}$$

Next, the vessels are also limited by their maximal speed $s$. It means that in each time window, the overall travelled distance is limited to $s \cdot D$, where $D$ is the duration of a TW.

$$\begin{aligned}&\forall \text{ subsequences } c_i \xrightarrow{t} \ldots \xrightarrow{t} c_j \text{ of the original path } c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} \ldots \xrightarrow{t_N} c_{N+1} : \\ &\sum_{k=i}^{j-1} \mathrm{d}(c_k, c_{k+1}) \leq s \cdot D\end{aligned} \tag{3.19}$$

Lastly, it is also necessary to ensure that on each path between two cells, the submarine will not for example hit a seamount[4]. This means that along the route, the actual depth of the submarine may not be higher than the local depth of the sea. This condition is checked by sampling the points along the path and ensuring that the vessel does not crash.

## 3.4 Algorithms

The task defined above can be solved by slightly altering the RRT* algorithm introduced in 2.1.2. In the altered version, each vertex of the tree will not only be defined by its position in the state space, but also by the arrival time to this position. Thus, the function add_vertex$(p, t)$ accepts two arguments, where the first one is the position $p$ and the second one is the time frame $t$ in which the position is reached. The get_random_state() function returns a random cell from the grid chosen with uniform distribution.

To clearly define the usage of the cost function in this situation, if a path of length $N$ consists of positions $p_1, \ldots, p_N$ with assigned time windows $t_1, \ldots, t_N$, the cost of the path equals $\text{cost}(p_1 \xrightarrow{t_2} p_2 \xrightarrow{t_3} \ldots \xrightarrow{t_{N-1}} p_{N-1} \xrightarrow{t_N} p_N)$.

[4] A mountain on the bottom of the ocean that does not reach the surface.

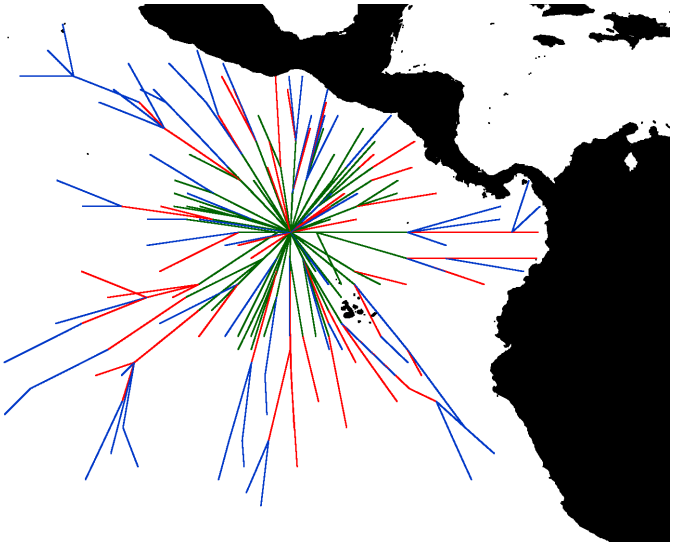


Figure 3.5: A tree that samples the Eastern Pacific. The meaning of the edge colours is described in the Section 4.3.

Four modifications of the RRT* are presented, Gradually Extending RRT* (GERRT*), RRT* with Time as Dimension (RRT*TD) and the corresponding balanced versions. The modifications share the same functions with RRT*, even the whole interface of extending and optimizing the tree is the same, except the additional time information. The common part of the algorithm is extracted in the Algorithm 15 that describes the process of inserting a new node to the tree based on the randomly sampled vertex $p_{\text{random}}$ with arrival in the TW $t$.

**Algorithm 15**: T.extend($p_{\text{random}}$, $t$, $\epsilon$, $d$)

```
p_near ← T.find_nearest(p_random);
p_new ← steer(p_near, p_random, ϵ);
if edge between p_new and p_near is feasible then
    p_best ← T.find_best_parent_in_vicinity(p_new,p_near,d);
    T.add_vertex(p_new, t);
    T.set_parent(p_new, p_best);
    T.optimize(p_new, p_best, d);
end
```

Slight changes are in the semantics of the feasibility checking function and optimizing function, since it is necessary to check the constraints mentioned in 3.3.1, especially the fact that the time windows assigned to vertices in every path in the tree from the root to any leaf need to establish a non-decreasing sequence.

Before the algorithms are presented, it is necessary to state that the constraint on speed significantly influences the node generating process. Let us denote $s$ as the speed of the vessel and $D$ the duration of one time window, as in the Section 3.3.1, then the distance that can be travelled within one time window equals $\theta = s \cdot D$. That results in the fact that a node that is reached in a time window $t \in \{0, \ldots, E\}$ must be closer to the initial position $p_{init}$ than $(t+1) \cdot \theta$. It means that the nodes from a time window $t$ can be distributed in an area proportional to $(t+1)^2 \cdot \theta^2$ if there are no significant irregularities

in the obstacles. This property can be seen in the Figure 3.5, where the distant areas are sampled more sparsely than the areas close to the initial position. That example was generated by GERRT*.

A question arises whether the algorithms should take this fact into consideration and generate more nodes in later time windows to cover the enlarged reachable state space. This results in the balancing aspect. In the unbalanced versions of the algorithms, the distribution of nodes into time windows is uniform, as opposed to the balanced versions, where the amount of nodes in the given TW $t$ depends on the time $t$ and grows quadratically with increasing $t$ to compensate for the larger sampling area.

### 3.4.1 Unbalanced versions

#### Gradually Extending RRT*

In this modification, the tree is first built as in the RRT* with time frame fixed to 0. Then, the tree is expanded with vertices from time frame 1. In this manner, the tree is gradually extended into all the time windows. The overall amount of vertices is evenly divided into the $E + 1$ time windows.

**Algorithm 16**: GERRT*($N$, $p_{\text{init}}$, $\epsilon$, $d$, $E$)

```
T ← empty_tree();
T.add_vertex(p_init, 0);
for t = 0, ..., E do
    for i = 1, ..., ⌊N/(E+1)⌋ do
        p_random ← get_random_state();
        T.extend(p_random, t, ε, d);
    end
end
return T;
```

#### RRT* with Time as Dimension

In RRT*TD, the time is viewed as another dimension and is assigned to vertices randomly by using the rand($\{0, \ldots, E\}$) function that chooses a random integer from the given set with uniform distribution, the same as in the Section 2.1.3.

**Algorithm 17**: RRT*TD($N$, $p_{\text{init}}$, $\epsilon$, $d$, $E$)

```
T ← empty_tree();
T.add_vertex(p_init, 0);
for i = 1, ..., N do
    p_random ← get_random_state();
    t ← rand({0, ..., E});
    T.extend(p_random, t, ε, d);
end
return T;
```

### 3.4.2 Balanced versions

#### Balanced Gradually Extending RRT*

This version is the same as GERRT*, but the total amount $N$ of vertices is not evenly divided into the time windows, but has quadratic distribution. That is, in the time window $t \in \{0, \ldots, E\}$, the amount of nodes is directly proportional to $(t+1)^2$ and can be exactly estimated as

$$\mathrm{N}(t) = \left\lfloor N \frac{(t+1)^2}{\sum_{t=0}^{E} (t+1)^2} \right\rfloor. \tag{3.20}$$

**Algorithm 18**: BGERRT*($N$, $p_{\text{init}}$, $\epsilon$, $d$, $E$)

**for** $t = 0, \ldots, E$ **do**
  $\mathrm{N}(t) \leftarrow \left\lfloor N \frac{(t+1)^2}{\sum_{t=0}^{E} (t+1)^2} \right\rfloor$;
**end**
$\mathsf{T} \leftarrow$ empty_tree();
$\mathsf{T}$.add_vertex($p_{\text{init}}, 0$);
**for** $t = 0, \ldots, E$ **do**
  **for** $i = 1, \ldots,$N($t$) **do**
    $p_{\text{random}} \leftarrow$ get_random_state();
    $\mathsf{T}$.extend($p_{\text{random}}$, $t$, $\epsilon$, $d$);
  **end**
**end**
**return** $\mathsf{T}$;

#### Balanced RRT* with Time as Dimension

Again, the time windows are chosen randomly in BRRT*TD, the distribution is not uniform, but instead follows the discrete distribution defined as

$$\mathrm{P}(t) = \frac{(t+1)^2}{\sum_{t=0}^{E} (t+1)^2} \text{ for } t \in \{0, \ldots, E\}. \tag{3.21}$$

**Algorithm 19**: BRRT*TD($N$, $p_{\text{init}}$, $\epsilon$, $d$, $E$)

**for** $t = 0, \ldots, E$ **do**
  $\mathrm{P}(t) \leftarrow \dfrac{(t+1)^2}{\sum_{t=0}^{E} (t+1)^2}$;
**end**
$\mathsf{T} \leftarrow$ empty_tree();
$\mathsf{T}$.add_vertex($p_{\text{init}}, 0$);
**for** $i = 1, \ldots, N$ **do**
  $p_{\text{random}} \leftarrow$ get_random_state();
  $t \leftarrow$ choose from $\{0, \ldots, E\}$ with distribution $\mathrm{P}(\cdot)$;
  $\mathsf{T}$.extend($p_{\text{random}}, t, \epsilon, d$);
**end**
**return** $\mathsf{T}$;

### 3.4.3 Summary of the algorithms

To summarize, the GE versions generate the nodes in earlier time windows first and after that, the vertices in further time windows are added. This is in contrast with the TD approach, in which the order of the time windows is mixed and generated randomly as well as the position of the vertices.

In the unbalanced versions, there is the same amount of vertices in each time window, as opposed to the balanced versions, where the vertex count in a time window is not uniform. Note that the actual distribution into time windows in the TD versions depends on random choice and is therefore not exact.

# Chapter 4

# Implementation

In this chapter, we first describe the functioning of the already implemented BANDIT simulation framework in the Section 4.1 and a newly created model of submarine is introduced in the Section 4.1.2 as an instance of an agent behaviour model. Then, the input parameters of the grid and path planning are described in detail in the Section 4.2 together with a discussion on their significance and meaning. The forms of the output with examples are described in the Section 4.3. Finally, the usage of the implemented algorithm is explained in 4.4.

## 4.1 BANDIT

BANDIT is an agent-oriented time-stepped simulation framework that was created to model maritime piracy and smuggling. As stated in Hrstka et al. (2015), it might have been one of the first maritime microscopic simulations that cover the individual behaviour and interactions between vessels, since the previous models were located only in limited areas or could not feature heterogeneous agents.

Simulations in this model are run in three steps. First, the actual scenarios are either randomly sampled from given parameter distributions or could be directly set as the input. Then, the scenarios are executed while logging important events and trajectories of the agents. Finally, the events are post-processed and output data are created.

The process of simulation execution in BANDIT interface uses two main components, the environment and the agents. In the environment, the state variables are stored and changed by the commands from the agents. On the other hand, the agents obtain information concerning their surroundings from the environment. The connection between the agents and the environment is mediated by the controller interface.

### 4.1.1 Agent Behaviour Model

The agent behaviour model is based on behaviour state machines (BSM) that are similar to hierarchical finite-state machines (hFSM). Thus, to introduce the BSM, we start with explaining finite-state machines (FSM).

The FSM is a mathematical model used to describe the functioning of a process. At given time, a FSM is in one of its states of which there is only finite amount. It is able to change its state through transitions that are triggered by events or stimuli from the outside. A FSM can be defined by an oriented graph where the vertices represent the states and the oriented edges represent the transitions. The output of a FSM can either be produced by the transition in the case of a Mealy machine or by the the individual states, as in Moore machines.

Like FSM, the BSM can be defined by a graph too, but unlike FSM, it features more complex states, i.e., the transition conditions (called guards) can be arbitrarily complicated. And if a transition guard passes, meaning that the condition is satisfied, it switches the state and can produce some output. Otherwise, the internal response guards are evaluated and if one of them is passed, it can also produce an output without switching the state.

The hierarchical property comes into play when no guard, either transition or internal response, is passed. Then, the trigger event is recursively applied on the BSM that is contained in the current state, if it is there. It is also possible to produce additional outputs when a state is exited and another entered. The functioning is described in more detail in Hrstka et al. (2015). On the outside, the BSM functions as a black box that is given inputs and reacts to them by creating outputs, it can not produce any output without a trigger from the outside.

### 4.1.2 The model of a submarine

In our case, the submarine in the simulation executes a voyage that is planned by one of the algorithms in 3.4. That is a sequence of cells $c_1, \ldots, c_N$ and time windows $t_2, \ldots, t_N$ meaning that a cell $c_i$ is reached in time window $t_i$.[1] It may be beneficial for the submarine to not only drive straight to a target, but to wait until for example some kind of risk disappears and then continue with the voyage.

Thus, in the implementation, the sequence of cells and time windows is converted to a sequence of locations $p_1, \ldots, p_K$ and boolean values $b_1, \ldots, b_{K-1}$. The locations $p_i$ contain not only the coordinates, but also the depth information. This can be easily interpreted by stating that the agent sails through the locations one by one and when it reaches the location $p_i$ and $b_i$ is true, then it waits until the next time window to continue to the next position. Otherwise, if $b_i$ is false, the agent continues to the next position $p_{i+1}$. The last value $b_K$ is not necessary, because when the agent reaches $p_K$, it finishes its route and does not continue anyway. Also note one implementation detail, the new sequence of locations can be longer, i.e. $K \geq N$, because if the agent needs to wait at one place for more time windows, it could not be described by the single boolean value. Thus, the situation is solved by repeating the same location in the sequence and setting multiple corresponding $b_i$ values to true. This results in the desired behaviour in which the agent stays at one place for multiple time windows.

The behaviour defined above is implemented in the simulation by three states, as seen in the Figure 4.1. The implementation is not straightforward, since the agents in BANDIT

[1] Let us emphasise that the time window $t_1$ was not mentioned, because the agent is placed to $c_1$ in the time window 0 even if it would not move until the next time window.

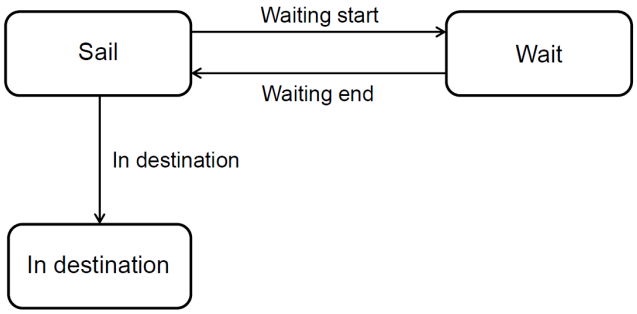



Figure 4.1: The state diagram of a submarine, as used in the simulation.

simulation can react only to events or alarms. Therefore, the submarine agent switches its state by alarms that it arranges itself. The same approach was used in the simulation framework to create the tarping behaviour[2].

The agent starts at the position $p_1$, is in the sail state and sets an alarm to end waiting in time $D$, where $D$ is the duration of one time window.

The path is then executed iteratively. The agent is at location $p_i$ and if $b_i$ is not true, the agent continues to $p_{i+1}$. Otherwise, if $b_i$ holds, then the agent switches to the wait state, where it stays until the waiting-end alarm returns it to the sail state and the agent continues to $p_{i+1}$. Right after the waiting ends, a new waiting-end alarm is set up to the time $t + D$, where $t$ is the current time. This process is iteratively repeated until the final location $p_K$ is reached and the agent switches into the in destination state.

## 4.2 Inputs of the path planning process

The inputs and settings of the algorithm can be roughly divided into two groups. The first group concerns the properties of the grid and the second contains the settings of the algorithms presented in the Section 3.4.

### 4.2.1 Grid parameters

The parameters of the grid were already introduced in the Section 3.1.1. Now the meaning of the parameters is explained in practice and we also discuss the choice of values.

The position of the area has clear meaning that does not need to be explained. However, it may be beneficial not to limit the planning area to a tight vicinity of the connecting line of the start and target locations, because the optimal paths may be located outside of this area, for instance to avoid regions with high risk of detection.

The size $s$ of the cells, as illustrated in the Figure 3.2a, determines the particularity of the grid. It is usually specified in kilometres and the main disadvantage caused by too many small cells would be the overall slowdown in the planning phase, as further elaborated

[2]In the tarping model, a go-fast boat travels at night and stops to be carried by the currents without interference during the day. The state is switched by alarms at sunrise and sunset that are scheduled by the agent itself.

in 5.2.1. Of course, the amount of cells is also limited by the memory of the computer that runs the simulation. For example, a grid that covers the whole Pacific ocean with $s = 100$ km consists of 32270 cells in the surface layer. Having more layers would multiply the overall amount of cells.

The depth interval $\Delta d$ determines the vertical size of the cells, as illustrated in the Figure 3.2b. It should be chosen large enough to provide significant physical difference in characteristics. That means, appropriate amount of layers is in the order of ones, not even tens. That results in the values of $\Delta d$ in tens of metres. For example, it is important to think of the fact that only the first layer is considered to be on the surface, as it is further explained below in the paragraphs about detectors. Too dense depth sampling would invalidate this idea from the physical point of view.

The depth $m$ of the deepest layer should be chosen with accordance to the physical abilities of the modelled submarine and current tactical situation. This means that in peacetime, submarines are allowed to maximally dive in the depths equal approximately to the half of their design depth.

To properly create the grid, we also need the RD function defined in 3.2.4. For user convenience, the risk of detection does not need to be estimated for each cell separately, but it is suitable to introduce detectors that automatically set up these values for cells in defined areas.

In the enclosed implementation, there are two basic types of detectors, a uniform detector and a Gaussian detector. The uniform detector is defined by its centre $c$, radius $r$ and risk value $v$ and increases the risk of detection by $v$ in each cell that is closer to $c$ than the distance $r$.

The Gaussian detector is defined by its centre $c$, standard deviation $\sigma$ and the value at centre $v$. Then, the risk of detection in a cell that is in the distance $d$ from $c$ is increased by the value $I$ defined as

$$I = v \cdot \exp\left( - \frac{d^2}{2\sigma^2} \right). \tag{4.1}$$

To improve the versatility of the model, the depth also influences the risk of detection. There are three options that model the effects.

- First, there does not need to be any depth influence and the risk only depends on the horizontal distance from the detector.
- Second, the detector is active only on the surface layer and can not detect anything submerged.
- Third, the risk of detection exponentially decreases with increasing depth of the cells. The parameters of the decrease are defined by one depth and corresponding percentage decrease, the other values are derived from these data.

Each detector is also assigned the interval of time windows in which it is active, allowing time-variant environment. Outside the interval, it does not influence the risk of detection.

The total risk of detection of a cell in given time window is defined as a sum of all the risks given by the currently active Gaussian and uniform detectors. If no detectors reach the cell in the defined time window, its RD value is set to 0. This can be done due to the fact that we chose the linear approach that was introduced and justified in the Section 3.2.4.

Lastly, we also need a depth information provider to properly create the grid according to the real world situation. For this, the ETOPO2v2 grid[3] with depth data was used and is necessary to generate the grid.

### 4.2.2 Planning algorithm parameters

In this section, the meaning and importance of the input parameters previously mentioned in the Sections 2.1.2, 3.3.1 and 3.4 is discussed.

The total amount of vertices $N$ should be in the order of hundreds or low thousands, unless really short paths are planned. If the path goes through complicated environment with a lot of obstacles, it is advisable that more vertices are generated in order to at least create a feasible path. The parameter $N$ should also be larger if there are more time windows. The computation time grows asymptotically quadratically with $N$ due to the tree optimization processes.

The end time window $E$ with the duration of a time window $D$ together define the total length of the simulation as $(E + 1)D$ (as defined in the Section 3.2.2, there are time windows $\{0, \ldots, E\}$). That is the time that the submarine has to finish the voyage. In the current implementation, there should not be too much time windows, for instance five, because the overall amount of vertices $N$ is divided into the time windows and large $E$ would cause the state space to be only sparsely sampled. And as described above, $N$ can not be increased without additional demands on the computation time.

The issue concerning the tightly bounded amount of time windows can be solved in the same way as in ABC, in the Section 2.1.3. It means that the whole voyage will not be planned at once, but only piecewise for the nearest future, evaluated and then the time windows will be shifted to predict further based on the previous result.

The steering distance $\epsilon$, as introduced in the Algorithm 1, has definitely to be larger than $s$ to allow the algorithm to function and properly expand the nodes. It is generally recommendable to choose it as a low multiple of $s$.

The optimizing radius $d$ with the amount of nodes $N$ are the parameters that most directly influence the quality of the solution. The value of $d$ must be chosen larger than $\epsilon$, for example a small multiple, i.e. $5\epsilon$. If the value was chosen to be too large, for instance equal to the circumference of the Earth, the algorithm will be unnecessarily slowed down and still may not produce the optimal solution, because it is based on random sampling.

[3]Specifically `ETOPO2v2c_f4_MSB.flt` that was taken from `https://www.ngdc.noaa.gov/mgg/global/relief/ETOPO2/ETOPO2v2-2006/ETOPO2v2c/raw_binary/`

The start and target points of the voyage $p_{\text{init}}$ and $p_{\text{target}}$ are given as latitude and longitude coordinates. The submarine always starts and finishes on surface in the cells that are nearest to the given coordinates.

These were the parameters of the planning algorithm in general, but additional domain-specific knowledge concerning the particular vessel is needed and listed below.

The next parameters are the preferences of the captain of the vessel, previously defined as $w_1$, $w_2$ and $w_3$ in the Section 3.2.3. These non-negative values are the weights of distance, risk of detection and time spent by the voyage in this specific order. The numerical value of distance is given in kilometres and the time as a time window. The values of RD were broadly discussed in the Section 3.2.4. If any of these weights are zero, it means that the particular factor does not matter to the planning.

If we focus on the first two components, the values mean that a submarine would consider 1 kilometre detour and increasing RD by $\frac{w_1}{w_2}$ equally costly (if $w_2 \neq 0$). This defines how long detour the vessel is willing to make to avoid detection. The particular choice of $w_2$ also depends on the values that the detectors have and the size of cells $s$, meaning that with large cells, the RD values or $w_2$ value should be scaled accordingly to keep the intended behaviour.

The weight of time $w_3$ defines how much the vessel prefers to wait on its path. For example, it may be favourable to wait until a detector disappears in the next time window rather than to make a detour or cross an area with high RD. Nevertheless, the absolute value of $w_3$ should be significantly higher than $w_1$ in order to scale between thousands (kilometres, length of the path) and small ones (time windows).

The time $t_{\text{submerged}}$ that defines how long the vessel can stay submerged is specified as some amount of time windows. This parameter is set according to the design of the vessel. If $t_{\text{submerged}} = 0$, then the vessel can not submerge and the voyage is planned only on the surface. In such case, it is recommendable to generate a grid only with the surface layer of cells to increase performance.

The last input is the distance $\theta$ that the modelled vessel can travel in one time window, respectively in the time $D$. This is the real input to the algorithm, as opposed to the speed mentioned in 3.3.1. However, the relation between the speed and the travelled distance can be calculated simply.

### 4.2.3 Parameter tuning

As the reader is already aware of, there is a lot of freedom in the choice of parameters. That might sometimes result in inappropriate settings and unsatisfactory behaviour of the algorithm. However, there is a form of assistance to the parameter tuning. One of the outputs of the algorithm is also a map depicting the generated tree.

This output is crucial when the planned paths do not seem high-quality. In this case, it should be looked at to properly choose the parameters. In the parameter estimation phase, strictly one tree should be created and according to the looks of the tree and the found paths, the inputs can be altered in order to obtain better results. For example, if the tree is too sparse, the amount of vertices $N$ should be slightly increased. Or, if the paths

are not very smooth, it may be applicable to increase the optimization radius $d$ while paying attention to the fact that both these modifications result in longer computation time. Other issues may be too low $E$ and $\theta$ that make the planned path impossible due to its length. When the output seems all right, the amount of generated trees should be increased to more than one.

Before the input section is concluded, we stress once more that it is necessary to ensure that $s < \epsilon < d$. Otherwise, the algorithm fails to produce any reasonable output.

## 4.3 Output

After the grid is generated and parameters for planning are set, the algorithm can be employed to generate multiple trees and pick given amount of the best paths from each one of them. Building a tree is considerably more time consuming than picking one more path from an already existing tree. However, choosing more than a few paths (2-3) from one tree creates bias, because the best paths from one tree usually share common parts. Also, umpteenth best path will likely be of poor quality and barely feasible.

The chosen paths are then simulated in the BANDIT framework and all the KML outputs that the framework can produce are therefore obtainable. The default output from the GUI is a heat map that aggregates the trajectories in all the paths and can also present them in varying time. In the heat map that can be visualised in Google Earth, the colours represent how many paths crossed the particular area and how much time the agents spent there with red being the most visited and green the least visited space.

During the implementation of this project, we also created an output that was mainly for the purposes of debugging, but is also quite useful when considering the quality of the planned paths. It is a map that features the planned paths with the time windows distinguished by colour. The intensity of RD is also shown on this map by the intensity of the yellow colour, but without the time window distinction. It is also produced as a secondary result of the planning together with the depiction of the generated tree, as already mentioned in the Section 4.2.3.

An example output was created using the settings listed in B.1. All of the possible outputs of the program are presented in this paragraph. In the Figure 4.2, there is a screenshot of the heat map as shown by Google Earth, then in the Figure 4.3, a part of the map with the planned paths is drawn. The yellow area represents the detector range, the particular parts of paths finished in time window 0 are green and the parts executed in the time window 1 are red. The last output is the Figure 4.4; there is a part of the map depicting the first of the 3 trees. The colours of the drawn paths depend on the time windows, the parts of paths that are executed in the time window 0 (respectively 1, 2, 3 and 4) are drawn with green colour (respectively red, blue, black and turquoise). In this example, only the first three TWs were used.

In the examples, we can see that the agent prefers to go around the detector because of the high $w_2$ weight. It is also worth noting that the input setting of this particular computation was to generate 3 trees and take 2 best paths from each of them. However,

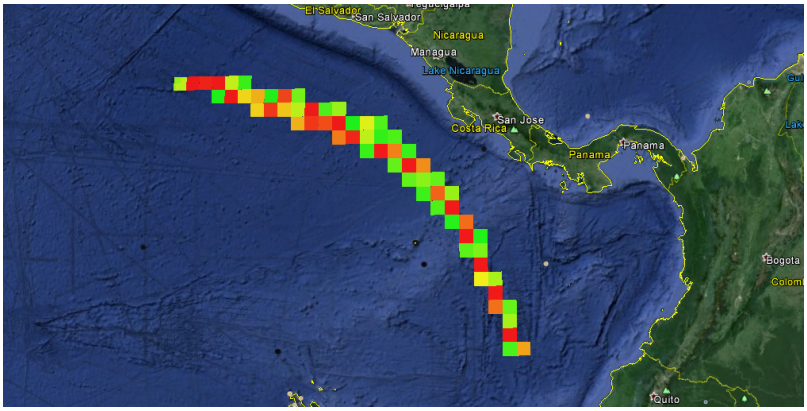


Figure 4.2: The screenshot of the heat map visualised by Google Earth.

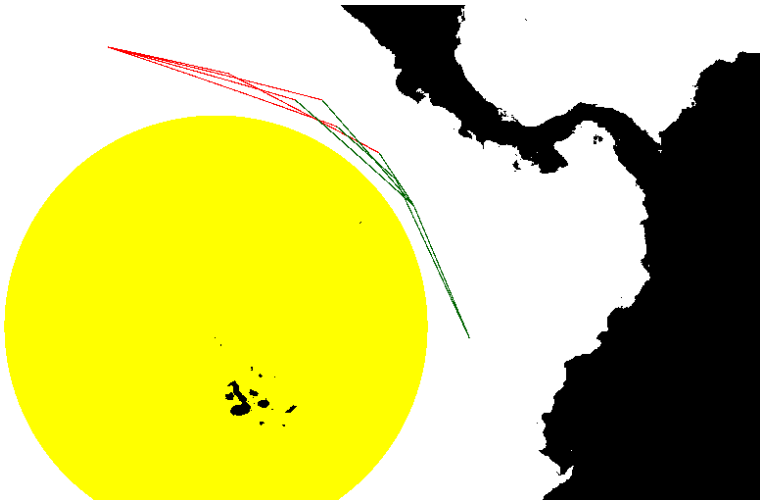


Figure 4.3: The paths drawn on a map by the debug output.

only one of the trees was able to produce two feasible trajectories, therefore there are only 4 paths in the Figure 4.3.

## 4.4 Use in practice

The implemented algorithm can be used for both planning or trajectory prediction of submarines. If we would like to plan the optimal path for a particular vessel, the parameters described in the Section 4.2 will be set up as well as the detectors and the algorithm will offer some paths in accordance with the given preferences. This approach is fairly straightforward.

In addition, the algorithm can be also used to predict the path of an adversary in a similar manner. Then, the input will be the same, except that only the detectors that the adversary is aware of are included. Another difference might be that we do not know the whereabouts concerning the goals or targets. Thus, expert knowledge needs to be utilised to estimate them and then let the algorithm predict possible alternatives. The assets then could be moved in order to detect the adversary in the predicted area and time.

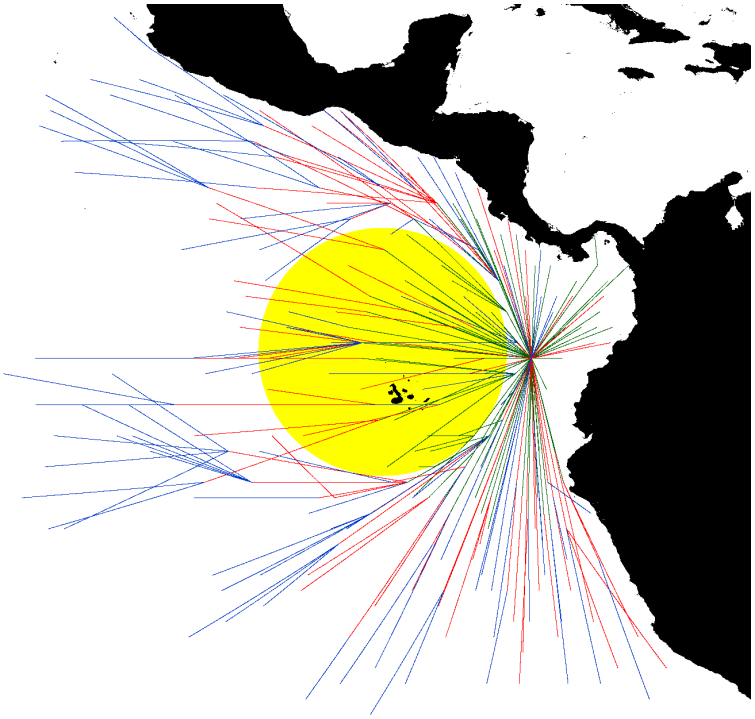


Figure 4.4: The tree from which one of the paths was taken.

# Chapter 5

# Evaluation

In this chapter, we compare the four presented algorithms in various situations and from different aspects in the Section 5.1. Then, the best performing method is analysed in the Section 5.2 and tested in detail in the Section 5.3.

## 5.1 Comparison of the algorithms

First, we would like to choose the best planning algorithm that could be used in actual situations. We will compare the alternatives introduced in the Section 3.4 in the terms of their time complexity and the quality of the found solutions.

### 5.1.1 Experiment 1

In simple input scenarios with few time windows (less than 4) and not many obstacles, the results are usually more or less similar. That is why we created more complicated scenarios, where the planner needs to avoid multiple detectors along the path. The scenario used in this experiment is defined in the appendix B.2 in detail.

| Algorithm | Cost of the best path (rounded, in thousands) | | | | | | | | Average cost |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
| RRT*TD | 13.6 | 10.9 | 14.0 | 10.4 | 10.6 | 18.3 | 38.2 | 10.3 | 15.8 |
| BRRT*TD | 27.4 | 11.7 | 19.1 | 71.7 | 24.5 | 17.8 | 60.5 | 53.1 | 35.7 |
| GERRT* | 10.1 | 10.7 | 11.3 | 10.3 | 10.3 | 10.0 | 10.4 | 10.7 | 10.5 |
| BGERRT* | 10.0 | 11.4 | 10.1 | 19.8 | 10.5 | 10.5 | 11.9 | 10.5 | 11.8 |

Table 5.1: Comparison of the algorithms in the terms of cost of the best path.

In the Table 5.1 and the Figure 5.1, the costs of the best paths in 32 repeated experiments with the same settings are shown. The GERRT* provided fairly stable results and the balanced version BGERRT* only once produced a low-quality solution, that was caused by exceptionally sparse sampling of the nodes in the target area.

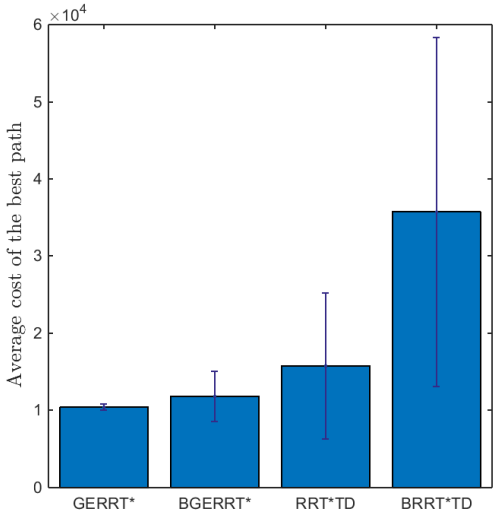



Figure 5.1: Average costs and their standard deviations.

On the other hand, it can be seen that the quality of the paths produced by the TD variants fluctuates substantially, even though these algorithms can provide a good solution, for example the fourth experiment with RRT*TD or the second in BRRT*TD, but on the individual results can not be relied. The reason for this is that the TD methods rely on randomness substantially and may generate a node in a later time window even as one of the first nodes and then the node is forced to be in the close vicinity of the initial node. This results in wasting many nodes in the surroundings of the initial node, even though they could be positioned further if they were produced later.

The GE methods do not suffer from this disadvantage because in GE methods, when the nodes from a time window $t$ are generated, the nodes in all the previous time windows had already been created. Thus, the nodes can connect to any of the previous nodes and actually expand in the full radius $(t+1) \cdot \theta$, as explained in 3.4.

| Algorithm | Average time (seconds) |
|---|---|
| RRT*TD | 44.7 |
| BRRT*TD | 40.7 |
| GERRT* | 87.1 |
| BGERRT* | 35.2 |

Table 5.2: Comparison of the algorithms in the terms of execution time.

In the same settings B.2, the time of computation for each method was measured[1] with the results listed in the Table 5.2. The time needed for the execution of the GERRT* method exceeds the other times approximately twice. Even though BGERRT* exhibited a fluctuation in the quality of the results, it has appreciably lower execution time. To compare the balanced and unbalanced versions fairly, the number of generated vertices will be taken into consideration in order to equalize the computation time in the next experiment.

[1]Experiments were performed on a computer with Intel Core i5-3210M CPU and 8 GB RAM.

### 5.1.2 Experiment 2

The results in the Tables 5.3 and 5.4 were generated with the settings B.3 and were run 6 times for each method and node count. A slight alteration of the previous settings was made by prolonging the active time of the detectors. This change however increases the requirements on the result significantly because the trajectory has to avoid multiple detectors that are in the way for the last 3 time windows. To offer more freedom of movement to the planner, the maximal travelled distance $\theta$ in one time window was increased.

| Algorithm | $N$ | Costs in thousands | | | | | | Average cost (thousands) |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | |
| RRT*TD | 3000 | 15.51 | 15.52 | 15.58 | 15.50 | 15.58 | 15.57 | 15.54 |
| RRT*TD | 3750 | 15.54 | 15.68 | 15.58 | 15.52 | 15.53 | 15.52 | 15.56 |
| BRRT*TD | 3000 | 15.52 | 17.45 | 15.86 | 16.93 | 15.86 | 17.59 | 16.54 |
| GERRT* | 2200 | 15.64 | 15.68 | 15.53 | 15.55 | 16.05 | 15.85 | 15.72 |
| GERRT* | 3000 | 15.48 | 15.51 | 15.52 | 15.39 | 15.47 | 15.43 | 15.47 |
| BGERRT* | 3000 | 15.52 | 15.61 | 16.38 | 15.58 | 15.66 | 15.61 | 15.73 |
| BGERRT* | 3800 | 15.54 | 15.51 | 15.60 | 16.11 | 15.56 | 15.56 | 15.65 |

Table 5.3: Comparison of the best cost of the methods with varying node count.

| Algorithm | $N$ | Average time (seconds) |
|---|---|---|
| RRT*TD | 3000 | 129.7 |
| RRT*TD | 3750 | 194.6 |
| BRRT*TD | 3000 | 113.0 |
| GERRT* | 2200 | 109.9 |
| GERRT* | 3000 | 193.1 |
| BGERRT* | 3000 | 110.0 |
| BGERRT* | 3800 | 192.6 |

Table 5.4: Comparison of the execution time of the methods with varying node count.

If we first look at the results with the same node count $N = 3000$ in the Table 5.3 or in the Figure 5.2, GERRT* provided more consistent costs than BGERRT* that on average had higher cost values and again produced a distinctive outlier with obviously sub-optimal value in the third run. Only the worst value produced by the unbalanced version (15522.5 in the third run) is subtly higher than the best value of the balanced version (15518.9 in the first run). Therefore we can conclude that with the same amount of nodes, the unbalanced GERRT* performs better in the terms of the quality of solution than BGERRT* under the same node count.

Now, we will discuss the TD versions. The costs of the best paths provided by the BRRT*TD again fluctuated as in the previous settings and this method is therefore the least credible. On the other hand, thanks to the increased amount of nodes in this scenario, RRT*TD did not produce any outliers this time and offered only slightly worse solutions than GERRT* with the same node count $N = 3000$.

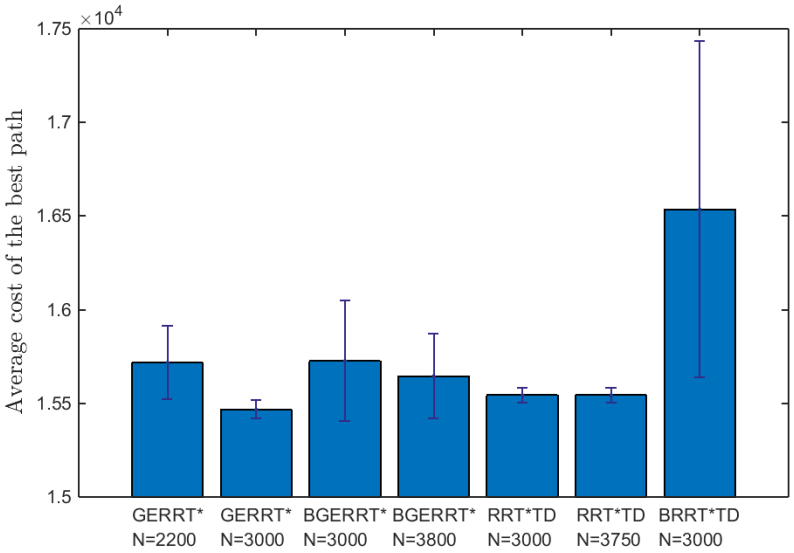



Figure 5.2: Average cost values with standard deviations in the second experiment.

The paths that were generated are shown in the Figure 5.3 for each method. To properly understand the results, we remind that in these settings, the four detectors are active from the time window 2 to the final time window 4. The parts of the paths that are executed in the time window 0 (respectively 1, 2, 3 and 4) are drawn with green colour (respectively red, blue, black and turquoise). The weight of risk of detection was set high, therefore the paths avoid crossing the detectors, except for the green and red parts, where they are not active yet. The distance at which the detectors are detoured is found as the balance between the travelled distance and detection risk. If the weight of detection risk in the cost function was set lower, the paths would be shorter and more eastward. All algorithms provided paths that manage to get across the uniform detectors before they become active.

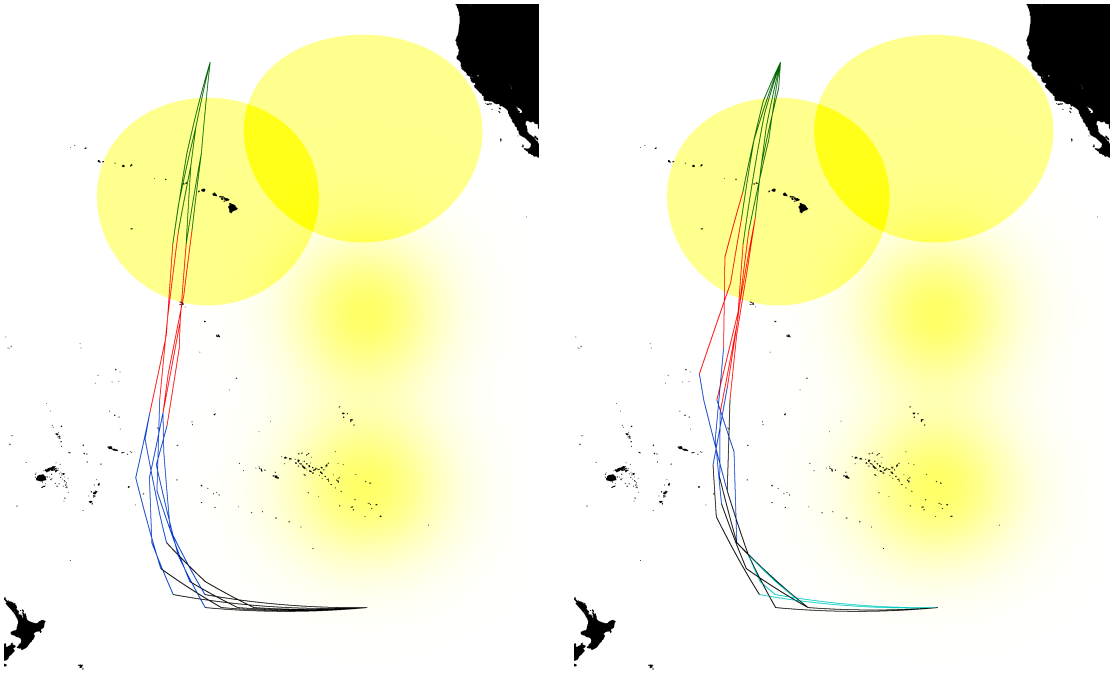
At first sight, the balanced methods do not provide consistent results, especially in the case of BRRT*TD. The most stable method is GERRT* that even managed to finish all of the 6 paths in the time window 3, as opposed to RRT*TD that needed to use the time window 4 to finish 3 of the 6 paths.

However, we are also interested in the runtime that is needed to obtain these results. The execution times do not vary significantly, so there is only the average time to each method and node count in the Table 5.4. It can be seen there that the average time needed by the unbalanced version was considerably longer than the average time of the balanced version when both of them were executed with $N = 3000$.[2]

That is why GERRT* was run also with $N$ lowered to 2200, so that the average time would be approximately the same as with BGERRT* with $N = 3000$. In this case, the average costs of the methods are roughly the same.

Vice versa, the balanced version was also executed with the node count increased to $N = 3800$ in order to equalize the time needed by GERRT* with $N = 3000$. Unlike the previous case, the costs of the unbalanced version tended to be lower or similar.

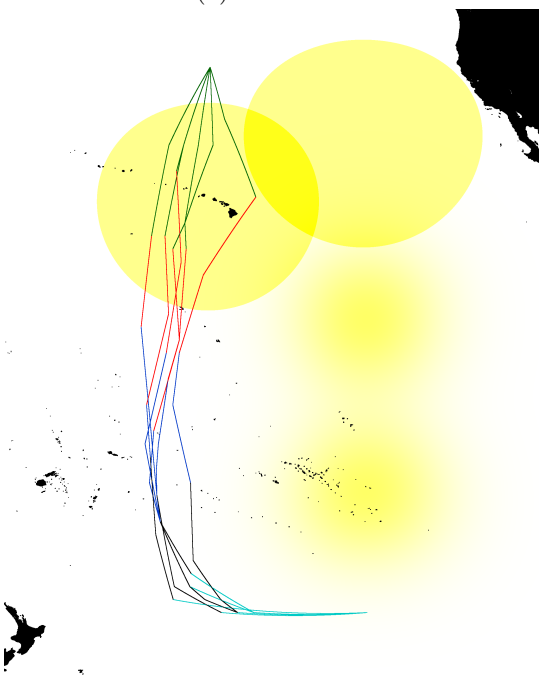
[2]This behaviour of the algorithm is explained in the Section 5.2.3.



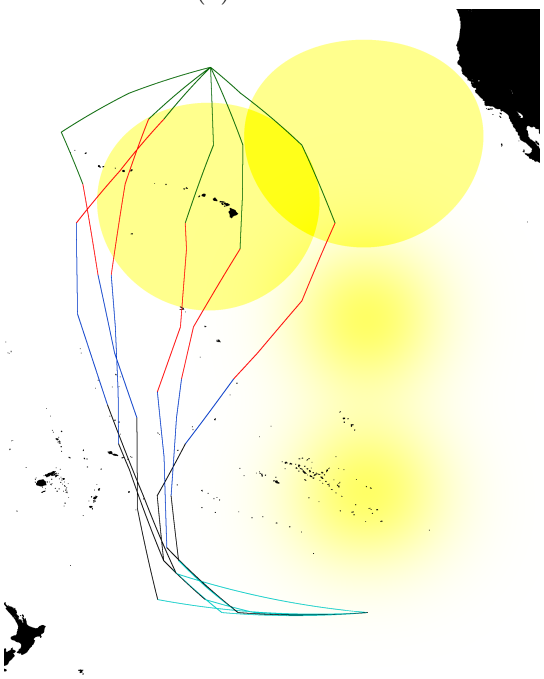
(a) GERRT*



(b) RRT*TD



(c) BGERRT*



(d) BRRT*TD

Figure 5.3: Comparison of the best paths produced by 6 repetitive runs of each algorithm.

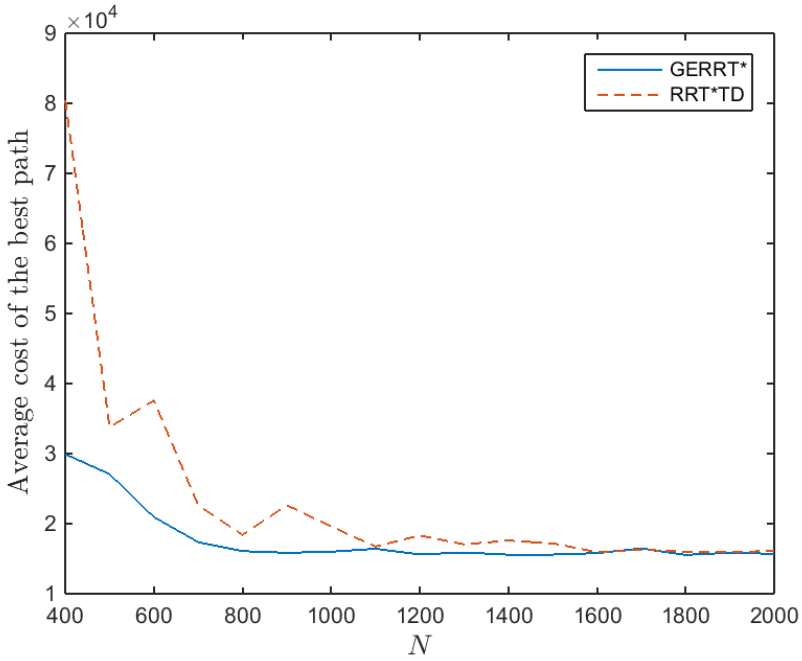



Figure 5.4: Dependence of the average cost of the best path on amount of nodes $N$.

Similarly, the amount of nodes for RRT*TD was increased to $N = 3750$ to offer the same execution time to both RRT*TD and GERRT*. We can not claim that the quality of the best paths increased with larger $N$ because the average cost actually slightly grew when compared to the settings with lower $N$. This only confirms that the results of the TD versions depend on chance heavily.

### 5.1.3 Experiment 3

In the previous result, the unbalanced versions had the best average cost of paths. Also, those were the methods that produced the paths with the smallest costs. The fact that the average cost of the paths produced by RRT*TD did not improve with increased $N$ asks for further evaluation. Therefore, both GERRT* and RRT*TD were run with the value of $N$ ranging from 400 to 2000 on the settings B.3. For each value, the cost of the best path and the execution time were recorded. Each algorithm was employed three times. The averaged results are shown in the Figures 5.4 and 5.5.

In the Figure 5.4, we see that the GERRT* stably reaches the near-optimal solution already with 800 nodes, unlike RRT*TD that does not steadily reach the values near optimum until $N = 1600$. Obviously, the GERRT* performs better or similarly for any amount of nodes in the pictured interval. Next, another difference between the methods is that the average cost produced by GERRT* improves almost monotonously in comparison with the results of RRT*TD that are so unstable with low values of $N$ that the averaging did not manage to clear it.

On the other hand, the Figure 5.5 manifestly demonstrates that the execution time of GERRT* is always longer than with RRT*TD. The time difference is not constant and increases with $N$. This is likely the only definite advantage of RRT*TD over GERRT*.

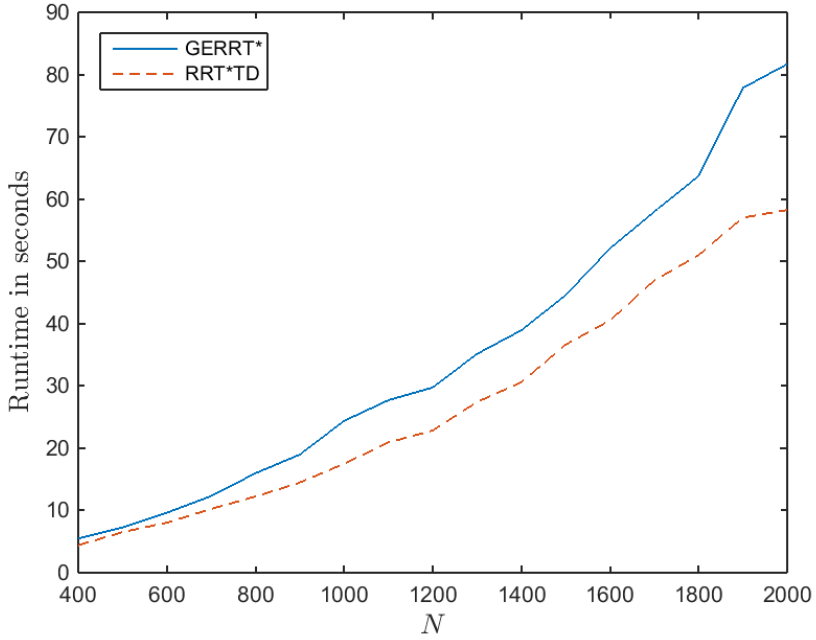



Figure 5.5: The average execution time of the algorithms for given $N$.

### 5.1.4 Summary

As we have seen in the previous experiments, it is clear that GERRT* provided the most stable and high-quality results. The balanced version, BGERRT*, might be considered as a possible alternative if the runtime of GERRT* was too high, even though the algorithm needs to be employed at least twice to make sure that the result was not an outlying solution far from the optimum.

The RRT*TD also needs to be applied to a single task repeatedly to make sure that the found solution is good. Another approach is to increase the amount of generated nodes, that reduces the influence of randomness and might lead to a better solution. The BRRT*TD is sometimes able to produce near-optimal solutions, but generally provides the worst results. For all the reasons mentioned above, we will use exclusively GERRT* in the next sections.

## 5.2 Time complexity

In this section, we present and discuss the influence of the input parameters on the time required for the execution. The interest lies mainly in the amount of generated nodes $N$, cell size $s$ and optimization radius $d$, as the other parameters either do not directly influence the computational time or are a priori known.

The experiments in this section will be again done using the settings B.3 with $N = 2000$. In the Section 5.2.1, the cell size $s$ varies and in the Section 5.2.2, the optimization radius $d$ is changed to show the differences.

We started talking about this topic already in the previous part, where we showed in the Figure 5.5 that the execution time grows approximately quadratically with the amount of nodes $N$ in the tree.

### 5.2.1 Cell size

The size of cells has a major role in the time spent to generate the tree. The reason for this is the need to create lists of cells on given path that are filtered by the algorithm presented in the Section 3.2.1 and further use the lists to assign a cost value to the path.

Elementary intuition suggests that the complexity should grow linearly with the overall amount of cells in the grid. This consideration is proven to be correct by the results shown in the Figure 5.6. We experimented with values of $s$ between 100 and 500 kilometres, these choices are all right with the condition $s < \epsilon = 600$ kilometres and generally correct when compared to the planned path that is approximately 10 000 kilometres long.

In the Figure 5.6a, the dependence of the runtime on cell size $s$ is pictured. The execution time decreases similarly as the amount of cells in the grid with increasing $s$ in the Figure 5.6c. This confirms our claim that the runtime in fact grows linearly with the amount of cells in the grid. This is also plotted in the Figure 5.6b, where the dependence is clearly linear.

Furthermore, we successfully approximated the actual amounts of cells in the Figure 5.6c by the least squares method with one basis function $s^{-2}$. The approximation precisely covered the graph on the whole interval with relative approximation error lower than 2.5 %, as shown in the Figure 5.6d. The approximation itself is not shown because on this scale, the plot exactly overlaps the the amount of cells and can not be distinguished by sight.

Of course, the amount of cells in the grid not only depends on the cell size $s$, but also on how many layers there are. In this case, there was only one layer because the maximum depth $m$ was smaller than the depth sampling interval $\Delta d$. If there were cells in $L$ layers, their overall amount would be approximately $L$-times larger than in the single-layer case and thus the computation would be $L$-times slower.

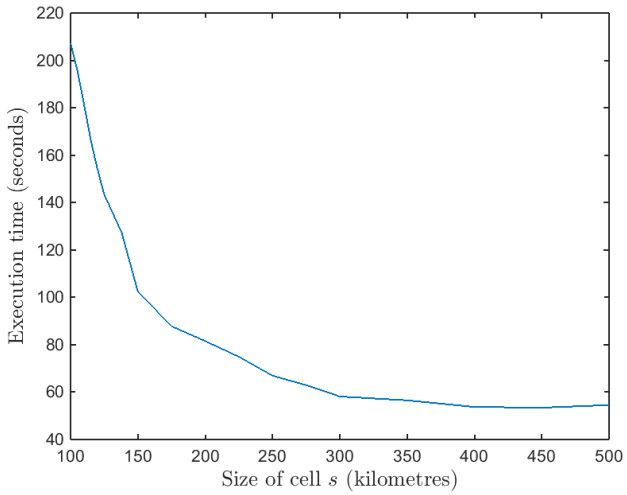
In conclusion, we can say that the amount of cells $M$ in the grid is directly proportional to the number of layers $L$ and inversely proportional to the squared cell size $s^2$, formally

$$M \propto \frac{L}{s^2}. \tag{5.1}$$

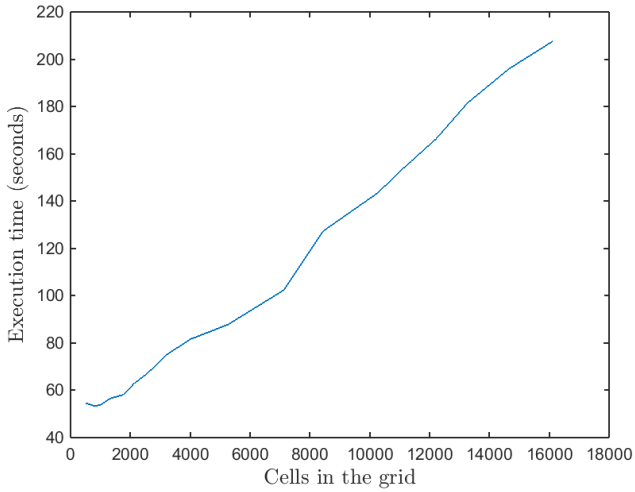
The runtime is then directly proportional to $M$.

### 5.2.2 The optimization radius

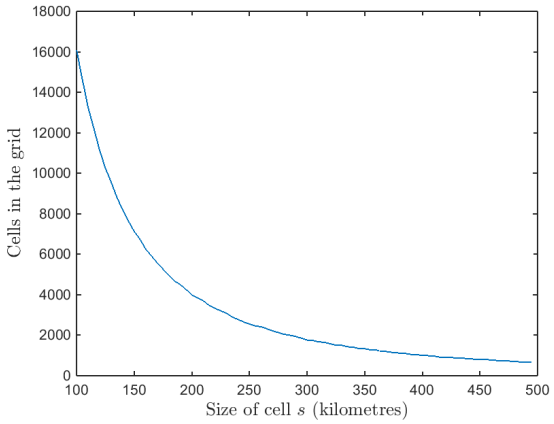
The last parameter that we discuss in terms of runtime is the optimization radius $d$. As used in the Algorithms 3 and 4, the parameter defines the size of the region around a newly created node in the tree, where the optimization processes are employed. The first



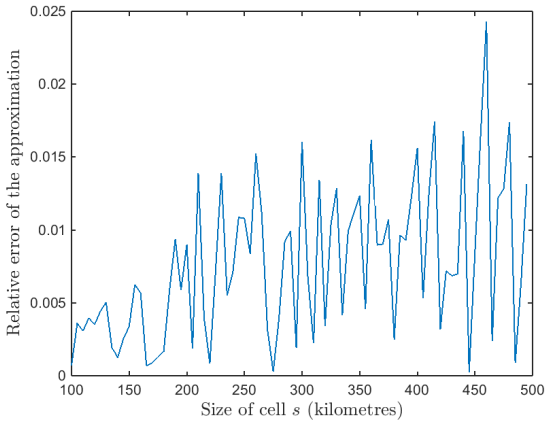

(a) The dependence of the runtime on $s$.




(b) The dependence of the runtime on the amount of cells in the grid.




(c) The amount of cells in the grid with respect to their size $s$.




(d) The approximation error of the amount of cells.

Figure 5.6: Evaluation of the influence of the cell size $s$ on the execution time.

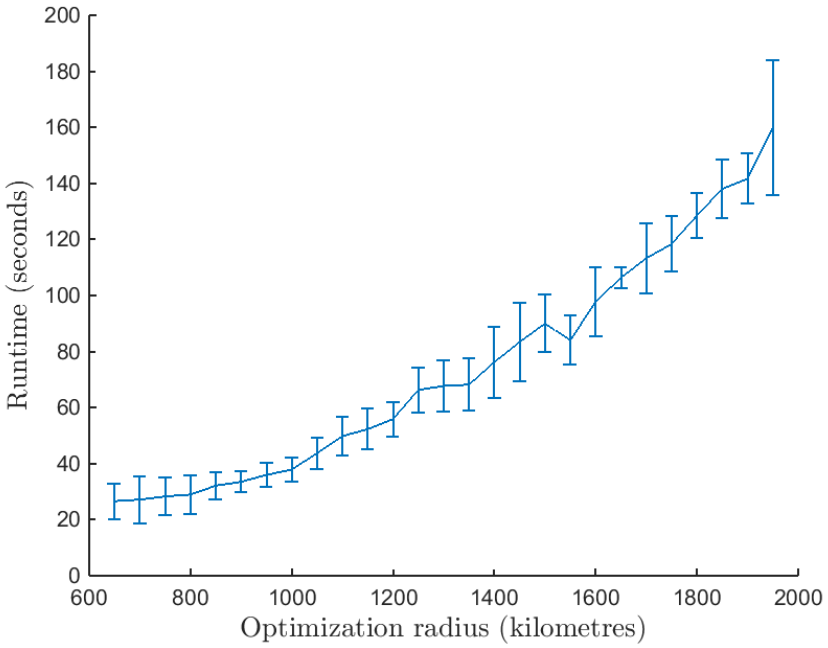


Figure 5.7: The average runtime for various optimization radii.

of them is looking for the parent node in the region that will provide the lowest cost value for the new node. Secondly, after the new node is joined to the tree via an edge to the parent node, all the other nodes in the vicinity are tested whether setting the new node as their parent would lower their cost.

Because the area of the region equals $\pi \cdot d^2$, the computational time should grow quadratically with $d$, if the nodes in the tree are sampled with uniform density around the area. The density is not precisely uniform over the whole state space because it is usually more sparse in the parts of the tree that are more distant from the root. Nevertheless, we just need to look at the small individual regions with size in the order of $d^2$, where the density actually is locally uniform.

Our prediction is confirmed by the Figure 5.7, where the average runtime for radii $d$ from 650 to 1950 kilometres are shown. For each value of $d$, the time was measured 6 times. There were non negligible deviations in the measured durations during the testing, therefore the standard deviation is also pictured in the plot for each value.

### 5.2.3 Analysis of a single execution

Until now, we dealt only with the outputs of the chosen methods, specifically the total runtime or cost of the solution and did not look inside the method. But the timing within the scope of one execution of the method is also interesting. To produce the output in the Figure 5.8, the time was measured from the initialization and each node was assigned the time when it was generated. This process was run 6 times and the results were averaged in order to produce a smooth graph.

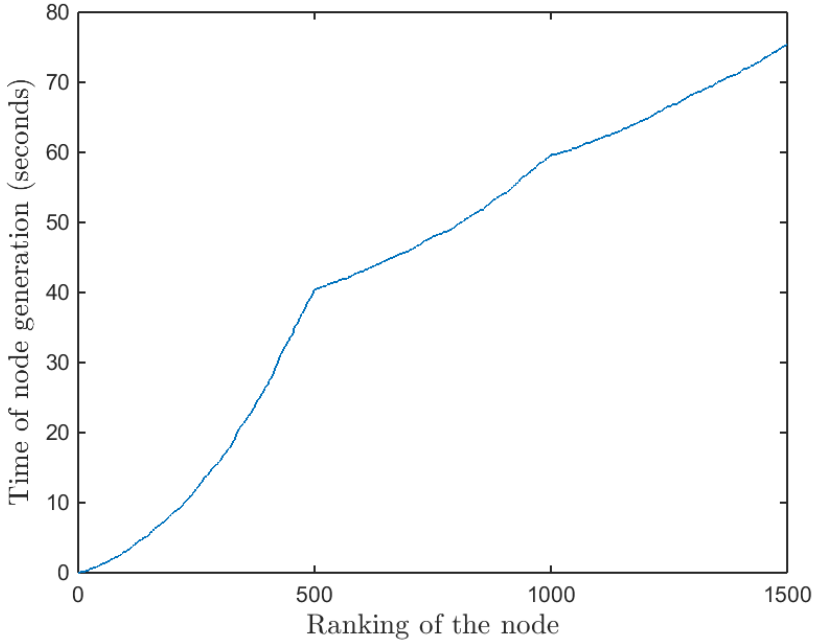



Figure 5.8: The times when individual nodes are created during one execution.

As it can be seen in the figure, the GERRT* was run with 1500 nodes in 3 time windows (TW), thus in each TW, there were 500 nodes. Exactly at the points that correspond to moving onto a next TW, there is a turn in the needed time. This is due to the fact that a node with an assigned TW $t$ is only able to optimize the costs of other nodes only if they are in the same or later TW $t' \geq t$. Thus, when the first nodes are added into a new TW, they are not able to optimize much, as opposed to the lastly generated nodes that can optimize all the previous ones. Note that the discontinuities here are caused by rewiring the edges in the Algorithm 4, not by finding the best parent node in the Algorithm 3, because setting parents is limited the other way around. That is, a node's parent can be only a node from the same or a previous TW.

However, there is another visible phenomenon in the Figure 5.8. The nodes generated in the first TW consume more than half of the computational time. The unbalanced distribution with regard to the reachable space causes it because the nodes in the TW 0 are distributed in the radius $\theta$, the maximal distance that can be travelled in one time window, around the root of the tree. That gives an area with size proportional to $\theta^2$ and node density $\frac{N}{3 \cdot \theta^2}$. In the next TW, the nodes can be distributed in the radius $2 \cdot \theta$ with the density $\frac{N}{12 \cdot \theta^2}$. Thus, using the knowledge from the previous section, the amount of nodes that fit into the radius $d$ will be approximately 4 times smaller in the second TW than in the first one. This does not happen precisely because as explained with the Voronoi regions in the section 2.1.1, the new nodes tend to be on the border of the already explored area and will therefore have higher density. Another factor is that the other optimization function behaves differently and there are also other processes in the algorithm that influence the speed. The reduction of nodes in the first TW is the reason why BGERRT* performed faster than GERRT*.

### 5.2.4 Summary

In this section, we found out that the time complexity of the algorithm is directly proportional to the squared amount of nodes in the tree $N^2$, the squared optimization radius $d^2$ and the cell count $M$ in the grid, formally

$$\text{runtime} \propto N^2 \cdot d^2 \cdot M \tag{5.2}$$

and

$$\text{runtime} \propto \frac{N^2 \cdot d^2 \cdot L}{s^2}. \tag{5.3}$$

These findings restrict the algorithm to be run with large input values. However, using too large values of these parameters is not recommendable even from the practical point of view because the algorithm will produce decent solutions even with lower settings and faster, as shown for example in the Figure 5.4.

## 5.3 Test scenarios

In this section, we present the results of the GERRT* algorithm on specific examples focused on different issues, such as avoiding detection and finding the shortest possible path.

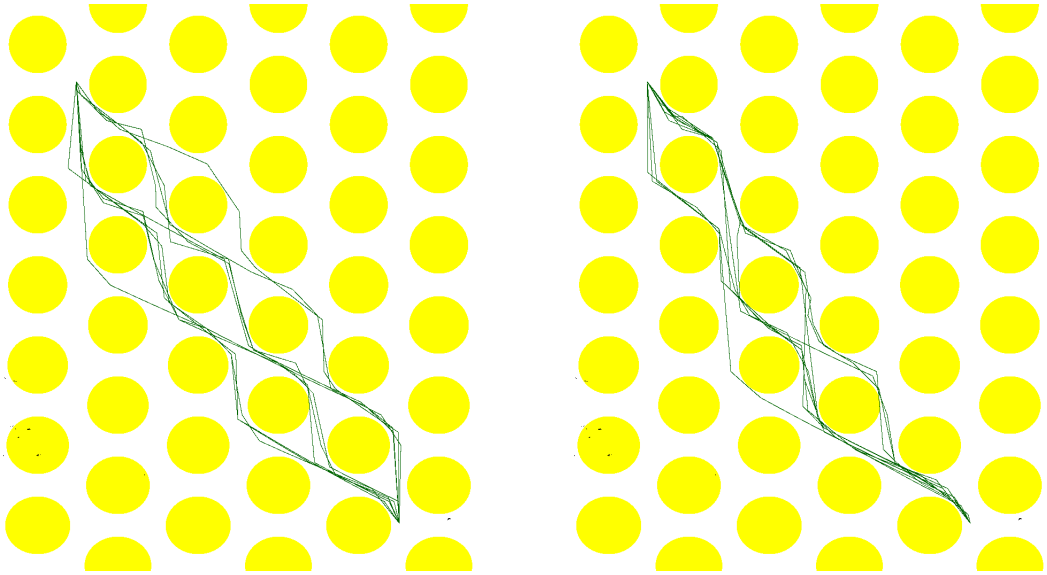
### 5.3.1 Array of detectors

In this scenario, we have created a hexagonal array of uniform detectors that the vessel needs to sail through. The exact settings of this scenario are listed in the appendix B.5. To experiment with the influence of the weight of risk detection $w_2$ in the cost function, we tried four different values of it and for each one, the algorithm was run 10 times with the results shown in the Figure 5.9.

First, the weight $w_2$ was set to 5000, a high value. In this case, the planned paths successfully avoid all the detectors and go around them via various detours, as it can be seen in the Figure 5.9a.

When the weight is decreased to 100 or 10, the paths already partially cross the areas with non-zero detection risk, especially in the case where $w_2 = 10$, shorter path is sometimes favoured over lower risk. The results with these settings are depicted in the Figures 5.9b and 5.9c.

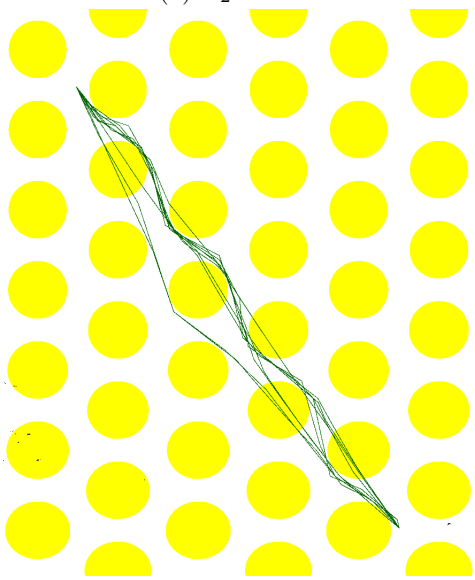
Eventually, if the risk of detection is not considered in the cost function at all with given $w_2 = 0$, the algorithm planned the route directly from the starting point to the end in all 10 cases, as shown in the Figure 5.9d.



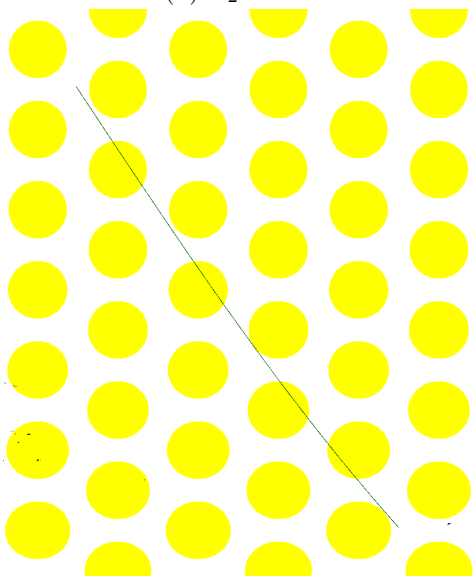
(a) $w_2 = 5000$



(b) $w_2 = 100$

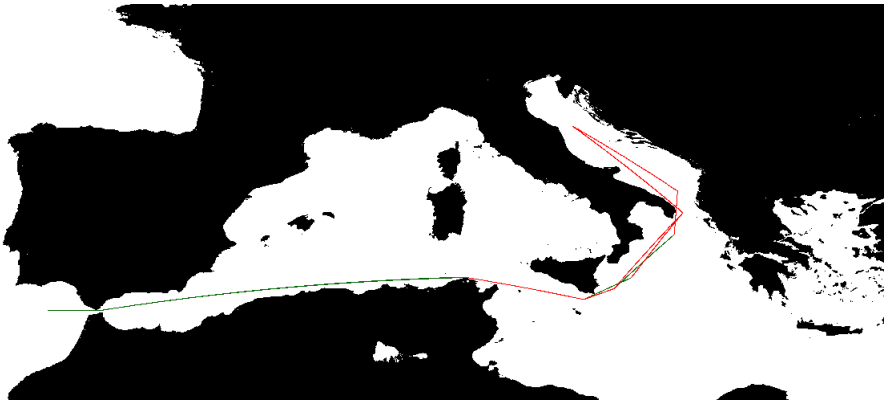


(c) $w_2 = 10$

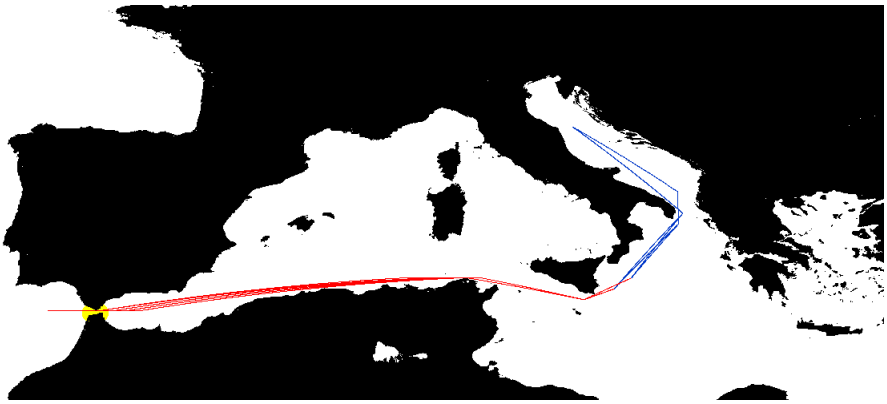


(d) $w_2 = 0$

Figure 5.9: The best paths in the environment for various $w_2$ weights.



(a) Without the detector.



(b) With the detector.

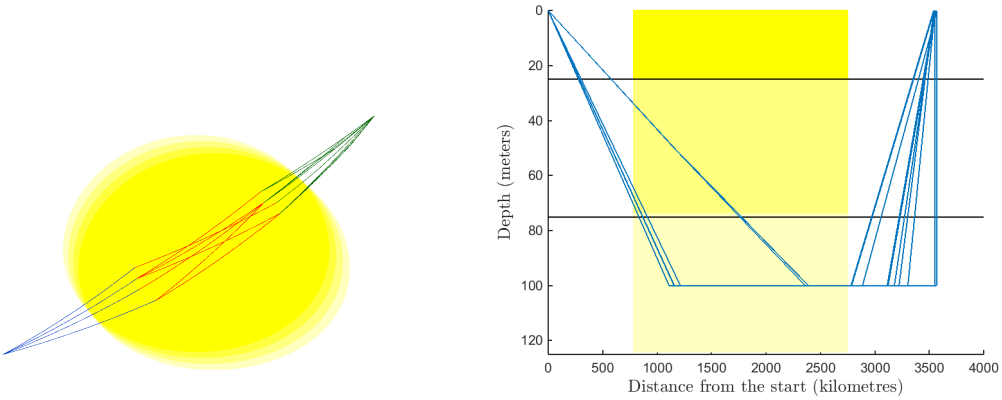
Figure 5.10: The best paths across the Mediterranean Sea.

### 5.3.2 Obstacle avoidance

Until now, the algorithm was only run on the high seas with negligible obstacles in the environment. That is why we came up with a scenario where obstacle avoidance could be tested. The area of the planning is the Mediterranean Sea, the starting point is some 200 kilometres westwards from the Strait of Gibraltar and the target is located in the northern part of the Adriatic Sea. The detailed settings are again listed in the appendix B.4.

In the basic version of this scenario, there are no detectors and the algorithm was able to reach the target in the time window 1 in all 10 planned paths. The differences between the paths are minor, as portrayed in the Figure 5.10a.

Additionally, we challenged the algorithm with a more complicated scenario, where a detector is placed at the Strait of Gibraltar and the preferences in the cost function are set to primarily avoid detection. The detector is active only in the time windows 0 and 2. In this case, the vessel always waits or moves just a little in the time window 0, sneaks through the strait in the TW 1 and finishes the voyage in the TW 2 without any risk of detection. If the time is disregarded, the planned paths depicted in 5.10b are similar to the ones from the previous case.




(a) Aerial view of the paths and the detector.

(b) Cross-section of the planned paths with highlighted detection areas with intensities and borders between layers.

Figure 5.11: The paths through the detector with exponentially decreasing risk.

### 5.3.3 Sailing under a detector

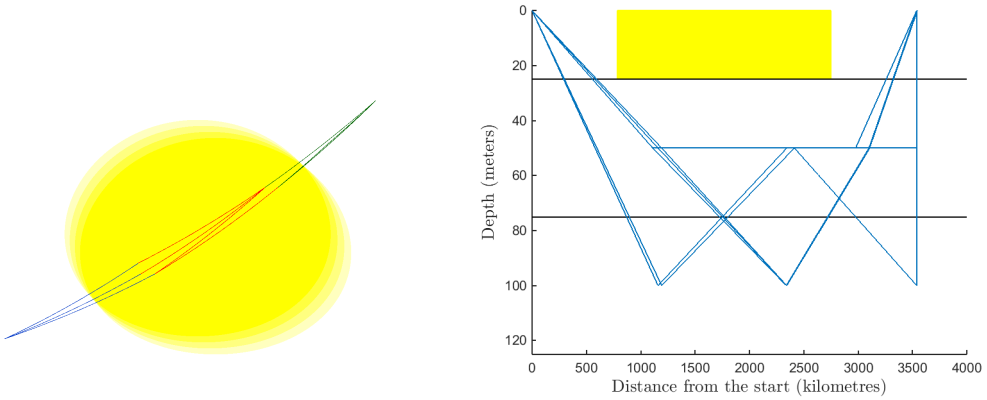
In this scenario, a submarine is trying to pass to the other side of a detector that almost completely fills the area between the start and target locations, therefore detouring would be too costly. However, the detector is fully effective only on surface and its efficacy is halved every 50 metres below surface. The rational thing to do is therefore sail as deep as possible because it is not penalized and lowers the detection risk.

The results of 10 runs in accordance with the settings B.6 are shown in the Figure 5.11a. In these settings, the grid has 3 layers with 50 meter interval and maximal depth of 100 metres. However, the aerial view gives us no information about the depths. Therefore, the paths were cross-sected in the Figure 5.11b, where the depth of the submarine is plotted against the progress on the path. There are also depicted the depth boundaries between the 3 layers of cells and the areas with non-zero detection risk are highlighted with yellow. As already noted previously, the risk equals 50 % of the surface value in the second layer and only 25% of the surface value in the third layer. Except two trajectories, majority of the planned paths are mostly in the deepest layer in the detection zone and surface directly after leaving it to reach the target zone.

We also experimented with a modification of these settings, where the detector was set to be active in the surface layer only. The results are presented in the Figure 5.12, where all the trajectories were able to avoid any detection and successfully sail under the detector.

### 5.3.4 Discussion

In the previous tests, the algorithm presented feasible results as expected, even though the found paths could undergo some smoothing procedure that would limit their variance and make them more direct, for instance the paths in the Figure 5.11a could be made shorter.




(a) Aerial view of the paths and the detector.

(b) Cross-section of the planned paths with the highlighted detection area and borders between layers.

Figure 5.12: The paths under the detector that is active on surface only.

Another aspect that was not mentioned before is that when more areas have non-zero detection risk and the weights in the cost function are set to highly avoid them, the found paths will feature larger variance. This is caused by the stochastic nature of the planning method. The algorithm samples the same amount of nodes, but many of these are placed into the regions with detection risk and the high $w_2$ weight prevents them from connecting to other nodes and being used further due to their large cost value. That leaves us with fewer nodes that can be actually used in the best path.

This can be seen in the Figure 5.10a, where the algorithm was able to find the shortest path along Africa in all of the 10 runs, whereas when the detector is added, the resulting paths vary slightly, as depicted in the Figure 5.10b. The same situation happened in the last scenario. If the detector could not have been avoided, the paths differed more significantly than when the detection could have been avoided completely, compare the variances of paths in the Figures 5.11a and 5.12a.

# Conclusion

In this thesis, we created a model of submarine that plans its path with the aim to minimise risk of detection, travelled distance and required time. The planner is based on random sampling, therefore the model can be used in Monte Carlo simulations.

To achieve that, we presented a gridding method that was successfully used to model real dynamic environment. The method proved to be fairly usable and scalable throughout the performed experiments.

Then, we defined a path planning optimization problem on the mentioned grid. During the formalization of the cost function, a question concerning the nature of detection risk arose. It was found out both theoretically and in simulated tests that the detection risk should not be represented as the probability of detection, but that a linear approach should be used. The linearity provided decent results and additionally offered possible generalizations and scalability.

To solve the optimization problem, we presented a modification of the RRT* algorithm, GERRT*. The modified method proved to be capable of operating in limitedly changing environment and planning trajectories in accordance with user-defined preferences. We concluded that the paths provided by the algorithm could undergo some post-processing that could rapidly improve the found solutions in few steps.

An Agent Behaviour Model of submarine for the BANDIT simulation framework was created and is able to successfully function within the simulation environment.

To add some ideas for possible further improvements, the time complexity of the GERRT* algorithm restricts it from being used in scenarios with many subsequent environment changes, even though it could be possible if a faster way of determining the cost function was used. Other possible improvements of the algorithm include experimenting with linear distribution of nodes which could shorten the execution duration or tailoring the random node sampling function to the environment, for example goal-driven or detection-avoiding sampling.

# Bibliography


M. Aid, W. Burr, and T. Blanton. Project Azorian. *The National Security Archive*, 2010. URL http://nsarchive.gwu.edu/nukevault/ebb305/.

Alabordache. Le sous-marin Téméraire, 2005. URL alabordache.fr.

J. F. Burns. French and British Submarines Collide. 2009. URL http://www.nytimes.com/2009/02/17/world/europe/17submarine.html?_r=0.

B. Clark. *The Emerging Era in Undersea Warfare*. 2015. URL http://csbaonline.org/publications/2015/01/undersea-warfare/.

Defence Supplier Directory. Defence Projects - Sonar 2087. URL http://www.armedforces.co.uk/projects/raq3f8d4e1b8587c#.VucfsvnhDIU.

Department of the Navy. SURTASS - LFA. URL http://www.surtass-lfa-eis.com/.

ESR. OSCAR third degree resolution ocean surface currents, 2009. URL ftp://podaac-ftp.jpl.nasa.gov/allData/oscar/preview/L4/oscar_third_deg/.

S. J. Freedberg. Transparent Sea: The Unstealthy Future Of Submarines. *Breaking Defense*, 2015.

F.-S. Gady. World's Largest Anti-Submarine Robot Ship Ready for Sea-Trials in April. *The Diplomat*, 2016. URL http://thediplomat.com/2016/02/worlds-largest-anti-submarine-robot-ship-ready-for-sea-trials-in-april/.

O. Grodzevich and O. Romanko. Normalization and other topics in multi-objective optimization. 2006.

P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/tssc.1968.300136. URL http://dx.doi.org/10.1109/TSSC.1968.300136.

O. Hrstka, Š. Kopřiva, J. Zelinka, and O. Vaněk. BANDIT Phase 1 report. Technical report, 2015.

J. Jenkins. Ocean Optics: Fundamentals & Naval Applications Technical Training Short Course Samples, 2012. URL http://www.slideshare.net/aticourses/ocean-optics-fundamentals-naval-applications-technical-training-short-course-sampler.

S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. *arXiv preprint arXiv:1005.0416*, 2010. URL http://roboticsproceedings.org/rss06/p34.pdf.

S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the RRT*. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1478–1483. IEEE, 2011.

F. Klügl. *Agent-based Simulation Engineering*. PhD thesis, University of Würzburg, 2009.

G. A. Korn and T. M. Korn. *Mathematical Handbook for Scientists and Engineers*. Mcgraw-Hill (Tx), 1967. ISBN 0070353700.

S. M. LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. 10 1998. URL `http://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf`.

S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and Prospects. 2001. URL `http://msl.cs.illinois.edu/~lavalle/papers/LavKuf01.pdf`.

B. Li, R. Chiong, and L.-g. Gong. Search-evasion path planning for submarines using the artificial bee colony algorithm. pages 528–535, 2014.

E. H. Lundquist. When Triton Circumnavigated the Globe. *Defense Media Network*, 2013. URL `http://www.defensemedianetwork.com/stories/when-triton-circumnavigated-the-globe/`.

S. Magnuson. DARPA's 130-Foot Crewless Ship to Set Sail in Spring. *National Defense Magazine*, 2016. URL `http://www.nationaldefensemagazine.org/blog/Lists/Posts/Post.aspx?ID=2082`.

P. Marks. Quantum positioning system steps in when GPS fails. *New Scientist*, 2014. URL `https://www.newscientist.com/article/mg22229694-000-quantum-positioning-system-steps-in-when-gps-fails/`.

National Geophysical Data Center. 2-minute Gridded Global Relief Data (ETOPO2) v2, 2006. URL `https://www.ngdc.noaa.gov/mgg/global/relief/ETOPO2/ETOPO2v2-2006/ETOPO2v2c/`.

M. Saha. *Motion Planning with Probabilistic Roadmaps*. PhD thesis, Stanford University, 2006.

J. Vincent. The US Navy's new autonomous warship is called the Sea Hunter. *The Verge*, 2016.

G. Wren and D. May. Detection of Submerged Vessels Using Remote Sensing Techniques. *Australian Defence Force Journal*, 1997. URL `https://fas.org/nuke/guide/usa/slbm/detection.pdf`.

# Appendices

# Appendix A

# CD content

Attached CD contains source codes of the algorithms, runnable GUI, and the thesis in the PDF format. The structure of the CD is described in the following table.

| Folder or file | Description |
|---|---|
| `dependency` | folder with compiled external libraries |
| `output` | folder where the outputs of the program are saved |
| `SVP` | folder containing the source codes of the project |
| `BP-1.0-SNAPSHOT.jar` | compiled source codes of the project |
| `startGUI.bat` | scripts for MS Windows and GNU/Linux that start the |
| `startGUI.sh` | GUI using the compiled sources |
| `thesis.pdf` | text of the thesis |

Table A.1: Directory structure of the CD.

The default planning area in the GUI is Central America, that corresponds to the latitudes between 35° N and 15° S and longitudes between 120° W and 60° W.

The application can not be run directly from the CD, because the planned paths in the `output` folder can not be overwritten on the disc. The solution therefore is to copy the files and run the application from a different location. Note that the resources in the `SVP/src/main/resources` folder are required to run the application. The folder includes the depth data, maps and the configuration file for BANDIT simulation.

# Appendix B

# Used settings

Since listing the extensive settings of the planning algorithm and grid directly next to each result might interrupt the readers' attention, the settings are presented here and are briefly linked in the text.

## B.1 Example output

| Variable | Value |
|---|---|
| Northwestern point of the grid | 69.5° N, 100.8° E |
| Southeastern point of the grid | 20° S, 65.7° W |
| Size of cell $s$ | 100 kilometres |
| Depth sampling interval $\Delta d$ | 50 metres |
| Maximal depth $m$ | 50 metres |
| Steer distance $\epsilon$ | 300 kilometres |
| Duration of one time window $D$ | 24 hours |
| End time window $E$ | 2 |
| Generated amount of nodes $N$ | 400 |
| Optimization radius $d$ | 1500 kilometres |
| Maximal travelled distance in one time window | 1200 kilometres |
| Maximal time submerged $t_{\text{submerged}}$ | 3 |
| Distance weight $w_1$ | 10 |
| RD weight $w_2$ | 1000 |
| Time weight $w_3$ | 10000 |
| Generated trees | 3 |
| Paths taken from each tree | 2 |
| Starting point $p_{\text{init}}$ | 2° N, 83° W |
| Target point $p_{\text{target}}$ | 11.4° N, 96° W |

Detectors:

- Uniform detector with the centre at 2° N, 92° W, radius 800 kilometres, risk value 0.5, active in time frames from 0 to 2. No depth influence.

## B.2 Experiment 1

| Variable | Value |
|---|---|
| Northwestern point of the grid | 69.5° N, 100.8° E |
| Southeastern point of the grid | 38.3° S, 65.7° W |
| Size of cell $s$ | 200 kilometres |
| Depth sampling interval $\Delta d$ | 50 metres |
| Maximal depth $m$ | 20 metres |
| Steer distance $\epsilon$ | 600 kilometres |
| Duration of one time window $D$ | 48 hours |
| End time window $E$ | 4 |
| Generated amount of nodes $N$ | 1500 |
| Optimization radius $d$ | 1500 kilometres |
| Maximal travelled distance in one time window | 2200 kilometres |
| Maximal time submerged $t_{\text{submerged}}$ | 1 |
| Distance weight $w_1$ | 1 |
| RD weight $w_2$ | 50000 |
| Time weight $w_3$ | 100 |
| Generated trees | 8 |
| Paths taken from each tree | 1 |
| Starting point $p_{\text{init}}$ | 40.4° N, 158.5° W |
| Target point $p_{\text{target}}$ | 35.7° S, 136.3° W |

Detectors:

- Uniform detector with the centre at 20.5° N, 158.9° W, radius 1600 kilometres, risk value 0.3, active in time frame 2. No depth influence.
- Uniform detector with the centre at 29.2° N, 137.4° W, radius 1600 kilometres, risk value 0.3, active in time frame 2. No depth influence.
- Gaussian detector with the centre at 19° S, 137° W, standard deviation 800 kilometres, risk value at the centre 0.4, active in time frame 2. No depth influence.
- Gaussian detector with the centre at 5° S, 137° W, standard deviation 800 kilometres, risk value at the centre 0.4, active in time frame 2. No depth influence.

## B.3 Experiments 2 and 3

The settings are the same as in the Settings 2, except the detectors are active from the time frame 2 to 4 and the maximal travelled distance in one time window was increased to 3000 kilometres. The amount of nodes $N$ changes, as defined in the Tables 5.3 and 5.4, if $N$ is not specified, it equals 3000.

## B.4 Obstacle avoidance

| Variable | Value |
|---|---|
| Northwestern point of the grid | 47° N, 9° W |
| Southeastern point of the grid | 30° N, 34° E |
| Size of cell $s$ | 50 kilometres |
| Depth sampling interval $\Delta d$ | 50 metres |
| Maximal depth $m$ | 20 metres |
| Steer distance $\epsilon$ | 600 kilometres |
| Duration of one time window $D$ | 48 hours |
| End time window $E$ | 2 |
| Generated amount of nodes $N$ | 3000 |
| Optimization radius $d$ | 1500 kilometres |
| Maximal travelled distance in one time window | 2500 kilometres |
| Maximal time submerged $t_{\text{submerged}}$ | 1 |
| Distance weight $w_1$ | 1 |
| RD weight $w_2$ | 5000 |
| Time weight $w_3$ | 1000 |
| Generated trees | 10 |
| Paths taken from each tree | 1 |
| Starting point $p_{\text{init}}$ | 35.8° N, 7.8° W |
| Target point $p_{\text{target}}$ | 43.6° S, 14.22° E |

Optional detectors (as explained in the second test scenario):

- Uniform detector with the centre at 35.9° N, 5.68° W, radius 50 kilometres, risk value 1, active in time frame 0. No depth influence.
- Uniform detector with the centre at 35.9° N, 5.68° W, radius 50 kilometres, risk value 1, active in time frame 2. No depth influence.

## B.5 Array of detectors

| Variable | Value |
|---|---|
| Northwestern point of the grid | 0° N, 140° W |
| Southeastern point of the grid | 30° S, 110° W |
| Size of cell $s$ | 30 kilometres |
| Depth sampling interval $\Delta d$ | 50 metres |
| Maximal depth $m$ | 20 metres |
| Steer distance $\epsilon$ | 600 kilometres |
| Duration of one time window $D$ | 168 hours |
| End time window $E$ | 0 |
| Generated amount of nodes $N$ | 1000 |
| Optimization radius $d$ | 1500 kilometres |
| Maximal travelled distance in one time window | 5000 kilometres |
| Maximal time submerged $t_{\text{submerged}}$ | 1 |
| Distance weight $w_1$ | 1 |
| RD weight $w_2$ | 5000, 100, 10, 0 |
| Time weight $w_3$ | 0 |
| Generated trees | 10 |
| Paths taken from each tree | 1 |
| Starting point $p_{\text{init}}$ | 2.5° N, 132.5° W |
| Target point $p_{\text{target}}$ | 27.5° S, 112.5° W |

The used detectors were uniform with radius 200 kilometres, risk value 1, active in time frame 0 without depth influence. Their centres are defined as follows.

- For all the latitudes 30° S, 25° S, 20° S, 15° S, 10° S, 5° S, 0° N, 5° N, 10° N, 15° N, 20° N, 25° N, 30° N the latitudes are 150° W, 140° W, 130° W, 120° W, 110° W, 100° W, 90° W.
- For all the latitudes 27.5° S, 22.5° S, 17.5° S, 12.5° S, 7.5° S, 2.5° S, 2.5° N, 7.5° N, 12.5° N, 17.5° N, 22.5° N, 27.5° N the latitudes are 145° W, 135° W, 125° W, 115° W, 105° W, 95° W, 85° W.

## B.6 Sailing under a detector

| Variable | Value |
|---|---|
| Northwestern point of the grid | 20° S, 50° E |
| Southeastern point of the grid | 50° S, 90° E |
| Size of cell $s$ | 100 kilometres |
| Depth sampling interval $\Delta d$ | 50 metres |
| Maximal depth $m$ | 100 metres |
| Steer distance $\epsilon$ | 600 kilometres |
| Duration of one time window $D$ | 24 hours |
| End time window $E$ | 3 |
| Generated amount of nodes $N$ | 2000 |
| Optimization radius $d$ | 1500 kilometres |
| Maximal travelled distance in one time window | 1230 kilometres |
| Maximal time submerged $t_{\text{submerged}}$ | 4 |
| Distance weight $w_1$ | 1 |
| RD weight $w_2$ | 50 |
| Time weight $w_3$ | 100 |
| Generated trees | 10 |
| Paths taken from each tree | 1 |
| Starting point $p_{\text{init}}$ | 25° S, 85° E |
| Target point $p_{\text{target}}$ | 45° S, 55° E |

Detectors for $i = 0, 1, 2, 3$:

- Uniform detector with the centre at (37 + $i\cdot$ 0.5)° S, (72 - $i\cdot$ 0.5)° E, radius 1000 kilometres, risk value 1, active in time frame $i$. Exponential decrease of risk value with depth, decrease by 0.5 in 50 metres.