Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Ola Nystein

Real-Time Strategy game inspired Multi-Agent path planning

Master's thesis in Cybernetics and Robotics
Supervisor: Kostas Alexis
Co-supervisor: Mihir Dharmadhikari and Mihir Kulkarni
February 2023

NTNU
Norwegian University of
Science and Technology

Ola Nystein

Real-Time Strategy game inspired Multi-Agent path planning

Master's thesis in Cybernetics and Robotics
Supervisor: Kostas Alexis
Co-supervisor: Mihir Dharmadhikari and Mihir Kulkarni
February 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Preface

This thesis concludes my work during the autumn of 2022 and early spring of 2023 to finish a Master's degree at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. This master's thesis has been completed under the supervision of Dr. Kostas Alexis.

The thesis is a continuation of the specialization project completed during the spring of 2022 in the TTK4550 course, which revolved around bridging the gap between the fields of multi-agent robotic control and large-scale unit control in *real-time strategy*(RTS) games. This thesis has significantly benefited from the work done by the Autonomous Robots Lab(ARL) at NTNU and Autonomous Systems Lab(ASL) at ETH Zürich concerning the DARPA subterranean challenge[1][2].

This thesis' main contribution is a proof-of-concept multi-agent path planner for medium-sized teams of unmanned aerial vehicles, designed to fit in a proposed RTS-like system architecture for large-scale control of robotic teams. The planner is implemented to run on the *Robot Operating System*(ROS)[3] framework. ARL and ASL have contributed to the development by supplying ROS packages that handle mapping, simulation, and control for use in the thesis. The planner, and its related modules, are written in C++ and use the ROS package manager Catkin to build and compile the project.

I want to thank my supervisor Kostas Alexis for excellent supervision throughout the semester, providing valuable insight when things looked tough. A special thanks go to Mihir Dharmadhikari and Mihir Kulkarni at ARL for engaging discussions and thorough debugging sessions. A big thanks to my friends and family for supporting me through my final semester at NTNU and to my significant other for keeping me healthy and fed.

08.02.2023
Ola Netland Nystein

Abstract

The use of robotic control and autonomous robots is at the forefront of today's industry and scientific research. With the help of advanced technology, society can now handle complicated and hazardous situations using remotely operated units overseen by humans. Despite great strides in controlling individual robots and even small groups of robots, there is still a need for methods to manage large teams of robots effectively. *Real-time strategy*(RTS) games have long been able to control hundreds of units effectively, but more research needs to be conducted between the fields of robotics and gaming to bridge the gap. This master's thesis aims to further bridge this gap by examining existing methods in both fields to find the approaches and principles used in gaming that are translatable for controlling large-scale robotic teams.

This thesis will present an overview of existing methods for multi-agent control, as well as design principles based on principles from gaming and the field of *Human-Computer Interaction*(HCI) to elevate user experience. A responsive asynchronous multi-agent path planner with robust static collision avoidance is developed and implemented, along with visual aids and text-based feedback tools, to enhance performance by giving the operator an intuitive user experience. The planner is based on an existing implementation of a rapidly-exploring random graph algorithm in combination with a probabilistic roadmap to provide the ability to handle multiple queries on the same graph structure. The algorithms are designed to operate in high-dimensional non-convex environments and the proposed planner will be demonstrated in a range of environments to showcase how they affect performance. The planner is designed to fit in a more extensive architecture inspired by existing state-of-the-art AI systems developed for playing RTS games. The work done in this thesis will hopefully inspire further research and development to bridge the gap between gaming and robotics to achieve effective systems for large-scale control of robotic teams.

Sammendrag

Bruken av autonome roboter og robotstyring står i fokus for nåtidens industrielle og vitenskapelige forskning. Ved bruk avansert teknologi kan samfunnet nå håndtere kompliserte og farlige situasjoner ved å bruke fjernstyrte enheter overvåket av mennesker. Til tross for store fremskritt innen kontroll av individuelle roboter eller små grupper av roboter, er det fortsatt behov for metoder for effektiv håndtering av store robotsvermer. Real-time strategy(RTS)-spill har lenge kunnet kontrollere hundrevis av enheter effektivt, men det er et hull i forskningen mellom robotikk- og spillindustrien. Denne masteroppgaven ønsker å bygge videre på broen mellom disse feltene ved å undersøke eksisterende metoder i begge felt for å finne tilnærminger og prinsipper brukt i spillutvikling som er overførbare til kontroll av store robotsvermer.

I denne oppgaven vil det bli presentert en oversikt over eksisterende metoder for multi-agentkontroll, samt designprinsipper for å forbedre brukeropplevelsen basert på prinsipper fra videospillutvikling og menneske maskin-interaksjon (MMI). En responsiv asynkron multi-agent-stiplanlegger med robust statisk kollisjonsunngåelse er utviklet og implementert, sammen med visuelle hjelpemidler og tekstbaserte tilbakemeldingsverktøy for å forbedre ytelsen ved å gi operatøren en intuitiv brukeropplevelse. Planleggeren er basert på en eksisterende implementering av en raskt-utforskende tilfeldig grafalgoritme i kombinasjon med et probabilistisk veikart for å muliggjøre håndtering av flere forespørsler på samme grafstruktur. Algoritmene er designet for å operere i høy-dimensjonale ikke-konvekse miljøer, og den foreslåtte planleggeren vil bli demonstrert i en rekke miljøer for å vise hvordan operasjonsmiljøet påvirker planleggerens ytelse. Planleggeren er designet for å passe inn i en større arkitektur inspirert av eksisterende toppmoderne AI systemer utviklet for å spille RTS-spill. Arbeidet utført i denne oppgaven vil forhåpentligvis inspirere videre forskning og utvikling for å bygge broen mellom spillindustrien og robotikk for å oppnå effektive systemer for storskala kontroll av robotsvermer.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Research on the autonomous control of robots plays a large part in today's industrial and scientific community. The advances made have made it possible for industrial companies and other institutions to employ state-of-the-art techniques and methods to utilize robots for challenging tasks such as fault detection, manual operations, and surveillance[4][5][6]. The majority of methods applied in today's society revolve around autonomous control of a single unit or small-scale multi-agent systems often surveyed by a human operator[7][8]. While some algorithms and software exist, not much work has been done to develop techniques for large-scale multi-agent control in applied robotics. A reason for this lack of research could be that it is currently not practically viable nor a pressing issue to be able to control hundreds of units collaboratively. This thesis aims to explore the topic of multi-agent control to see which existing methods and algorithms could be used to make a practically viable multi-agent control system.

A motivation for this project was to explore if some techniques or principles used in *real-time strategy* (RTS) games could be applied to controlling large groups of robots in a realistic simulated environment. RTS games have ways of managing large numbers of robots at once, each with a specific role and goal, but not all of which are essential for the overall mission to be successful[9]. This type of redundancy in units and abilities could be valuable in real-world situations, such as disaster response or urban search-and-rescue missions, where mission completion is crucial. These use cases present a situation where the robots operate in unknown environments and must explore and search without initially observing the complete operation space. Multi-agent teams can consist of units with different capabilities and could therefore be used for a broader range of tasks than a single robot. Additionally, multiple tasks can be assigned to a team, with the units or a unit manager dividing the tasks among them.

In the preliminary project of this thesis completed in the course TTK4550, it was found that real-life robotic control and RTS games have some fundamental differences in the lowest unit control and management level. When the environment is initially unknown, a robot must possess an advanced motion planning system and a mapping and localization method to function autonomously[1]. These requirements are not present in the case of RTS games as the environment is fully observable, so accordingly, the techniques used for motion planning, mapping, and localization must stem from the world of robotics. RTS games can still offer inspiration for solving higher-level unit management, such as order distribution and choosing the subsequent actions, and providing a good user experience for the operator. Unit control in RTS games and robotics are highly similar on a superficial level, especially in visualizing the problem for a player or operator. Therefore, this thesis will explore existing techniques and methods for problem visualization in RTS games to emulate this in the proposed multi-agent path planning system.

This project will explore existing multi-agent robotic control methods and investigate how these can be used in an RTS-like architecture for large-scale robotic teams. The specific use case that will be investigated is the exploration of unknown environments and how to achieve intuitive robotic fleet control by employing principles from RTS games.

1.2 Master thesis goals

This thesis aims to further the research done in the author's project thesis on RTS game-like control on large robotic teams. The main focus of this report is to present a multi-agent path planner based on random-sampling algorithms that provide an intuitive way for operators to control robotic fleets. The planner design will draw inspiration from principles from large-scale unit control in RTS games, as well as existing methods in multi-agent robotic control. The planner should be tested in diverse operating spaces to ensure environment agnosticism and showcase the configuration difference needed to operate in distinct environments. The proposed planner will be designed to fit into an RTS-like system architecture.

Some desired properties set for the resulting method are that it should be intuitive to control for human operators and possess the possibility to scale the controlled fleets. These criteria stem from the world of RTS games, where human players control hundreds of units simultaneously. The planning method developed should provide quick, anytime solutions as RTS games and real environments have strict real-time constraints in their planners as changes in the environment can frequently happen during the planning phase. The robustness of the planner should not be compromised by increasing planner speed to such an extent that the paths returned result in collisions. The planner should take heed of static and known dynamic obstacles in the form of other units controlled by the same planner. The simulated environments used in this thesis will be completely static, with the exception of the controlled units. Therefore, local reactive collision avoidance of unknown dynamic obstacles and path recovery is assumed to limit the thesis scope.

A stable network connection between the units and the planning system is assumed throughout this thesis to allow for uninterrupted communication. This is a simplification of the planning problem, as network disruptions may occur in real-life operations. This assumption lowers the needed autonomy in each unit, only requiring them to have a robust motion controller to command the robots to a given waypoint. This assumption is reasonable as this project primarily focuses on software and will obtain all simulation results. The multi-agent path planner is meant to fit into a larger RTS-like architecture where stable communication is required for operation and information passing.

This thesis aims to incorporate principles from the field of *Human-Computer Interaction*(HCI) in planner development to deliver an intuitive RTS-like experience. Providing an intuitive experience for operators implies good visualization of the environment and robot positions and simple control of the robot's goals. This goal will be achieved by developing a global minimap that gives the operator a simplistic view of the environments and the agents, as well as utilizing existing visualization tools to give more detailed information about the agents' current mission.

The scope of this thesis is to create a robust and quick multi-agent motion planner to work in conjunction with the existing VoxBlox[10] mapping framework. The planner will be designed to fit inside a more extensive RTS-like architecture and will be developed to possess the ability to operate with limited knowledge about the environment a priori. The secondary goal is for the planner to deliver a satisfactory user experience based on principles from HCI and gaming. Related work in the fields of robotics and RTS games will be examined to determine the methods suitable for use in the control of large-scale robotic teams.

1.3 Contributions

This thesis continues the work of bridging the gap between RTS game methods and robotic control by presenting a proof-of-concept method for the control of large-scale robotic teams with sizes of up to ten units, which still delivers fast planning. The idea is that the developed planner combined with a reasonable and robust cohesive leader-follower procedure, such as clearance corridoring[11], could provide great scalability to control teams for up to hundreds of units. This approach divides large fleets of robots into multi-agent units to plan for several robots as if they were one. At the same time, the leader-follower procedure ensures cohesive movement inside each unit. This thesis' main contributions are

- A functioning asynchronous multi-agent path planner for control of teams for medium-sized teams demonstrating robust and quick planning in both obstacle-heavy and open environments. The planner possesses static collision avoidance and conflict detection between paths to avoid collision between units. The planner is based on a rapidly-exploring random graph algorithm and contains elements from a probabilistic roadmap method to enable multiple queries on the same graph and provide effective search in non-convex high-dimensional spaces.
- Tools for enhancing user experience based on principles in HCI and RTS games. The tools include a global minimap and configurations of existing visualization packages in the *Robot Operating System*(ROS) framework[3].

1.4 Related work

A lot of progress has been made in the field of autonomous control of robots over the last decades, both in academics and industry. In contemporary industry, robots are often tasked with completing *dirty, dangerous, or dull* tasks, but the scopes may increase as technology advances[12]. The demand for large-scale multi-agent control of robots as a fleet may arise to handle the increased scopes of tasks robots could carry out on their own and to increase their task completion effectiveness. To determine which methods from the world of RTS may be viable for use in real-life robotics, they need to be investigated and compared to their current counterparts in robotics.

1.4.1 Multi-agent path planning

The path planning problem posed by RTS games is, as mentioned, substantially simpler when compared to real-life robotics. Common path planners used in gaming are BVP[13] or Monte Carlo algorithms[14], to name a few, which all are based on planning in fully observable environments that are pre-discretized to cells for path generation and collision checking. As this thesis proposes a multi-agent path planner meant to function in environments where the operating space is unknown a priori, the space needs to be discretized online by a sophisticated mapping module that bridges the gap between continuous actuation control and planning[15]. Due to these key differences, the multi-agent path planner will primarily be based on existing work and methods from real-life robotics.

As stated by Kolushev and Bogdanov in [16], it is logical to divide the problem of path planning for multi-agent robotic systems into two sub-problems. The first is path planning and optimization for each robot with static collision avoidance and consideration of other units' movement. This subproblem could be handled by a decentralized planning algorithm such as the one proposed in [17]. The algorithm is based on a random sampling algorithm called *Closed Loop Rapidly-exploring Random Tree**(CL-RRT*) explained in[18]. A decentralized approach means that each unit has a high degree of autonomy as it plans its own local path for execution without necessarily communicating with the other units in the system. This approach is typical for large-scale robotic teams as the scalability for the local planners is linear with the number of agents. This approach poses a challenge of reaching global feasibility when units have intersecting areas of operations or need to cooperate. Consensus-based approaches require agents to exchange information on each agent's state to reach a global consensus on path choice. This exchange reduces the scalability as each agent needs to review a proposed path for approval, which increases complexity and latency in the planning, but has shown potential when operating in sparse environments where planning is infrequently required. Another approach is to assign a priority to each agent and order the paths by sequential planning following the priorities[19][20]. The sequential planning approach used in distributed computer systems may suffer from busy units blocking idle units from task execution which limits the performance of the system[21]. This approach also assumes uninterrupted communication between units which may not be the case in real-life environments. The D* Lite Multi-Agent path planner proposed in [22] is a decentralized planner where robot paths are modeled as temporal obstacles where paths are optimized for time. The star(*) class of algorithms[23] are search algorithms that require a pre-existing graph or grid representation of the environment.

The counterpart of decentralized planners is to have a centralized algorithm where a type of supervision algorithm plans keeps track of all agent states and subsequently plans paths for them. This planning manager gathers information about the environment collected from the sensors onboard all units, as well as the unit's current motion. The *Coordinated Multi-Robot Exploration Aquila Optimaizer*(CME-AO) proposed in [24] is a centralized multi-agent path planner based on the Aquila optimization algorithm and combines collision-free path planning with a swarming method. This method is designed for operating in unexplored environments and plans paths by discretizing the environment in cells. A variant of a planning manager may also be used in a decentralized algorithm, where it is only used as a synchronization platform for environment and agent information. In contrast, the planning is done individually by the robots. Utilizing a planning manager for path-finding tends to increase the complexity of the planning problem due to

the increased dimensionality of solving paths for several agents simultaneously. This approach uses synchronized planning where all paths are returned simultaneously. Alternatively, a queue-based system may be used to decrease the complexity by dividing the multi-agent path planning problem into sequential single-agent problems. These approaches are prone to the same performance issues due to blocking as decentralized sequential algorithms due to the fixed queue order.

A requirement for robotic teams operating in unknown or partially explored environments is a functional online map-building module that discretizes the environment as it is explored for collision checking during planning. This functionality is crucial for enabling the usage of robots to explore areas otherwise inaccessible or too dangerous for humans. The *Defense Advanced Research Projects Agency*(DARPA) Subterranean Challenge is a prime example of an initiative to accelerate scientific progress in this area, as the challenge was aimed at encouraging universities and companies to create autonomous systems capable of navigating unmapped subterranean environments[2]. The winning entry was called *GBplanner*. It utilized graph-based path planners, including a local and global planner, which were based on the *Rapidly-exploring Random Graph*(RRG)-algorithm[1]. The mapping module used in GBplanner, which is based on Signed Distance Fields, will be used in this thesis and further explained in section 3.3.1. The competition also featured other path planning algorithms, such as a variant of the A* algorithm, known as *hybrid-state A**[25].

1.4.2 Collision avoidance

The second sub-problem for multi-agent path planning mentioned by Kolushev and Bogdanov in [16] is local reactive collision avoidance of unforeseen obstacles along with path recovery. This feature is required for systems where the units operate in dynamic and unpredictable environments with potentially moving obstacles unknown to the planner a priori. Local collision avoidance in robotics can be divided into two categories. Avoiding dynamic uncontrollable obstacles and avoiding teammates controlled by the same planner moving along the same path are two separate problems requiring their own solutions. Global collision avoidance of static objects and known teammates moving to other target regions could be handled by a global path planner. A local collision avoidance module can also handle avoiding teammates that are not part of the same swarm.

Controlling a large team of robots as a cohesive swarm while not colliding with other swarms is called *segregative behavior*. Popular methods for achieving swarm control are often based on Reynolds *Boid*-model[26]. The Boid model aims to imitate natural flocks of birds when controlling the swarm to achieve cohesive movement toward the target area while avoiding collisions between units in the swarm. Boid is based on the social potential field method, which can suffer from oscillations and local optima when operating in unexplored environments[27]. Collision avoidance in potential field methods assumes that the approaching obstacle does not have its own collision avoidance method, which can create problems when encountering non-cooperative units employed by a different planner. Velocity obstacle-based collision avoidance, such as the *Reciprocal Velocity Obstacle*(RVO) method[28], is an approach that assumes that the encountered unit employs the same decision process leading to a wait-and-see-like behavior that provides slow, but stable obstacle passing. Santos et al. propose a hierarchical collision avoidance module that combines velocity obstacles and flocking by abstracting encountered swarms as a single unit to achieve segregated behaviour[27]. This method is unfortunately also prone to the same problems as other velocity obstacle-based methods, such as RVO, as it often acts too conservatively by prioritizing lower speeds to achieve collision avoidance. Cohesive motion between units can also lead to a possibility of abstracting the controlled unit as a single unit to drastically reduce global planning requirements, such as the method described in [11].

Local reactive collision avoidance is not required in RTS games, as the entire state space and its obstacles are always known a priori when planning for collision-free paths. However, some RTS games employ versions of flocking algorithms based on the Boid model to achieve cohesive movement that feels realistic[29]. Global collision avoidance in RTS games is often solved by combining simple search algorithms, such as A*[30], with potential field algorithms. An example of this is used in the well-known StarCrafts-series[31]. As the static basic A* algorithm is unable to deliver satisfactory multi-agent path planning due to the strict real-time constraints of RTS

games, potential field algorithms are used to repel units away from obstacles as they move along the path generated by A*. Potential field methods are prone to reaching local optima, but a way to handle this is to introduce repelling fields along the executed path as the unit passes[31].

1.4.3 User Experience

One of this thesis' sub-goals is that the proposed planner should deliver some degree of RTS-like user experience. Achieving this requires an overview of existing principles for human interfacing used in RTS games to tie the relevant parts into a real-life robotic system. Wenjun and Xiudong present a hierarchical semantic differential model to categorize elements of HCI and *Enjoyment Quality*(EQ) for different groups of systems in [32]. An industrial system for controlling robotic teams would often fall into the *key system* where the chief design goal is utility to ensure quick and errorless operation if mission completion is paramount. Designs with utility in focus are often minimalistic to ensure non-cluttered information for operators to interpret but often require long training times. The user interfaces of RTS games deal with how to efficiently organize relevant information and intuitive control mechanics for the units. As stated in [33], a clear presentation of player goals with concise feedback to provide a helpful learning experience is critical to achieving effective game design. An important design decision when designing games in general, including RTS games, is to categorize which information is necessary for a player to complete a mission[34]. This recommendation translates easily to the world of robotics, where mission failure often has worse consequences than in games, and filtering out information that may only confuse an operator is essential for achieving the best possible design.

HCI is also an important topic of discussion and research in the robotics industry, where a subset of HCI called *Human-Robot interaction*(HRI) often is the main topic. Yanco and Drury [35] offers an overview of the taxonomies of HRI with a perspective from the field of HCI. The taxonomy can work as a guideline for system design, as it presents the needs different systems may require. An assembly robot frequently needing hardware maintenance demands vastly different design decisions than a distributed multi-robot system used for exploration or surveillance. The latter needs an interface similar to a game-oriented user interface from the HCI domain, as the direct human-to-robot contact is on a superficial level, such as action control or monitoring. Other system features that should impact system design on a user experience level are the autonomy level of the units, the people-to-robot ratio, and team size[35]. An example of an implemented and tested system in the HRI field is the ROS-based TurtleUI[36], which is a generic *Graphical User Interface*(GUI) for robot control. TurtleUI bases its implementation on well-known principles from HCI, such as minimizing screen pollution, while also aiming its design toward software developers who can use the GUI for easier error identification and localization.

1.5 Thesis structure

This thesis is organized as follows. The first chapter contained some background and motivation for the thesis, a literature review of the current standings in multi-agent control and collision avoidance in both robotics and RTS games, and a presentation of some principles and approaches from HCI and HRI. Section 2 contains relevant theory and concepts used in the rest of the thesis. Software frameworks and tools are presented in section 3, along with a detailed explanation of the developed planner. The implemented tools to enhance user experience are described in section 3. Section 4 presents results from simulating and tuning the multi-agent path planner, along with some insights on how the tuning affects user experience. Section 5 concludes the report and provides some thought on further work that could be done concerning the themes explored in this thesis. The code written for this thesis is enclosed with the report in the delivery of this thesis.

2 Theory

In this section, the general problem formulation for path planning is recounted before the relevant theory used to develop the multi-agent path planner is presented. Additionally, the techniques behind the mapping framework are described. As this master's thesis is based on the work done by the author in the specialization project course TTK4550, large parts of the theoretical material are the same. Parts of this chapter are hence the same as in the preliminary project thesis.

2.1 Path and Motion Planning

Although methods used for path and motion planning in RTS games could be viable for use in robotics, this project utilizes a method from the world of robotics as these have been tried, tested, and proven robust for real-life use[37] [38]. As described in [38], the robotic motion planning problem can be stated as finding a sequence of control inputs to make the robot move from its initial state to one of the goal states while not colliding with the environment or other obstacles. As described in section 3.2, the interface between planning and controller allows for simply providing a waypoint in the form of a pose, described in eq. (2), to make the robot move to said waypoint. The algorithm described in this part is thus a path planner whose task is to decide the series of waypoints to describe a path from the start position to the goal region. Ideally, the algorithm used to solve the path-planning problem should be *complete* and *optimal*.

2.1.1 Problem formulation

The path planning problem is formalized in [38] and will be recounted in this section. Let $\chi = (0,1)^d$ be the full configuration space, i.e. full operating environment, where $d \in \mathbb{N}, d \geq 2$. χ_{obs} is the obstacle region such that the closure of the set $\chi \setminus \chi_{obs}$ is the closed set of obstacle-free space χ_{free}. x_{init} is the initial condition and is an element of χ_{free}. χ_{goal} is the goal region and is the open subset of χ_{free}. The triplet $(\chi_{free}, x_{init}, \chi_{goal})$ defines the path planning problem. The function σ is defined as $\sigma : [0,1] \to \mathbb{R}^d$, the total variation of σ $\mathrm{TV}(\sigma)$ can be defined as

$$\mathrm{TV}(\sigma) = \sum_{\substack{i=1 \\ n \in \mathbb{N}, 0=\tau_0<\tau_1<\cdots<\tau_n=1}}^{n} |\sigma(\tau_i) - \sigma(\tau_{i-1})| \tag{1}$$

If $\mathrm{TV}(\sigma) < \infty$, then the σ can be said to have bounded variation and can be called a path if it is continuous. A path is collision-free if $\sigma(\tau) \in \chi_{free}$ for all $\tau \in [0,1]$. A collision-free path is feasible if $\sigma(0) = x_{init}$ and $\sigma(1) \in \mathrm{cl}(\chi_{goal})$. The task of the path-planning algorithm is to return the path $\sigma(\tau)$ if the path is feasible, and failure if not.

2.1.2 Algorithms

The planning algorithm developed in this thesis is based on two existing well-known planning algorithms combined into one algorithm to keep some desired properties from both. This chapter will include theoretical background on why these algorithms were chosen and some comparisons with other approaches. The algorithms are a *Probabilistic Roadmap*(PRM)-algorithm and the *Rapidly-exploring Random Graph*(RRG)-algorithm. RRG differs from PRM in that it incrementally builds a single connected roadmap, while PRM may produce a forest-structured roadmap with disconnected segments. The single-unit planner proposed in the author's preliminary project report from TTK4550 used the related algorithm RRT* proposed first in [39] as its main planning algorithm. RRT*, although faster and more memory-efficient than RRG[38], produces a tree structure for motion planning instead of a graph, which is not convenient for multi-agent planning as a tree only has one root, corresponding to a single robot starting position, for finding the shortest path.

2.1.3 Multi-Agent path planning

The classical path planning problem presented in section 2.1.1 can be extended to a multi-agent path planning problem by having a triplet $(\chi_{free,i}, x_{init,i}, \chi_{goal,i})$ for each unit i. The initial conditions x_{init} and goal regions χ_{goal} must be different for each unit, while the obstacle-free space χ_{free} may be the same in a coupled planning algorithm, such as the one developed in this thesis. A *coupled planning algorithm* is an algorithm that aggregates all individual units into one system and solves the planning problem by applying single-unit planning algorithms[20].

A popular measure of solution quality is the sum of time each agent spends outside their goal areas. This measure is used in the MA-RRT*, which takes in an n-tuple of starting waypoints $(s_1, \ldots, s_n)$ and goal waypoints $(d_1, \ldots, d_n)$, where (s_i, d_i) denotes the starting waypoint and goal waypoint of unit i[40]. The algorithm returns a set of paths p_i for each robot i where $start(p_i) = s_i$ and $end(p_i) = d_i$. The paths returned must be separated to avoid collisions, that is, $\forall j, k, t : j \neq k \Rightarrow dist(p_i[t], p_j[t]) > d_{sep}$ where d_{sep} being some arbitrary separation distance usually in relation to robot size.

The methods for solving the classical multi-agent planning problem, such as the MA-RRT*, return paths for all units simultaneously, which is different from the approach developed in this thesis. Path planning in user-controlled RTS games gets orders one at a time. Hence the approach used in this thesis is an asynchronous queue-based multi-agent planner. This choice means that the actual path planner need not solve a multi-agent planning problem but rather be able to solve a sequence of single-agent planning problems quickly.

2.1.4 Completeness

One of the desired properties of a planning algorithm is *completeness*. If, and only if, an algorithm returns a solution in finite time if it exists for a specified input, the algorithm is said to be complete. Otherwise, it should return failure [41]. Complete planning algorithms have high complexity resulting in high computational costs when dealing with complex environments, thus making them unsuitable for most practical uses. Therefore, the completeness requirement must be relaxed to find a suitable planning algorithm. As described in [38], relaxing the constraint only to require *resolution completeness* works for explicit representations of environments up to five dimensions. The planning problem in this thesis is far too complex for these types of representations. Hence the algorithms need to rely on implicit representations of the configuration space. Sampling-based algorithms omit the use of explicit environment representation by utilizing a collision-checking module to determine the feasibility of robot configurations in the configuration space[42]. PRM and G-RRT* are sampling-based algorithms that fulfill a relaxed completeness requirement called *probabilistic completeness*. Algorithms are said to be probabilistic complete if and only if the probability of failure if there exists a solution approaches zero as the number of sampled configurations approaches infinity. PRM and RRG-based algorithms have a probability of failure that decays exponentially with respect to the number of points sampled[43][44].

2.1.5 Optimality

An optimal motion planning algorithm returns the lowest cost path from a starting configuration to a given goal region with respect to some measure, such as distance traveled, time used, or energy consumed. Distance traveled will be used in this project as this is not dependent on differences in units' energy consumption or velocity constraints. Although producing optimal paths has been proven to be very computationally expensive and challenging even in basic cases[45], there are methods for approaching optimality. As mentioned in [46], RRG is *asymptotically optimal*, meaning that it will, in most cases, produce a reasonably good path without excessive computational cost[38]. A modification to increase the connection radius from $\gamma_{RRT^*}(\frac{\log n}{n})^{1/d}$ to $\gamma_{RRT^*}(\frac{\log n}{n})^{1/(d+1)}$ to properly consider the added time-dimension that decides the ordering of the randomly sampled points is suggested by [47]. The proof in [46] shows that the total distance cost of the path returned by RRT* converges to a robust optimum as the number of samples approaches

infinity.

2.1.6 Anytime algorithms

Another desired trait of a planning algorithm functioning in real-time systems is that it is an *anytime algorithm*. As stated in [37], an anytime motion planning algorithm should have some form of completeness guarantee as well as asymptotic optimality, both of which are qualities possessed by RRG[38]. Additionally, anytime algorithms should overlap execution and computation in time, meaning that they should possess the ability to improve the current plan while executing parts of it. The anytime trait is desirable because a robot usually spends more time executing a plan than finding the plan. This effect leaves time for online computation on the unit while it is moving, again decreasing the total time used to solve and execute the planning problem.

2.1.7 Multi-query

As discussed, the RRG algorithm has several desirable qualities for a planner, but it is a single-query planner. Single-query means that RRG will erase the graph between planning iterations and start entirely anew when a new goal destination is given to the planner. The graph-building phase is the most time-consuming in random sampling algorithms, and to allow for greater scalability, a *multi-query* algorithm is desired. Multi-query algorithms keep the graph intact between planner queries allowing for quick look-ups to check if a feasible path already exists and can be returned immediately. This feature increases the scalability of the queue-based planner because it frees up computation time for the algorithm when similar queries arrive rather than rebuilding graphs in the same space for different units. PRM algorithms can easily be multi-query and hence will be used in the design of the path planner developed in this project[48].

2.1.8 PRM

PRM[48] is a class of sampling-based algorithms that, unlike RRT-based algorithms, does not erase the built graph or tree between queries, thus possessing the multi-query trait. The basic PRM algorithm described in [48] is reproduced in algorithm 1, and the primitive procedures and variables it uses are listed below.

- R is the roadmap consisting of vertices $R.V$ and edges $R.E$. Vertices are configurations in the obstacle-free space that are part of the roadmap, while edges are free paths between two vertices noted by (v_1, v_2) where v_1 and v_2 are vertices.
- *SampleConf* returns a configuration q sampled uniformly from the configuration space at random.
- *FreePath*(q_1, q_2) is a function that returns true if and only configurations q_1 and q_2 can be connected with a straight line in the obstacle-free space.
- *FreeConf*(q) returns true if and only if the configuration q is in the obstacle-free space.
- *Vertices*(R) returns the number of vertices in the roadmap R.
- N is a tuning parameter deciding how many vertices the roadmap maximum should contain.

Algorithm 1 PRM(q_1, q_2, N)

```
1: if FreePath(q_1, q_2) then
2:     return Path between q_1 and q_2
3: end if
4: R.V ← q_1, q_2
5: repeat
6:     q_new ← SampleConf
7:     if FreeConf(q_new) then
8:         R.V ← R.V ∪ {q_new}
9:     end if
10:    for each Node v ∈ R.V such that v ≠ q_new do
11:        if FreePath(q_new, v) then
12:            R.E ← R.E ∪ (q_new, v)
13:        end if
14:    end for
15: until q_1 and q_2 are in the same connected component of R ∪ Vertices(R) > N + 1
16: if FreePath(q_1, q_2) then
17:    return Path between q_1 and q_2
18: else
19:    return Failure
20: end if
```

The basic PRM algorithm in algorithm 1 builds a roadmap of vertices and edges to find a way from q_1 to q_2 in the obstacle-free configuration space. In this version, the algorithm terminates when q_1 and q_2 are connected in the roadmap, but this is not required for using PRM. If multiple queries are expected, one could run the building phase for a set amount of time or until a set amount of configurations is connected to create a roadmap with good coverage over the configuration space. Good coverage means that the planner has a higher chance of returning queried paths between configurations in the free space without the need to sample more points. A requirement for achieving good coverage with this tactic is that the total state space is known, which is different in this thesis. Graph building is also the most time-consuming part of random-sampling algorithms, making this approach a poor choice for situations with real-time constraints. Another approach is a node enhancement procedure that steps in if the current roadmap is insufficient. The most straightforward node enhancement procedure is seen in the main loop of algorithm 1, where the algorithm keeps adding new samples to the roadmap to increase connectivity until the path is found. This enhancement procedure can be modified to more efficiently explore narrow passages, focus near the unit, or concentrate on sparse areas in the roadmap[49].

2.1.9 RRG

This project's main motion planning algorithm will be the RRG algorithm first proposed by Karaman and Frazzoli in [39]. The algorithm was a proposed improvement of the existing RRT algorithm, which is not asymptotically optimal. The algorithm's task is to incrementally build a traversable undirected graph from unit positions towards the goal region given. The basic RRG algorithm presented in [39] is shown in algorithm 2, with primitive procedures used by the algorithm listed below. RRG takes an initial unit position x_{init} and a maximum number of samples N as input.

- *Sample*$_i$ returns a sampled point $x_i \in \mathbb{R}$ from the obstacle-free space $\chi_{free} = \text{cl}(\chi \setminus \{\chi_{obs}\})$. Where cl$(\cdot)$ denotes the closure of a set, χ is the full configuration space, χ_{obs} is the obstacle region.
- Given a graph $G = (V, E)$, where the vertices $V \subset \chi$, and a random-sampled point $x \in \mathbb{R}$ from χ_{free}, *Nearest* returns the closest vertex in V in terms of euclidean distance.

- Given two 3D-points $x, y \in \chi$, *Steer* returns a new 3D-point $z \in \chi$ such that $\|z - y\|$ is minimized while $\|z - x\| \leq \mu$ is still true for a distance $\mu > 0$. In the context of RRG this function moves the sampled point closer to the closest existing vertex along the line of direction between the two points. This function only does something if the sampled point is outside the sphere created with the point returned by *Nearest* as centre, and μ or the connection radius proposed in [47] as radius.
- *ObstacleFree* returns *true* if the line segment between two given 3D points in χ lies completely in χ_{free} and false otherwise.
- *Near* returns the vertices in a given graph $G = (V, E)$ that lies inside a sphere defined by the given radius $r \in \mathbb{R} > 0$ and centre $x \in \mathbb{R}^3$. This function uses the connection radius proposed by [44]; $r = \gamma_{RRT^*}(\frac{\log n}{n})^{1/(d+1)}$.
- *Cost(x)* returns the accumulated cost of the distance to travel along the graph $G = (V, E)$ from the source vertex x_{init} to the input vertex x.
- *Card(V)* returns the number of vertices in the set V.

Algorithm 2 RRG

$V \leftarrow \{x_{init}\}; E \leftarrow \emptyset; i \leftarrow 0$
while $i < N$ **do**
  $G \leftarrow (V, E)$
  $x_{rand} \leftarrow \textsc{Sample}(i); i \leftarrow i + 1$
  $x_{nearest} \leftarrow \textsc{Nearest}(G = (V, E), x_{rand})$
  $x_{new} \leftarrow \textsc{Steer}(x_{nearest}, x_{rand})$
  if $\textsc{ObstacleFree}(x_{nearest}, x_{new})$ **then**
    $X_{near} \leftarrow \textsc{Near}(G = (V, E), x_{new}, \min\{\gamma_{RRT*}(\log(\text{card}(V))/\text{card}(V))^{1/(d+1)}, \mu\}))$
    $V \leftarrow V \cup \{x_{new}\}$
    $E \leftarrow E \cup \{(x_{nearest}, x_{new}), (x_{new}, x_{nearest})\}$
    for each $x_{near} \in X_{near}$ **do**
      if $\textsc{CollisionFree}(x_{near}, x_{new})$ **then**
        $E \leftarrow E \cup \{(x_{near}, x_{new}), (x_{new}, x_{near})\}$
      end if
    end for
  end if
end while
return $G = (V, E)$

The RRG expands a graph aimlessly in a configuration space, being able to deliver a path to a region if, and only if, a point in that region has been sampled. The graph grows denser with the number of samples, thus providing both asymptotic optimality and the probabilistic completeness of offering a solution given sufficient time to sample points. When RRG has built its graph, a target can be searched for by traditional shortest-path algorithms for graphs, such as Dijkstra's algorithm[50]. The space and time complexities of PRM and RRG are shown in table 1. Although PRM offers a better space complexity than RRG, this is at the cost of asymptotic optimality[38], hence why RRG is used as the core of the planner implemented in this thesis. An example of a graph created by RRG with a single robot operating in a narrow corridor is shown in fig. 1. Figure 1 is retrieved from the GBplanner paper[1].

Algorithm	Processing time complexity	Query time complexity	Memory complexity
RRG	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
PRM	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

Table 1: Complexity of the RRG and RRT* algorithms, where n is the number of vertices in the tree or graph

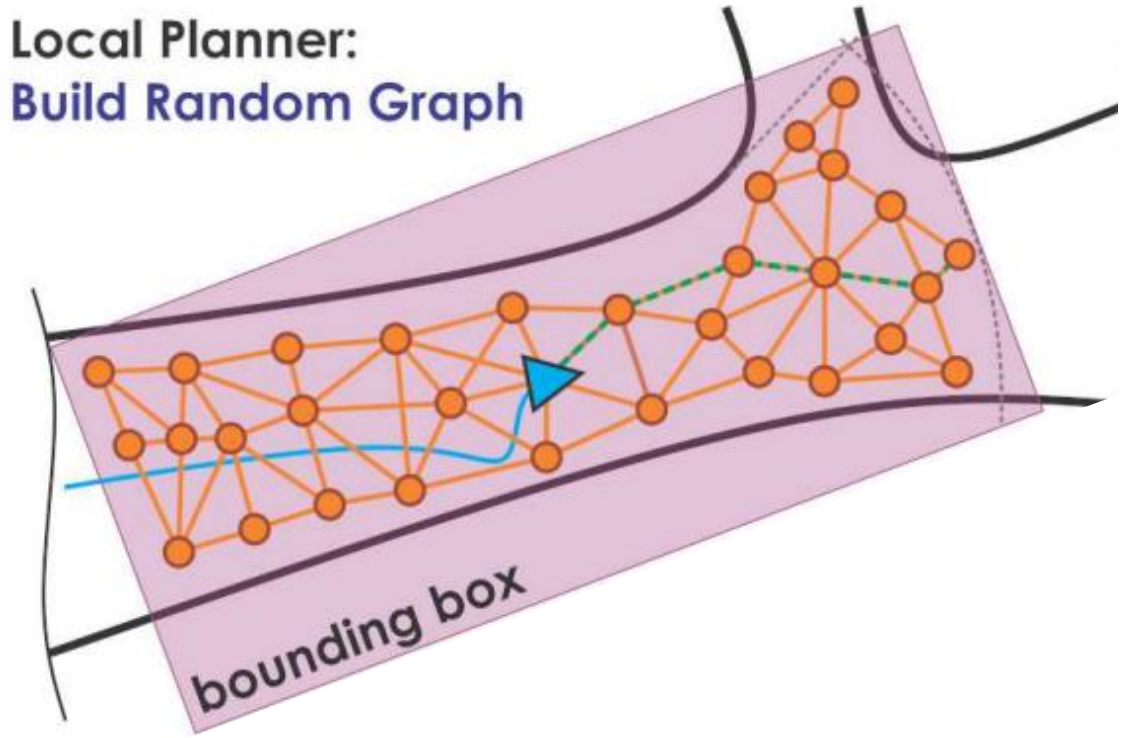


Figure 1: An example of a local graph built by an implemented version of RRG in a narrow corridor. The vertices are marked as orange circles, while an arbitrary path found in the graph is marked as a green dotted line. The blue triangle is a robot, and the purple outline is the planner's local bounding box which limits the reach of the graph built[1].

2.2 Mapping

As the mapping method used in this thesis is identical to the method used in the author's specialization project in TTK4550, the following section is retrieved from the project report. The environments used in this project are high-dimensional spaces with a large number of obstacles, and describing them explicitly results in a high computational cost to create a solution[38]. This is not viable considering that the solution paths have to arrive fast to ensure effective planning, especially when the goal is to achieve an effective planner for the control of large robotic teams. Therefore the mapping method used in this project describes the obstacles implicitly. Occupancy maps have been a popular method for describing obstacles for robotic motion planning, but Oleynikova *et. al.* proposed a method combining *Truncated Signed Distance Fields*(TSDF) and *Euclidean Signed Distance Fields*(ESDF)[42]. This method was later implemented by Oleynikova *et. al.* as an open-source C++ module called *VoxBlox* which is described later in section 3.3.1[10].

Signed Distance Functions(SDF) are methods of measuring how close a given point is to a border of a metric space[51]. By dividing the volume to be mapped into voxels, i.e., volumetric pixels, SDFs are a valuable tool for perceiving and mapping the environment with its obstacles and collision checking for path planning. ESDF and TSDF both use SDFs but in slightly different ways. In both, a voxel with a positive distance value signals that the voxel is in free space, while a negative

value signals that the voxel is occupied. Surface detection is done by detecting zero-crossings, while points in unknown space are not given a distance value. TSDF is a representation most commonly used for the perception side of mapping due to its high precision of surface detection and accurate modeling of the error of depth estimates. The distance stored in each voxel in TSDF is calculated by finding the distance between the center of a given voxel and the closest surface along the ray direction from the sensor's viewpoint. Figure 2 shows a TSDF where the black line symbolizes an object surface. The values are also truncated to only exist near surfaces which makes for a low space complexity[42]. In ESDFs, the distance is calculated as the Euclidean distance to the nearest occupied voxel. A visualization of an ESDF is shown in Figure 3 where the black line represents a surface. ESDF-based methods allow for fast collision checking, which makes them useful for path-planning and optimization[42]. Figure 2 and Figure 3 are retrieved from Oleynikova *et. al.*[52].

The method proposed in [42] combines ESDF and TSDF and results in a single map representation suitable for online construction and planning. New sensor scans are converted to a TSDF represented by calculated projective distances inside some truncated distance close to the surface. The ESDF distances are then found for all other voxels in the explored volume by iteratively computing the euclidean distances based on the projected distances inside the truncated zone. The method also allows for dynamically growing of the map by voxel-hashing, as it is not required to know the maximum map size beforehand[10]. Voxel-hashing is particularly useful when exploring fully unknown environments, as is assumed in this project.

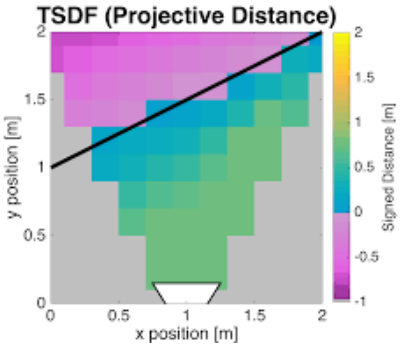


Figure 2: Visualization of a TSDF where each cell stores the projecting distance from the sensor to the surface[52]

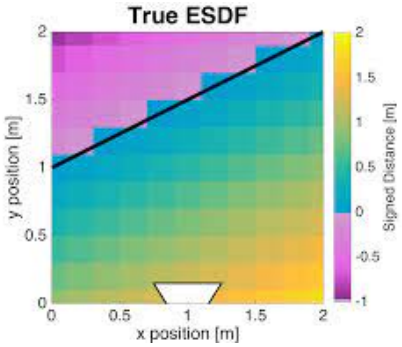


Figure 3: Visualization of an ESDF where each cell stores the true Euclidean distance to the surface[52]

3 Implementation and method

This chapter contains a description of existing packages and frameworks used in the implementation of the queue-based path planner. A thorough explanation of the planner's behavior and reasons behind design choices will conclude this chapter.

3.1 Simulation environment and software development

This project was created using the *Robot Operating System* (ROS), an open-source framework for robotics software development. It is written in C++, as most of the libraries and packages used in the planner are also in C++. The *Gazebo v.11.9.0* simulator was used to test the project and is compatible with the used ROS version (ROS Noetic). The ROS visualization tool, Rviz, was used to view the robot's perception of the environment, including frames, sensor data, and algorithms[53]. A visualization package developed in conjunction with GBplanner was also used to display the VoxBlox map and other planner elements in rviz. The environments used to test the planner are a narrow labyrinth-like environment, as well as an open canyon-like environment which is further described in section 4.

3.1.1 Robot Operating System

ROS is a framework that enables the development of robotic systems to centralize the majority of their functionality. This framework provides the services that one would expect from a full operating system, including but not limited to low-level control, real-time message passing between processes, and software package management[3]. In this report, certain terms will be introduced and their definitions provided. Further information on ROS can be found on the ROS documentation website[54].

- **Nodes** are executable programs that communicate with other nodes using ROS. In this project, the nodes include the controller and the planner.
- **Messages** data structures passed between nodes for communicating. Their structure is defined by text files with the suffix *.msg*.
- **Topics** are buses where messages are sent and can be read. They are defined by a name in string format. Nodes can publish or subscribe to a topic to respectively send on and read from the topics. An example of a topic used is the odometry topic used for publishing the estimated pose of the robot at a given interval.
- **Services** and **service clients** form the basis for ROSs' request/reply communication paradigm. Text files with the suffix *.srv* define the data structure of a request and the subsequent reply. Nodes can advertise a service and have a client that calls a service from another node. The main planner service is called with a target pose, and the planner then replies with the solution path if it was able to find one - failure otherwise.

3.2 Robot model and controller

The motion planner developed in this thesis is designed to work with any position controller that takes a 3D point and orientation in free space as input and moves the robot to that location. The specific robot used in testing is the *Cerberus M100: Medium UAV*, a medium-sized unmanned aerial vehicle that utilizes a sensor fusion that combines 3D medium-range LiDAR, visual camera, and inertial measurements system to perceive its environment and determine its position[1]. The M100 is treated as a black box in this project and is accessed through a planner-motion interface module[55]. The M100 model used in simulations was obtained from the *rotorS Gazebo* library[56]. For collision checking purposes, the robot is modeled as a 3D box, denoted the *robot bounding box*. The interface module was largely pre-existing and only required minor modifications to work with

the motion planner in this project. It receives a list of poses and passes them to the controller, which then determines the necessary actuator inputs to move the robot along the specified path. In this project, only the x and y coordinates of the target points are used, with the z coordinate always being set to 2 m due to the flat nature of the simulation environment. The orientation, represented by the quaternion subvector $q = [q_1, q_2, q_3, w]^T$ in eq. (2), describes the rotation from the robot/sensor frame to the world frame. The position controller used in this project is the *Lee Position Controller* from the rotorS simulation control library[56].

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ q_1 \\ q_2 \\ q_3 \\ w \end{bmatrix} \tag{2}$$

Equation (2) defines the structure of a pose vector in this project. The first three entries (x, y, z) describes a point $\in \mathbb{R}^3$, and the last 4 entries (q_1, q_2, q_3, w) describes an orientation on quaternion form[57].

3.3 Pre-requisites

This section will provide an overview of the main packages used to construct the autonomous motion planner. A list of additional libraries used in this project can be found in appendix C.

- The **Planner-Control-interface** (PCI) acts as a link between the position controller of the robot and the planner developed in this project. It is discussed in more detail in section 3.2[55].
- **RotorS Simulator** is the library used to simulate the robot and its sensors and control system in Gazebo[56].
- **KdTree** is a package that manages a k-dimensional tree[58] object in C++[59].

3.3.1 VoxBlox

VoxBlox[10] is an open-source implementation of the SDF-mapping method proposed in[42], which is designed to quickly build and update ESDFs from TSDFs for fast local mapping with efficient collision detection. It runs on the CPU of a Micro Aerial Vehicle (MAV), a type of small unmanned aerial vehicle. Building a map with VoxBlox can be divided into two steps: constructing and expanding a TSDF using incoming sensor data and updating the ESDF by propagating changes from the TSDF to the ESDF. The system uses voxel hashing to store voxels, which allows for faster insertion and lookup compared to the *Octomap* framework[60]. Voxel hashing improves the insertion and lookup complexity from Octomaps $\mathcal{O}(\log n)$ to Voxblox's $\mathcal{O}(1)$. The robot is modeled as a 3D bounding box for collision-checking purposes. The VoxBlox system and its maps can be accessed and visualized through the *MapManager* interface and Rviz. MapManager is a part of the *Planner Common*-package, described in the next section. An example of a Gazebo simulation compared to the same environment as perceived by VoxBlox and visualized with Rviz is shown in Figure 4.

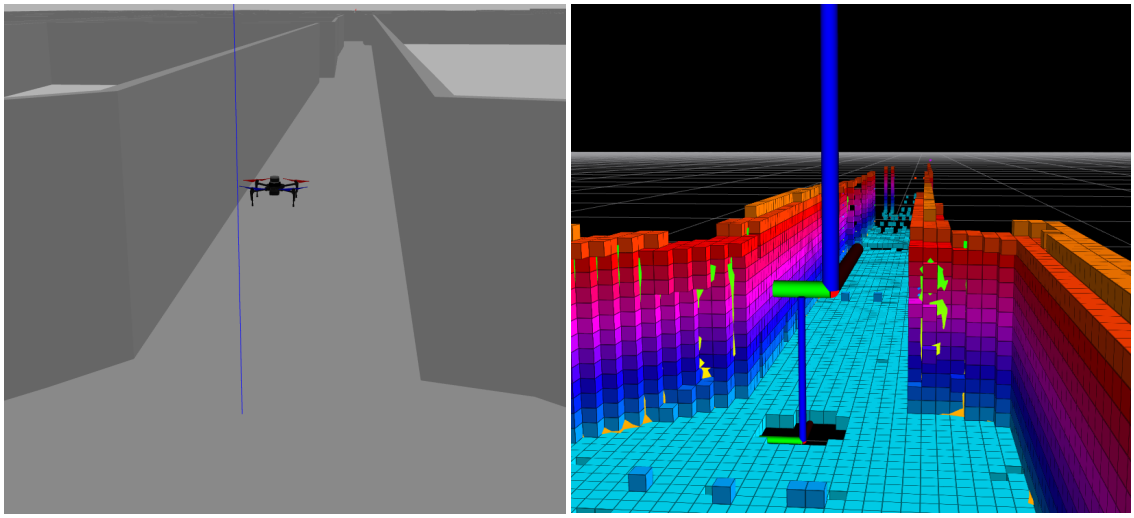


Figure 4: Comparison of the visualization of the Gazebo Simulation(a), and the percepted map created by VoxBlox (b). The MAV in (a) is indicated by the large red, green and blue axes in the centre of (b).

3.3.2 Planner Common

The *Planner Common* package is a group of packages designed to assist with planning tasks. It was originally developed by Dang *et al.* for use in the GBplanner project[1], and was shared privately for this project. Minor modifications were made to some of the packages for them to be compatible with the planner developed in this thesis, but the functionality remains the same. Planner Common is located in the core planner in the architecture shown in fig. 7. The most important functionality provided by Planner Common is

- Graph Management. An interface between the planner and the k-d-tree package[59] used to maintain the tree.
- Map Management. An interface between the planner and VoxBlox used to retrieve information from the map created by Voxblox, including collision-checking.
- Parameter parsing. Loading sensor, robot and environment parameters from configuration files.
- Random sampler. A C++ class that enables random sampling from Uniform, Normal and Cauchy-distributions.

3.4 Multi-Agent Path Planner

The preliminary work to this thesis included a proposed design for a complete architecture for an RTS-inspired system for large-scale multi-agent control. The architecture in fig. 5 reflects the recurring theme of hierarchical AI systems in RTS games[61][62]. The hierarchy follows a bottom-up principle for information flow, while decisions and commands move downwards. The planning managers, who are employed by function managers, assign targets to the local path planners. Each group of units has its own designated planning manager. The function managers oversee specific task types. The highest level of control lies with the operation commander, who creates long-term plans and ensures their successful completion. The operation commander can be a human operator or a neural network-based AI. The main objective of the planner developed in this thesis

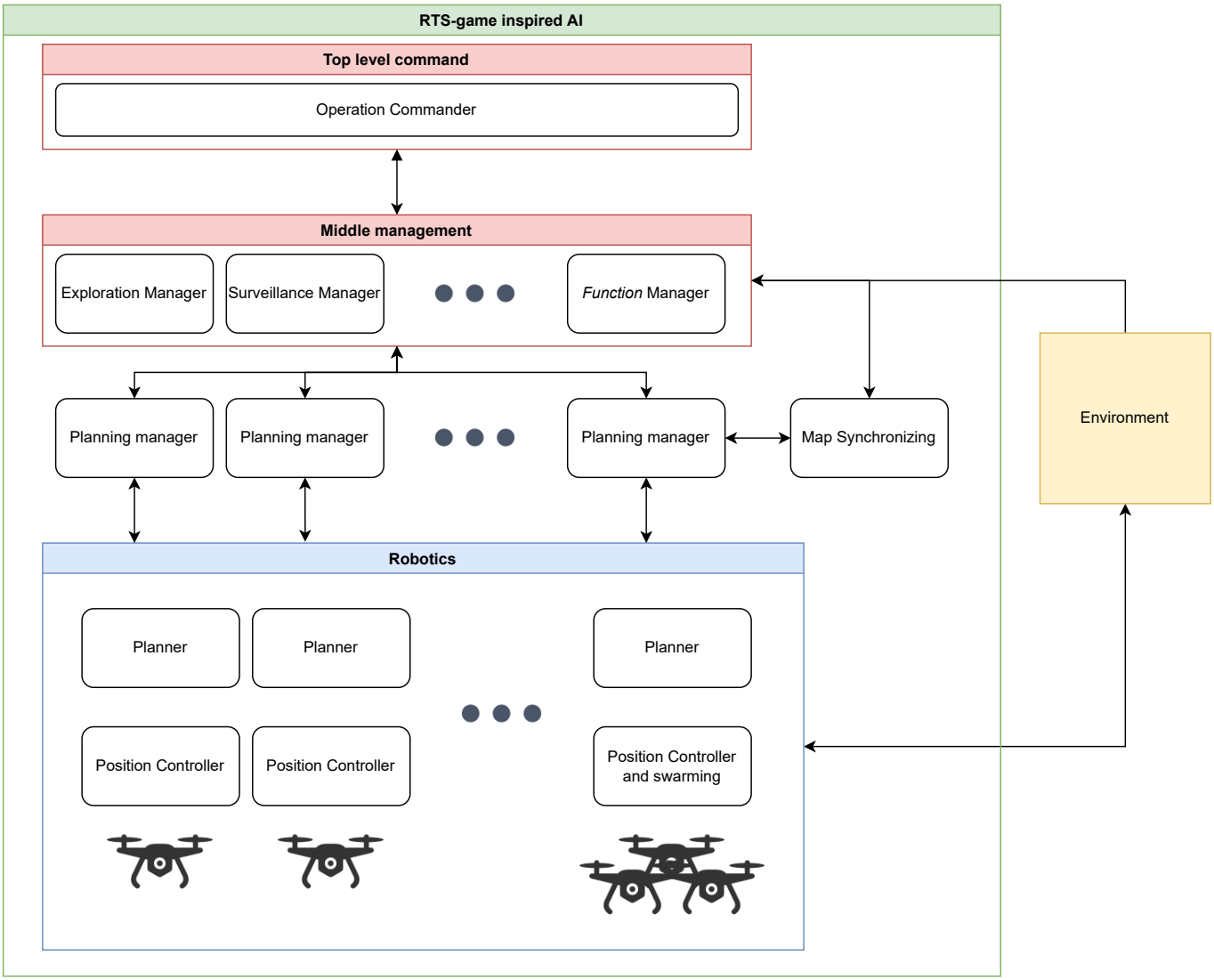


Figure 5: Proposed architecture of an RTS-inspired AI-system for control of large robotic teams. Retrieved from the author's project report in TTK4550 as preliminary research to this thesis.

is to produce collision-free paths for a multi-agent team while maintaining an RTS-like feel when controlling the units. This implies that the functionality should cover the **Planning manager** as well as the **Planner** boxes in fig. 5.

The planning algorithm is designed to fulfill the desired traits described in section 2.1. The implementation provides scalable, fast, anytime path planning for multi-agent path planning problems. Real-time constraints were prioritized in the development to give an RTS-like control experience. A simple dynamic obstacle avoidance method is implemented between the units, and the planner itself accounts for static obstacle avoidance. The resulting planner algorithm is used in a system architecture that provides an RTS-like multi-agent control based on a FIFO(first in, first out)-queue system. The overall system architecture is shown in fig. 6.

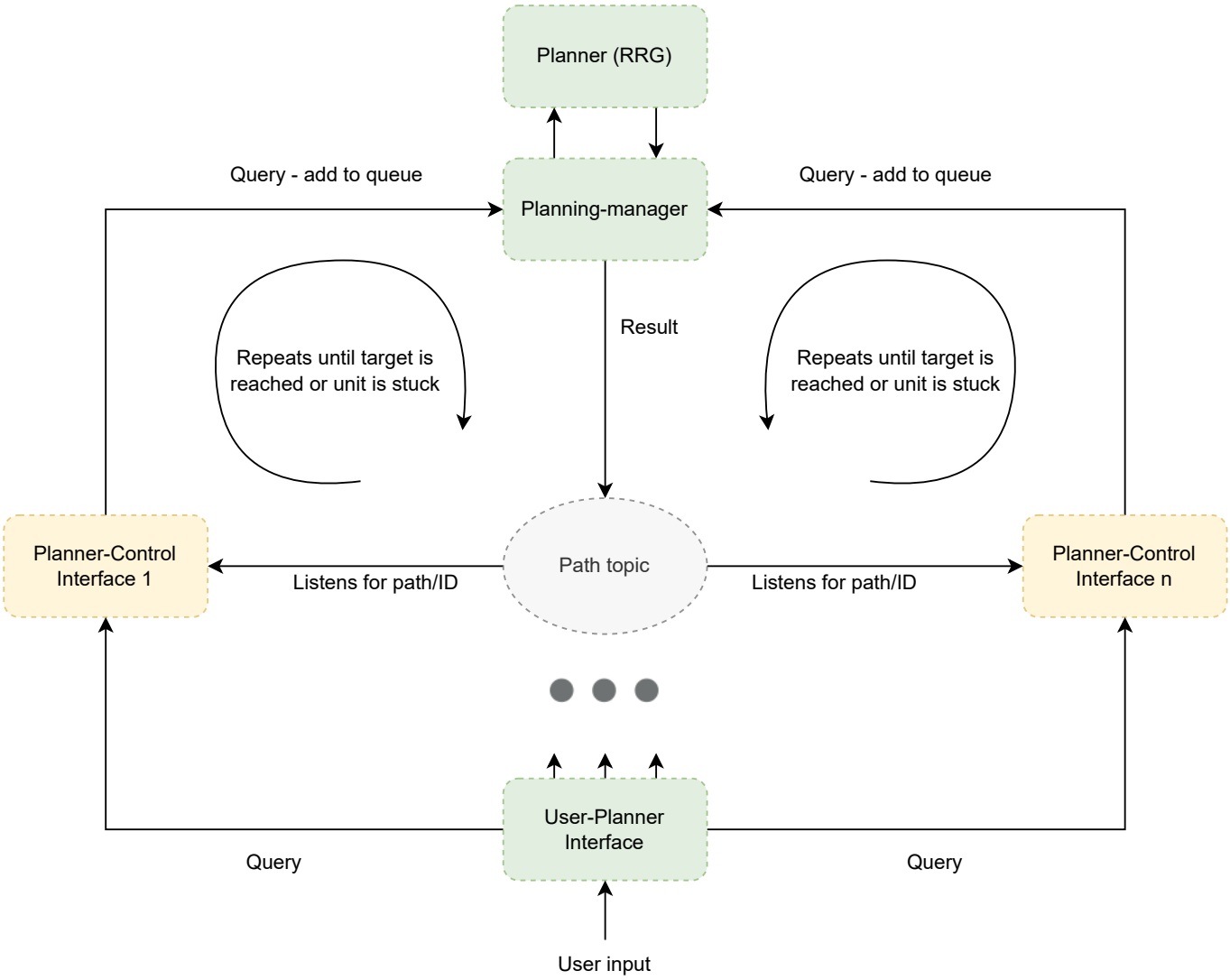


Figure 6: The queue-based system architecture. Green boxes indicate that the functionality they provide was developed in this thesis, while yellow indicates largely pre-made modules.

3.4.1 Graph management and core planner

The core planner's task is to expand the global graph towards given goal regions while maintaining the graph to be used for multi-query purposes. It should prioritize real-time constraints to deliver an RTS-like experience while still providing robust and cohesive planning. The planner needs a map for collision-checking while expanding the graph provided by the VoxBlox module described in section 3.3.1. VoxBlox subscribes to a ROS topic where sensor data from a unit is published to create the map. When dealing with several robots, combining the sensor data from all units is preferred to create a single occupancy map for collision checking. Having a single occupancy map lowers the memory complexity of saving and maintaining the map and offers the perk of utilizing the perceived environment of all robots without the planner having to iterate through maps to find the correct information. Perfect localization is assumed for the robots, such that aggregating all the pointclouds into one is sufficient for combining the map. The full architecture of the planner and its relations with the planning manager and unit is depicted in fig. 7. The green boxes indicate the RRG implementation and its internal parts, while the blue boxes are pre-made modules and config files used inside the planner. The red boxes are parts of the Planning manager, responsible for managing the queue, publishing results, and aggregating the sensor data. The yellow boxes are units themselves, described in section 3.2.

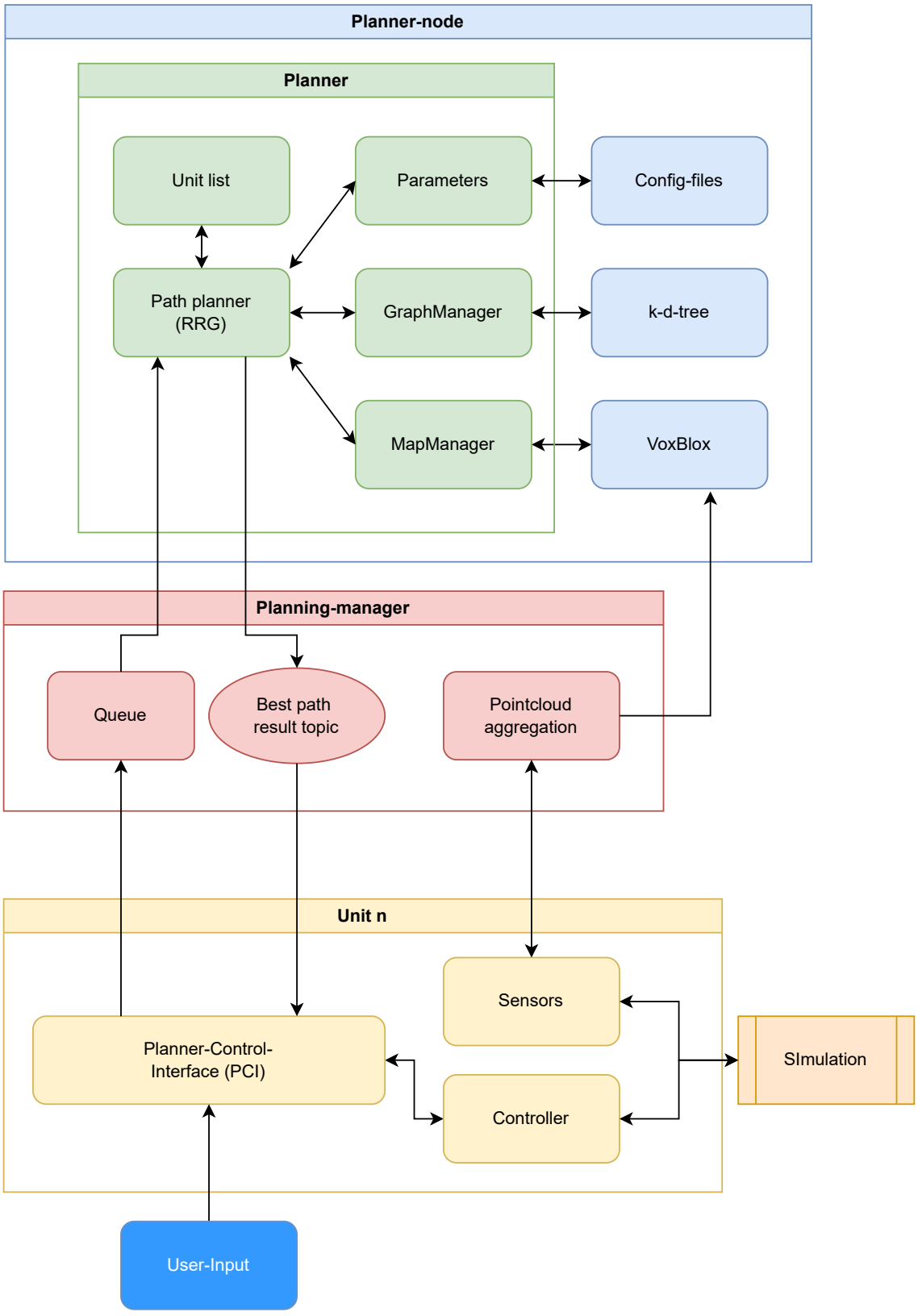


Figure 7: The full architecture of the developed multi-agent planner system.

The implementation is primarily based on the RRG implementation described in section 2.1.9, but some modifications had to be done to account for additional constraints. The planner operates in an unknown environment, and goal regions may therefore lay outside of the mapped space. A

modification to the problem statement is required to account for this challenge.

Recall that χ denoted the full configuration space and was divided into the obstacle space χ_{obs} and free space $\chi_{free} = \text{cl}(\chi \setminus \{\chi_{obs}\})$. To account for unmapped areas, $\chi_{unknown} = \chi \setminus \{\chi_{obs} \cup \chi_{free}\}$ is introduced as a new subspace. As presented in section 2.1.1, a part of the definition of a feasible path is that the path $\sigma(\tau)$ must fulfill $\sigma(1) \in \text{cl}(\chi_{goal})$. If $\chi_{goal} \in \chi_{unknown}$ the condition that a path must be collision-free $\sigma(\tau) \in \chi_{free}$ for all $\tau \in [0, 1]$ fails. To handle this issue, the planner must produce shorter path segments completely in χ_{free} while iteratively expanding the graph when new sensor data is available until $\chi_{goal} \in \chi free$ and thus reachable by the planner. The desired property of asymptotically optimal global planning can not be reached when the whole configuration space is not observed. However, the planner may still aim to produce near-optimal *committed* paths within the mapped area. The committed paths are produced for each planner iteration's first element in the query queue. In every iteration, the following happens.

1. The active unit's current position is checked in relation to the graph. If it is close enough to an existing vertex, its position will be added as a new vertex to the graph. Otherwise, the unit's position is added as a disconnected vertex to start a new graph segment.

2. The current goal region is connected or disconnected from the graph. If it is connected, the planner provides the resulting path and moves on to the next query in the queue.

3. The planner samples points from χ inside $V_{planner}$ and checks if they are obstructed, free, or unknown. If the sampled point is inside χ_{free}, it is approved, and the graph expands towards the sampled point. The tuning variable N_V decides how many vertices should maximum be added in each iteration before moving on.

4. Edges are added from the newly added vertex V_{new} to the nearest vertices inside some range d_{near}. The tuning variable N_E decides how many edges should maximum be added in each iteration before moving on.

5. The active unit's next waypoint is decided, and its path is passed to the planning manager. If a vertex V exists in the same connected graph segment with position $x_V \in \chi_{goal}$, the algorithm will provide a path to the goal. The goal region may lie inside an obstacle, $x_V \in \chi_{obs}$. In this case, the planner provides a path to the closest vertex V_n of χ_{goal} called the *target neighbor*. If no point inside χ_{goal} is sampled, an intermediate waypoint is determined by minimizing the euclidean distance from each vertex to the goal. The heuristic is

$$h(V) = c(line(x_V, x_T)) \tag{3}$$

   where $c(line(x_V, x_T))$ returns the cost of the line segment between vertex V's position x_V and the target position x_T.

6. Dijkstra's algorithm[50] is used to find the shortest path on the graph from the unit's position to the current waypoint.

The random sampler is limited to selecting points within a specific area around the robot's position, known as the planner's *local bounding box*, in order to improve the chances of finding a point that can be connected to the existing tree. The size of this area should be determined based on the environment and task at hand. When working in an unknown area, the initial local bounding box should be at least slightly smaller than the sensor range to increase the chances of finding points within the mapped area. The bounding box could also be dynamically adjusted as the graph grows. A measure of vertex density in the graph could tell the sampler to exclude areas with sufficient coverage to increase the probability that valuable points are being added to the graph further. To limit the scope of this project, a static bounding box of 20 m x 20 m x 2 m is used with the robot's position as the center, while the tuning variables mentioned above will be used to affect the planner's performance.

Algorithm 3 Main planner loop

$it \leftarrow 0$
$V_{unit}.x \leftarrow x_{unit}$ ▷ Update unit vertex state with current position
$G \leftarrow V_{unit}$
if isTargetConnected$(x_{target}, V_{unit}, G) \cap NumWaitingUnits > N_u$ **then**
    return GetShortestPath$(V_{target}, V_{unit}, G)$
end if
while ReachedTarget $== False \cap$ card$(V) < N_V \cap$ card$(E_{added}) < N_{EO} \cap it < N_{it}$ **do**
    $it \leftarrow it + 1$
    $V_i =$ SampleVertex$_i$
    if expandGraph$(T = (V, E), V_i) == false$ **then** ▷ Check if vertex is successfully added to tree
        continue
    end if
    if $c(line(V_i.x, x_{target}))$ <targetRadius **then** ▷ Check if added vertex is close enough to target
        WP $\leftarrow V_i$ ▷ Update waypoint as target
        ReachedTarget $\leftarrow true$
        break
    end if
    if ReachedTarget $== False$ **then**
        $WP \leftarrow$ getNearestVertex(x_{Target}, G)
    end if
end while
return getShortestPath$(\text{WP}, V_{unit}, G = (V, E))$ ▷ Return shortest path to waypoint

Algorithm 3 describes the main loop of the path planner. The first thing that happens in each iteration is checking if a feasible path already exists from the unit to its goal in the graph. To prioritize short planning time to allow for more robots, an eventual preexisting path can be returned without improvement if the queue size is larger than the variable N_u. N_u is a variable communicated from the planning manager to the planner expressing how many units are currently waiting for their next path segment. The variable is included to enable the planner to balance quick query results and being able to exploit RRG's asymptotic optimality described in section 2.1.5 through sampling more points. N_u could be a percentage of the total amount of robots or an absolute number telling the planner to quickly return a path if the queue grows to that size. It could also be manually tuned and remain constant if this fits the use case. $N_u = 0$ makes the planner immediately return an existing obstacle-free and feasible path if it exists. In contrast, N_u larger than the actual number of units makes the planner aim to improve the path by sampling more points. $N_u = 0$ will be used for the rest of this thesis. The code for the main planner loop is enclosed in appendix A, while the **ExpandGraph**-function is included in appendix B.

If a path does not initially exist, the planner expands the graph. The planner has four termination triggers for the graph expansion that can be tuned to control the length of each iteration. The variable *it* is a basic incremental variable that decides the maximum number of points the planner should try to add to the graph. N_V and N_E are, respectively, the maximum numbers of vertices and edges added to the graph in each planning iteration. The last termination case is if the planner manages to find a path to the final target, in which case it will return the path to the target. *it*, N_V, and N_E affect the duration of the graph-expansion phase and should be tuned to fit the operational environment, temporal constraints, and the number of units. The tuning of these variables will be discussed in section 4.1.

Suppose the destination is not within the mapped area and, therefore, cannot be reached by the tree created by the RRG algorithm. In that case, the next destination is chosen using the heuristic described in eq. (3). This heuristic is simple but effective in selecting an appropriate intermediate target waypoint. This path planner is intended to be used in a real-time strategy game-like environment where an operator, whether it is an AI or a human, will oversee the robot's actions. Due to this, the robot does not require advanced autonomy. The simple heuristic does make the planner prone to finding some local optimum or "dead end" where it is currently on the

best vertex in the graph and unable to expand closer to the goal region. If the planner causes the robot to become stuck, it will return an empty path to the planning manager, indicating that the unit needs to be helped by the operator directing it to a different target. A stuck robot may also get help from other units, coincidentally expanding the graph in the right direction, but this can not be guaranteed. The fully implemented behavior of the planner is illustrated in fig. 8.

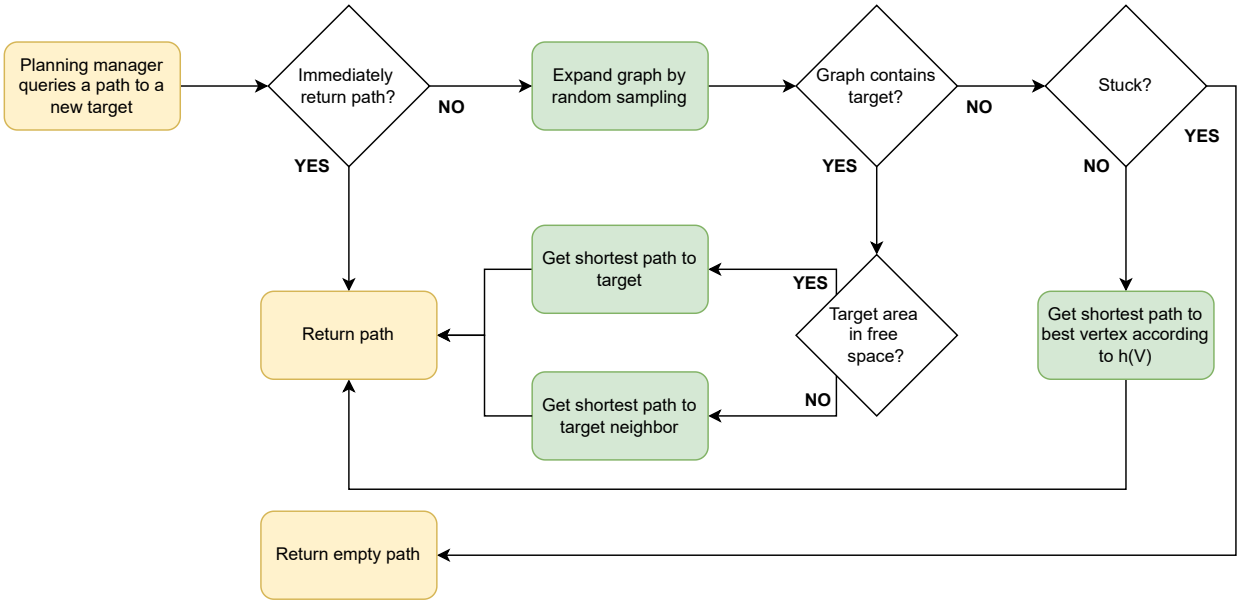


Figure 8: The behavior of the implemented planner.

3.4.2 Planning manager

The main idea behind the planning manager is to utilize computational downtime when units are moving to improve current paths or start planning for other units. Using a FIFO queue-based system simplifies the multi-agent planning problem to a sequence of single-planner problems, which are solved by the planner described in section 3.4.1. As RRG is both anytime and asymptotically optimal, it can produce a resulting path quickly, with increasing quality the more time it gets to expand its graph. There exists only one instance of the planner and planning manager independent of the number of units, meaning that more units give longer queues and, thus, longer waiting times for each unit. Tuning the planning variables mentioned in section 3.4.1 can help the planner become more efficient, although producing worse paths as RRG samples fewer points. Decreasing the planning time for each iteration also increases the probability of RRG failing to produce an obstacle-free feasible path due to the probabilistic completeness of the algorithm described in section 2.1.4. In short, shortening planner time yields diminishing returns as the number of robots grows, and another approach needs to be considered. The idea behind this implementation is that the queue system, which is effective for medium-sized teams, can be expanded by adding sophisticated swarming or following algorithms to make the units move in cohesive teams treated as one unit by the planning manager. This design choice could make it possible for large-scale robotic team control by globally abstracting the units to simplify the problem and solve cohesive movement locally. The addition of swarming or unit following into this system is outside the scope of this thesis and should thus be researched further.

The planning manager keeps track of all units, their positions, and their goals while communicating with each unit through their respective PCI mentioned in section 3.2. After the PCIs execute an initialization movement, the units constantly transmit their current positions and goals to be used by the planner. This information is also used to aggregate the pointclouds used in the creation of the VoxBlox map. The system architecture's mode of operation is centralized and communicates on a need-to-know basis. Hence all mission-critical information is kept at the planner/planning-manager level in the hierarchy shown in fig. 5. This design trait causes the robots to be unaware

of each other, thus marking friendly units as obstructed. Viewing other robots as obstacles seems beneficial for collision avoidance but pose a problem when iterating through the queue. As an aggregate of all pointclouds is used for collision-checking, each robot exists inside an obstructed area in the VoxBlox-map. A pointcloud filter was added to tackle this problem by removing all points in a sphere around the units, thus making them invisible to each other. The filter does not affect obstacle avoidance between units, as the planner has all robots' current positions and paths.

3.4.3 Dynamic Obstacle Avoidance

To account for several units operating in the same space, some form of dynamic obstacle avoidance method must be in place. Dynamic obstacle avoidance is not the main focus of this thesis; hence the method chosen was simplistic to limit the time used in the implementation phase. The method needed rather be more conservative than liberal to ensure that the robot did not collide with each other. The collision check happens after the planner returns the path to the planning manager before delivering the path to the robot for execution. The implemented collision checking method consists of the following procedures.

- Partitioning the path of the current robot u_a into segments consisting of straight lines between the vertices of the path.
- Predicting the time u_a will be at each vertex and each segment midpoint.
- Iterating through all the other units u_c. If the u_c is currently moving, the same partitioning and time prediction is made to this unit's current path. If the unit is idle, only the unit's current state needs to be checked for collisions.
- If the path of u_a intersects with any of the paths of moving u_c at similar times or states of idle u_c, a possible collision is detected, and u_a's path is deleted. No re-planning method has been implemented in this thesis if a possible collision is detected. The planner simply returns an empty path and asks the operator to re-queue a different target or at a different time.

3.4.4 User experience

One of the goals of this thesis is that the developed multi-agent planner should deliver an RTS-like experience for operators when commanding units. A subset of more concise design objectives was set, drawing inspiration from the fields of HCI and HRI mentioned in section 1.4.3 to achieve the goal of delivering an RTS-like experience. The first subgoal is that the proposed interface design should enable the operator to understand the current progress of mission execution. Clear and concise feedback and visualization are needed to ensure the operator's interpretation is straightforward. There are some limitations in the visualization software rviz, so a terminal is used alongside it to provide feedback messages. The messages are limited to a single terminal for all unit- and planner-related feedback to minimize screen pollution. Using a single terminal for all feedback messages can lead to an overload of information from a single source. However, it is counteracted by only printing information in the terminal that is not exhibitable in rviz. The messages are easily customizable as print statements in the system code and can be tailored to a specific use case if needed. For this thesis, the messages have been limited to informing the operator of graph expansion updates, path execution status, queue status, and error messages. An example of a terminal showing text-based feedback is depicted in fig. 9. All planner or PCI messages are marked with a unit ID, while messages originating from the planning manager are marked with MANAGER.

```
  UPI: You clicked, 21.924503, -0.425629 for unit 2
MANAGER: Order from unit 2 added to queue
MANAGER: planner for unit 2 starting
Graph is empty
UNIT 2: Target is too far from the established graph, starting sampler to expand
Formed a graph with [101] vertices and [1122] edges with [119] loops, ntn[0]
UNIT 2: Target not yet reached by roadmap, updated waypoint as best vertex
UNIT 2: No collisions detected
MANAGER: Best path found for unit 2
PCI: Unit 2 updated current path
Reconnect done
Waiting done.
PCI: Unit 2 not yet reached target, requeuing
MANAGER: Order from unit 2 added to queue


  UPI: You clicked, 14.336237, -0.293986 for unit 1
MANAGER: planner for unit 2 starting
MANAGER: Order from unit 1 added to queue
UNIT 2: near a vertex
UNIT 2: Successfully added current state to graph
UNIT 2: Target is near a vertex
UNIT 2: Could not successfully add current target to graph
Formed a graph with [202] vertices and [2969] edges with [135] loops, ntn[2]
UNIT 2: No collisions detected
MANAGER: Best path found for unit 2
MANAGER: planner for unit 1 starting
PCI: Unit 2 updated current path
Reconnect done
UNIT 1: on a vertex
UNIT 1: Target is near a vertex
UNIT 1: Successfully added current target of to graph
UNIT 1: Target already connected! Returning path without expanding graph
Formed a graph with [203] vertices and [2970] edges with [0] loops, ntn[1]
UNIT 1: No collisions detected
MANAGER: Best path found for unit 1
PCI: Unit 1 updated current path
Reconnect done
Waiting done.
PCI: Reached final target for unit 2, no need for requeuing
Waiting done.
PCI: Reached final target for unit 1, no need for requeuing
```

Figure 9: Text-based feedback

The GBplanner visualization package mentioned in section 3.1 is used for the visualization of robots, their paths, and the environment in the rviz main window. A unit and its current path visualized in rviz are pictured in fig. 10. The rviz main window requires the operator to pan and move the camera around manually to localize the robot they want to inspect, which may be challenging when working with a multi-agent system in a large environment. An overview of the total explored area, as well as the units and their positions, is provided by a minimap developed for this thesis. The minimap is a dynamic image visualized in a tab in rviz. It communicates with the planning manager to constantly have updated positions and map information. A depth image style was used for the minimap to visualize a three-dimensional environment in 2D. The elevation of obstacles is illustrated by gradually changing pixel color by z-coordinate. Each robot was given a color corresponding to its unique identification number to differentiate between the units. This color was also used for coloring the paths in the main rviz window to make the association between the minimap and the main window simpler for operators. The resulting minimap is shown in fig. 11 where four units operate in an open, canyon-like environment. The minimap shows unit positions as colored squares, their targets as colored crosses, and the environment as a gray-scaled depth image.

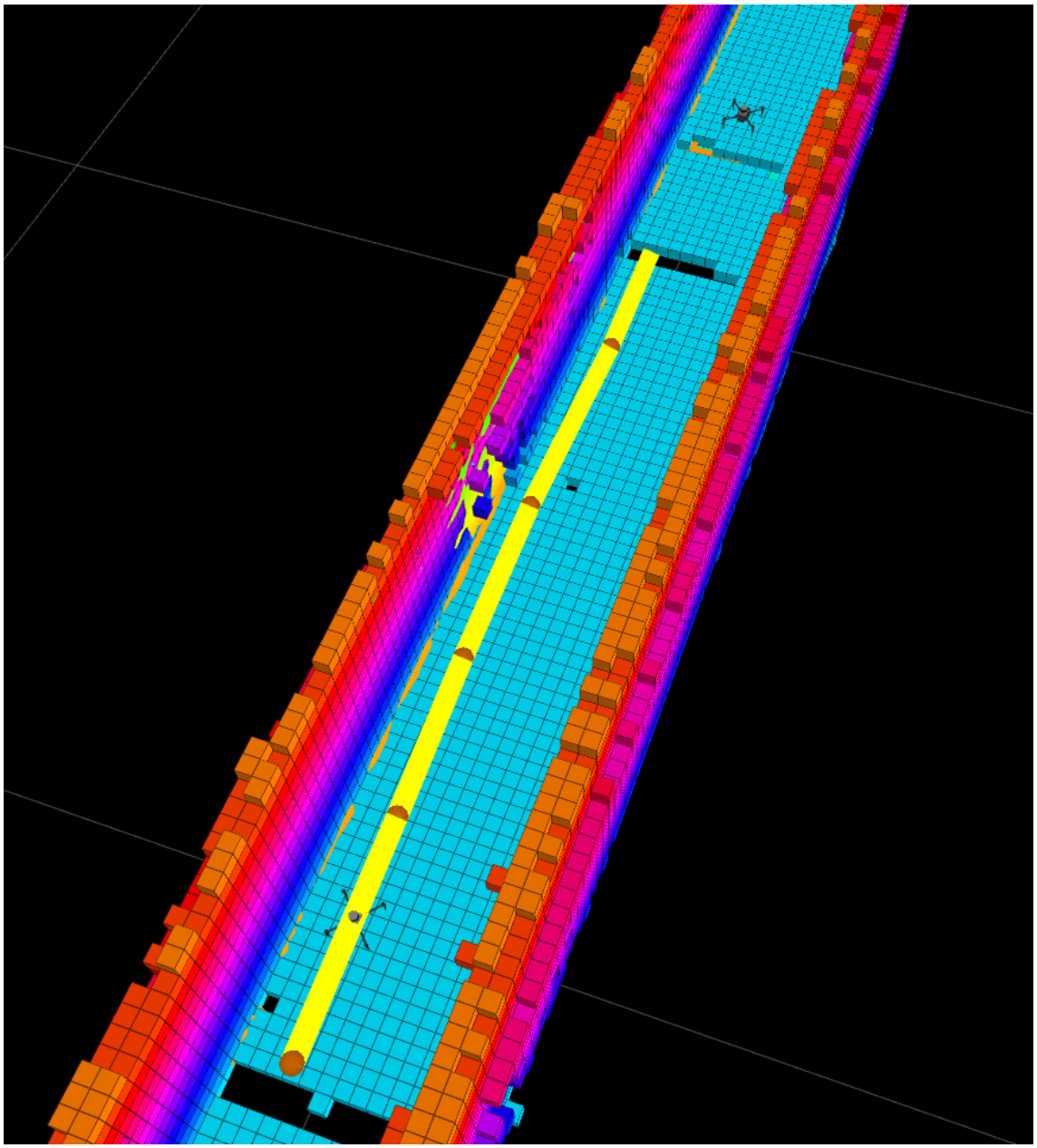


Figure 10: The rviz main window showing a robot and its current path in a narrow environment. The moving unit is shown inside the path in the bottom right, while an idle unit can be seen in the top right.

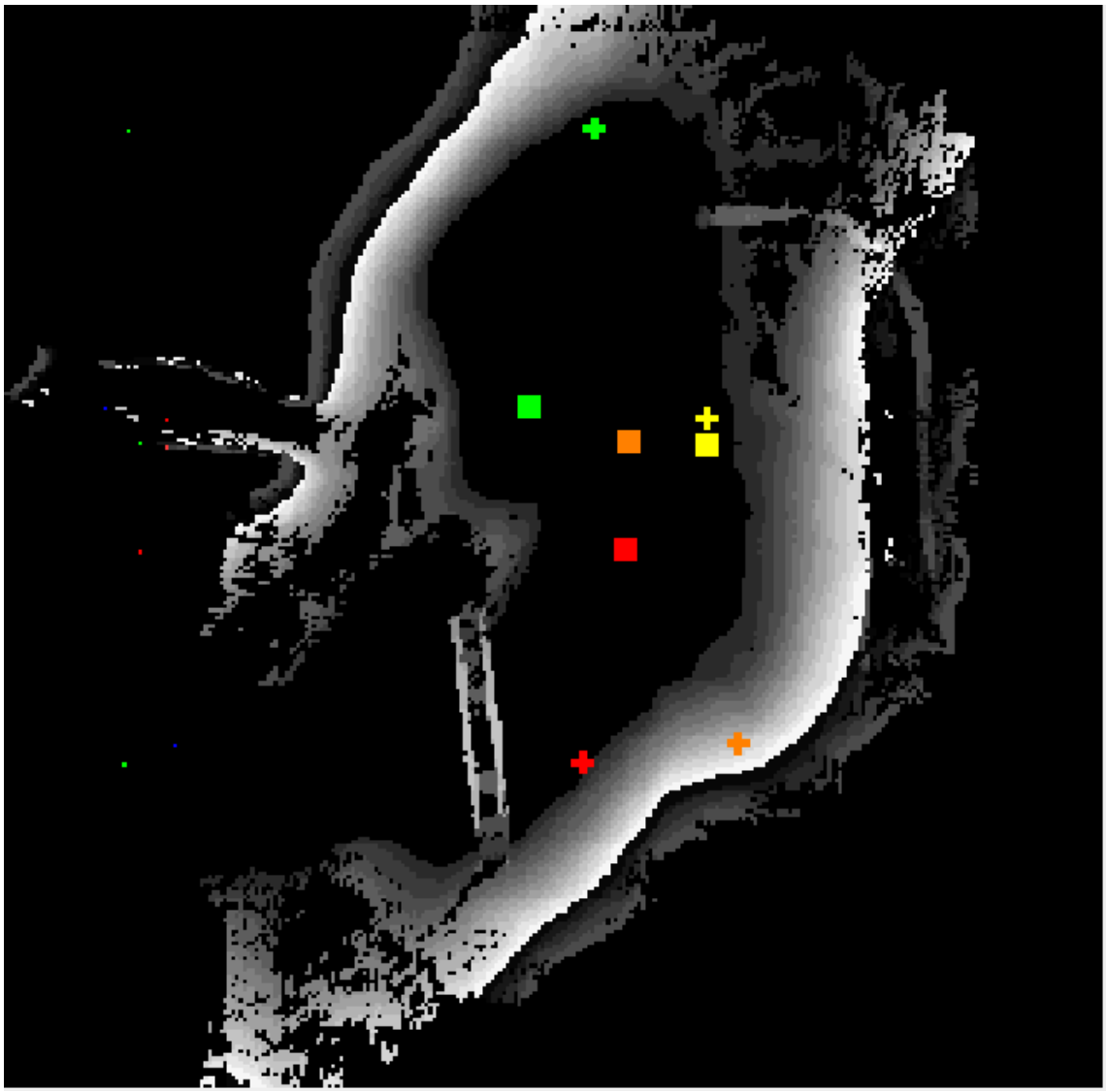


Figure 11: The minimap showing unit positions, squares, and their targets, crosses. Each color corresponds to a unit/target pair.

The second subgoal is that the interface should offer intuitive and simplistic control of where the units move. The chosen approach for this is utilizing the existing rviz tool called *Publish Point* to broadcast a 3D point clicked in the rviz main window to the **/clicked_point_[unit_id]** topic. The developed *User-Planner interface*(UPI) module shown in the architecture drawing in fig. 6 listens on all clicked point topics and relays the point to the correct PCI based on the unit identification number in the topic. The PCI then enters the loop of querying the planner for a path to the clicked point until it is reached or the unit is stuck. The clicked point tool is somewhat limited as it only detects clicks where the ray from the mouse click collides with a texture. This limitation means that only mapped voxels are viable for this user input method. However, it is possible to query the robots to targets in unexplored areas by using the existing ROS functionality of accessing the UPI node through a terminal. The UPI only extracts x- and y-coordinates from the clicked points, keeping the z-coordinate to $2m$ above ground level. The main window in rviz should be angled to a top-down view to get the desired behavior of the clicked point function.

The last subgoal is related to the planner itself and states that latency should be minimized in the system to achieve RTS-like adherence to real-time constraints. The latency minimization is considered during tuning and configuring the planner parameters in section 4.1. Achieving low latency is also kept in mind for the feedback message system, as the operator gets affirmations from the UPI momentarily after ordering a robot and error messages as soon as they occur.

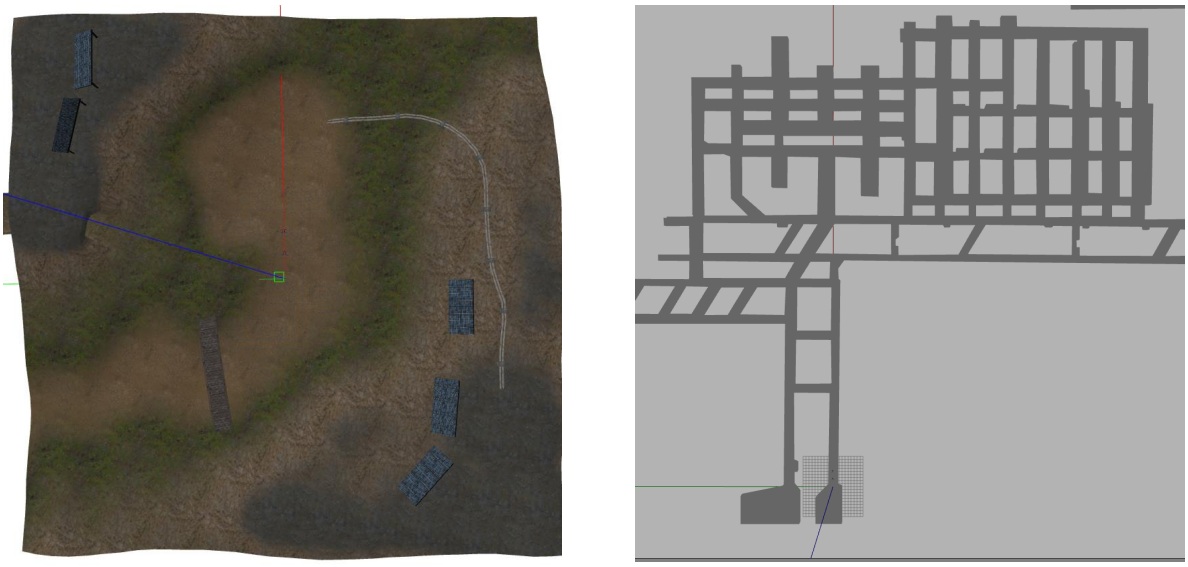
4 Results

This section presents a thorough analysis of the multi-agent planner's performance. The planner will be tested in various environments to showcase the operating environment's impact on path quality and graph maintenance. Firstly, the planner will be tested in a canyon-like open environment with few obstacles other than smooth elevation changes. The second environment is a narrow labyrinth-like environment where robot motion is naturally much more constrained. The results will lead to a proposed configuration of planner parameters that fit the different environments and an analysis of how the parameters should be tuned to fit other operating environments. The values presented are coarsely tuned to showcase the impact they have on performance. Other values may result in better planning, especially in different operating environments. None of the tests resulted in collisions with the environment or between units as the rather conservative collision avoidance module described in section 3.4.2 was used. Finally, some tuning results concerning the user experience will be presented to provide insight into how well the planner system succeeds in delivering an RTS-like experience.

4.1 Computational results

To analyze the performance of the planner, planning with a team of six robots will be performed in two different environments. The developed planner in this master's thesis is asynchronous, meaning that the timing and order of target commands will significantly impact planner behavior. The targets will be issued to the planner at approximately the same intervals by a human operator to make the tests as comparable as possible. This is to get results corresponding to the intended use-case where an operator, human or AI, commands the units. Additionally, the operator will pick the targets pseudo-randomly by clicking the map to achieve some degree of diversity in the tests conducted.

The planner will first be tested in an open valley-like environment with a few static obstacles as well as some changes in the ground level to show that the planner handles elevation differences as well as horizontal planning. The open environment is from an open-source Gazebo world project called Gazebo models and worlds[63]. The planner will then be tested in a more open-spaced environment to showcase the impact differences in environments make on performance. The second environment is a flat, narrow labyrinth-like environment where the number of static obstacles is high, but the feasible operating space for a robot to move in is more limited than the first. The narrow environment is a model of the National Institute for Occupational Safety and Health (NIOSH) mine in Pittsburgh. The model is retrieved from the Autonomous Robotics Lab at NTNU, which was used in the DARPA-challenge[2] under the development of the exploration planner GBplanner[1]. The environments are shown top-down in fig. 12 where they are visualized in Gazebo.



(a) (b)

Figure 12: The two environments used for planner testing visualized in Gazebo. The open environment is shown in (a), while the narrow environment is shown in (b).

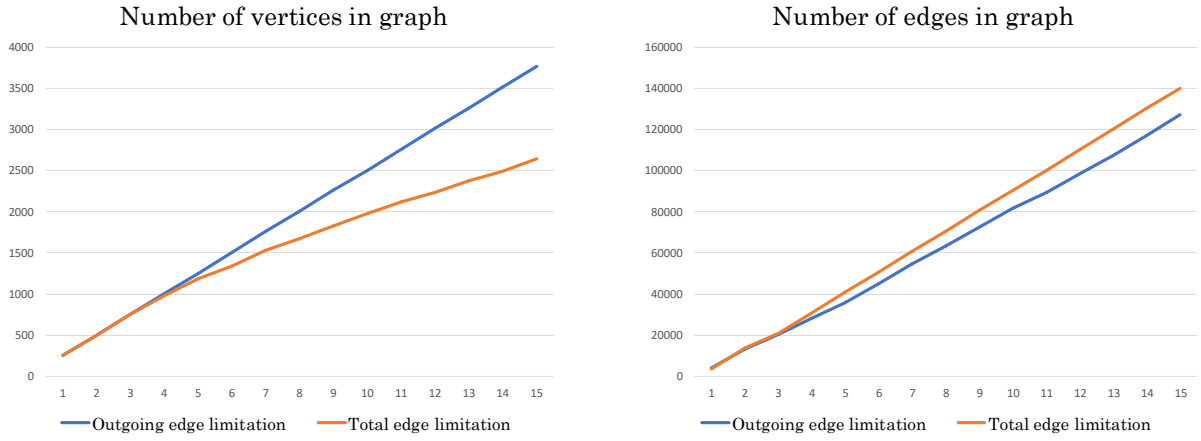
4.1.1 Tuning parameters

Several parameters influence the performance in addition to the environment when testing the planner. Some of the parameters are highly correlated and must be tuned with some coherence to achieve satisfactory performance. The tuning parameters deemed the most important are

- N_V, the maximum number of vertices added to the graph each planner iteration. This parameter directly influences how good coverage the graph achieves over a specific volume, as well as how quickly the planner approaches optimal path planning as discussed in section 2.1.5.
- N_E, the maximum number of edges added to the graph each planner iteration.
- N_{EO}, the maximum number of outgoing edges per vertex. This parameter greatly influences the connectivity of the graph, as well as path optimality.
- $d_{E,max}$ and $d_{E,min}$ are, respectively, the maximum and minimum lengths of edges added to the graph. These parameters decide the step resolution of the graph while also affecting the maximum reach of the graph expansion per iteration.
- r_T is the radius of the sphere with the target point as the center. The sphere is called the target area and should be adjusted to fit the accuracy requirements of the planner.
- $V_{planner}$ is the bounding box of the planner, described in section 3.4.1, which defines the volume from where the random sampling algorithm may pick points.

The scope of the testing was limited by pre-tuning some of the parameters to reasonable values that produced acceptable results. As the planner is to adhere to strict real-time constraints mimicking the behavior of an RTS game, one of the most important performance metrics is planner time. Total planning time can be divided into graph expansion time T_G and path extraction time T_P, the latter including the simple collision checking routine described in section 3.4.2. Through the testing, it was found that path extraction using Dijkstra's algorithm gave negligible contributions to the total planner time, making T_G the most important metric. This trend can clearly be seen in

the fourth column in both table 2 and table 5. The graph expansion part of the planner is RRG-based and has, as mentioned in section 2.1.9, a processing and query complexity of $\mathcal{O}(n \log n)$, where n is the number of samples added as vertices to the graph. This means that N_V, along with N_E, directly affects T_G. When having a limit on N_E, the planner exhibited a behavior where the number of it would reach N_E each iteration before reaching N_V, with increasing pace. The reason for this effect is that as the multi-query graph grows, there are more vertices to connect with for each vertex added. The limitation of N_E resulted in a slow stagnation of vertices added while the graph grew. To account for this issue, N_{EO}, which limits the number of outgoing edges per vertex, was introduced to enable the planner to add more vertices without reaching N_E too quickly. Testing with respectively $N_{EO} = 80$ and $N_E = 10000$ gave similiar growth of $N_{E,total}$, however, N_{EO} did not stagnate N_V. The results from this test can be seen in fig. 13. Using N_{EO} distributes the edges more evenly among the vertices and will always result in less than, or equal to, the number of edges added with a restriction N_E, making N_E superfluous as a tuning variable and was therefore omitted from further testing.



(a) (b)

Figure 13: (a) Shows the evolution of N_V when expanding the graph over 15 planner iterations, while (b) shows the evolution of N_E

An important criterion of the planner is to reach a given target area χ_{goal}. The planner's ability to produce a path to the target area depends on the random sampler picking points to add to the graph. The probability that a sampled point is in the obstacle-free volume V_{free} is

$$p_{free} = \frac{V_{free}}{V_{planner}}$$

where $V_{planner}$ is the bounding box of the sampler. The following probability that a sampled point is inside χ_{goal}, modeled as a sphere with the volume $V_{target} = \frac{4}{3}\pi r_T^2$, is given by

$$p_{target} = p_{free} * \frac{V_{target}}{V_{free}} = \frac{V_{target}}{V_{planner}}$$

r_T should be adjusted to fit the accuracy requirements of the planning problem, but it is important to note that decreasing r_T too much without adjusting other parameters may yield unsatisfactory results. The planner tended to oscillate around the target area as the probability of sampling inside the target area gets very low. To enable low r_T, N_V needs to be increased as the total probability that a sampled point inside V_T with N_V vertices is given by

$$p_{target}(N_V) = 1 - \overline{p_{target}}^{N_V}$$
$$p_{target}(N_V) = 1 - (1 - p_{target})^{N_V}$$
$$p_{target}(N_V) = 1 - (\frac{V_{planner} - V_T}{V_{planner}})^{N_V}$$

which approaches 1 when $N_V \to \infty$. Decreasing $V_{planner}$ will also result in a larger probability to sample inside V_T, but $V_{planner}$ was held constant as N_V was deemed more interesting to vary because of its direct influence on T_G. It is also important to note that p_{target} only affects the planner if the target area is inside $V_{planner}$. If a target further away is queried, the planner will continue to add vertices until N_V or N_{EO} is reached to find the path to the vertex inside $V_{planner}$ closest to V_T. Tuning showed that r_T should at least be half the size of the robot bounding box mentioned in section 3.2 for the planner to produce paths ending reasonably close to the target point without oscillating due to low p_{target}.

The planner should also provide reasonably short and smooth paths to minimize robot movement, and total distance traveled, as path execution is more time-consuming than planning in most cases. The path lengths were compared to direct lines from the units' current position to their current waypoint to measure the optimality of the paths produced by the planner. Parameters tightly coupled with path shape are the edge length restrictions $d_{E,max}$ and $d_{E,min}$. Having a high $d_{E,min}$ and $d_{E,max}$ limits the planner to only taking large steps between vertices, which respectively decreases the graph's maximum and minimum resolution. Larger steps make for worse collision avoidance as longer edges give coarser turns in the returned paths. Additionally, too high $d_{E,min}$ often resulted in the mentioned oscillation around V_T as the short edge needed for the robot to enter V_T could not be added. Lowering $d_{E,max}$ and $d_{E,min}$, respectively, increases the minimum and maximum resolution of the graph, as longer edges are not added. However, this restricts the reach of the graph per iteration if N_V is not increased accordingly, resulting in more planner iterations needed for the robots to cover a distance. To keep the perk of having smooth turning with high maximum graph resolution, as well as the ability to travel long straight distances where possible to shorten the total path length, a reasonably large interval $d_E = d_{E,max} - d_{E,min}$ was chosen. The parameters were held constant through testing at $d_{E,min} = 0.3$ and $d_{E,max} = 6$, which gives the planner freedom to add edges of fitting length, although with less predictability than a lower d_E.

4.1.2 Planning in an open environment

A total of 13 tests were conducted to investigate the impact of N_V and N_{EO} on T_G in an open environment and how they affect path optimality. Each test consisted of querying the planner between 20 and 30 times to command the robots around in the environment depicted in fig. 12 (a). A team consisting of four to six robots was used. The units were initialized in the center of the map, depicted in fig. 14 as a blue frame, at $[x, y, z] = [2.5i, 0.0, 1.5]$, where i is the unit ID number of the robot going from 0 to $N_u - 1$. The planner was tasked with moving the units to different map sections, depicted as white frames in fig. 14.

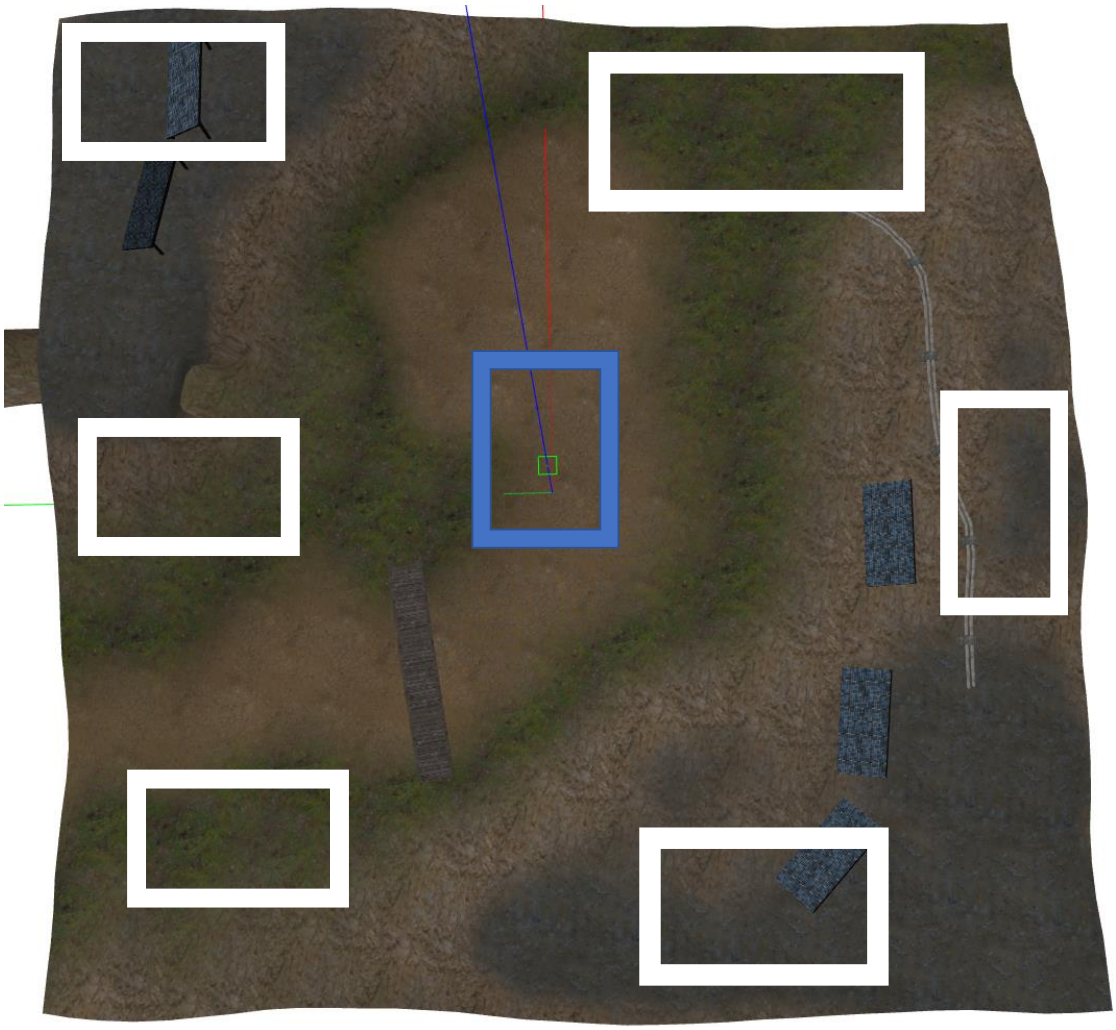


Figure 14: The planning problem for performance testing in an open environment. White frames symbolize the target areas, while the blue frame encloses the starting area of the units.

The performance results are shown in table 2, where each column's best value(s) is highlighted in bold. The path extraction time T_P results showed that it was negligible, as expected, in all cases compared to T_G. The lowest pairing of N_V and N_{EO} resulted in the lowest average graph building as expected. Having $N_V = 50$ consistently gave quick planning but also had the worst overall path quality, with paths around 20% longer than the optimal path. This result is intuitive to understand as fewer vertices give fewer configuration options for the planner to build a path. Additionally, the data path optimality results for $N_V = 50$ are skewed due to the planner producing short straight paths around the target area due to the oscillation described in section 4.1.1. Doubling r_T allowed the planner to more quickly find a bath to V_T without oscillating and gave even worse path optimality results around 40% longer than the optimal path. These results and their standard deviations can be seen in table 3, exhibiting by far the worst path optimality. As seen in table 2, increasing N_V generally resulted in better path quality, although only up to a certain point. Having $N_V = 500$ and $N_{EO} \in [20, 40, 80]$ gave as worse paths as $N_V = 50$. The reason for this is that the planner had a too harsh restriction on adding edges. Limiting a sampled point to only connect to the N_{EO} closest vertices when $N_V >> N_{EO}$ made the edge lengths shorten, as most points had many near existing vertices as the graph grew. Having only short edges limited the reach of the graph as N_{EO} quickly was reached, creating clusters of very dense graph parts while unable to reach large parts of the map in only one iteration. The effect was self-reinforcing as dense areas tended to become denser with N_{EO} capping out quicker and quicker. A test was done with $N_{EO} = 160$ to tackle the clustering issue, which resulted in better path quality, but T_G grew to almost 3 seconds which is unacceptably long for a planner with strict real-time constraints.

The clustering effect was also observed with $N_V = 250$ and $N_{EO} = 20$, but the threshold N_{EO} needed for acceptable path quality was much lower than with $N_V = 500$. This threshold is tightly correlated with N_V as N_{EO} needs to be high enough for the planner to approach the practical upper limit of path quality set by N_V by avoiding clustering, giving a more uniformly dense graph. Having $N_V = 250$ and $N_{EO} = 40$ gave satisfactory path quality while also resulting in substantially lower T_G than $N_V = 500$. This configuration also resulted in the most pre-existing path queries as the graph grew uniformly distributed. Having high N_V generally made the planner able to find pre-existing paths for units moving to previously explored space. However, the clustering at $N_V = 500$ again created problems for units that needed to move through sparse areas of the graph. After the units had moved to their initial target regions, they were commanded to another target area.

The standard deviations σ of the results in table 2 can be seen in table 4. The standard deviation of T_G is naturally tightly coupled with the number of vertices, where lower values of N_V generally give more predictable building times. $N_V = 250$ exhibited the best overall consistency in path quality as it was not affected by the clustering as much as $N_V = 500$, while still having enough configuration options to produce good quality paths.

The parameters should be tuned to fit the use case, where real-time constraints must be weighed against path quality desires. To provide scalable, fast planning like an RTS game, lowering T_G should be prioritized over path quality. $N = 100$ gave much lower T_G than $N_V = 250$ while not worsening the path quality substantially as long as N_{EO} was high enough, although the path quality consistency was much worse than $N_V = 250$. $N_V = 100$ also provided an acceptable rate of pre-existing paths found. The parameter configuration of $N_V = 100$ and $N_{EO} = 80$ is recommended for use in an open environment. Having $N_V \in \{100, 250\}$ and $N_{EO} \in \{40, 80\}$ seems, in any case, to provide satisfactory planning in an open environment with a static $V_{planner}$ of 20 m x 20 m x 2 m.

Multi-agent planner performance with varying N_V and N_{EO} in an open environment					
N_V	N_{EO}	Mean T_G[s]	Mean T_P [s]	Pre-existing path rate	Path optimality
50	20	**0.051**	**0**	6.2%	1.214
	40	0.062	**0**	0%	1.166
	80	0.089	0.001	7.7%	1.248
100	20	0.061	**0**	5.0%	1.188
	40	0.131	0.001	15.8%	1.186
	80	0.235	0.002	14.3%	1.106
250	20	0.172	0.001	20.0%	1.235
	40	0.406	0.004	**25.0%**	1.091
	80	0.621	0.005	**25.0%**	**1.070**
500	20	0.566	0.006	10.5%	1.229
	40	1.081	0.010	10.5%	1.185
	80	1.964	0.011	14.3%	1.167
	160	2.693	0.026	24.0%	1.088

Table 2: Performance of the planner with a varying number of vertices N_V and outgoing edges per vertex N_{EO} in the graph in an open environment

Performance results of $N_V = 50$ with increased r_T		
N_{EO}	Path optimality	σ of path optimality
20	1.429	0.463
40	1.365	0.397
80	1.378	0.298

Table 3: Planner performance in an open environment with $N_V = 50$ and doubled r_T

σ of performance results in an open environment			
N_V	N_{EO}	σ of T_G[s]	σ of path optimality
50	20	0.011	0.294
	40	0.018	0.172
	80	0.043	0.290
100	20	**0.009**	0.295
	40	0.025	0.431
	80	0.083	0.216
250	20	0.026	0.443
	40	0.120	0.116
	80	0.160	0.125
500	20	0.116	0.309
	40	0.173	0.215
	80	0.473	0.251
	160	0.724	**0.077**

Table 4: Standard deviation of planner performance with a varying number of vertices N_V and outgoing edges per vertex N_{EO} in the graph in an open environment

4.1.3 Planning in a narrow environment

A total of 12 tests were conducted to investigate the effect N_V and N_{EO} has on planner performance in narrow environments. Navigating several robots in a narrow environment requires more consideration from the operator than in an open environment due to the increased spacial limitations. The operator must wait for a robot to move far enough at the start before commanding another robot to avoid the planner returning with an empty path due to a possible collision detected as the robots are initialized close to each other. The test is designed with exploration in mind; thus, the robots will split from each other at each junction in the labyrinth in the map shown in fig. 12 (b). The units were initialized similarly to the previous test at $[x, y, z] = [2.5i, 0.0, 1.5]$, where i is the unit ID number of the robot going from 0 to $N_u - 1$. Each test was conducted with a team of five robots and ran until the total number of queries was between 20 and 30.

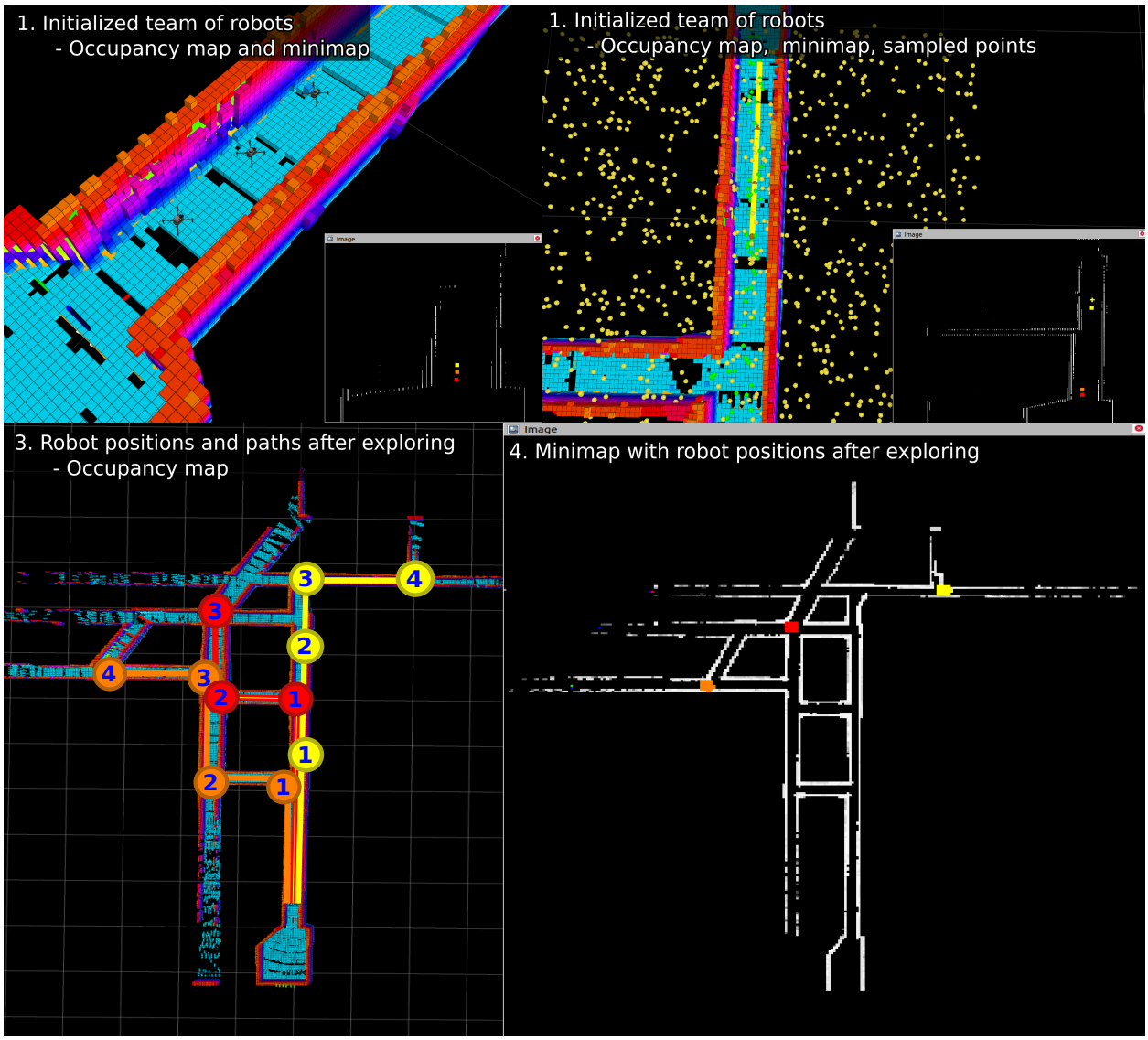


Figure 15: An instance of teamed exploration in a narrow subterranean environment. Frame 1 shows the initialized robot team and VoxBlox occupancy map in the main rviz window, and the minimap depicts obstacles and robot positions. Frame 2 shows robot 2 after two iterations. Robot 2 has a yellow color code, and the current path of the robot is shown as the yellow line. The yellow points are rejected samples, while the green points are accepted samples. Frame 3 shows the top-down view of a semi-explored environment in the rviz main window after 3-4 planning iterations per robot. Representations of the executed paths are drawn in the robot color codes, with numbered circles representing the orders each robot received. Frame 4 depicts the minimap at the same timestamp as frame 3, providing a simplistic view of the whole environment and clearly showing the robots' positions. A demonstration of a teamed exploration mission similar to the one depicted in this figure can be found at https://youtu.be/cdr3fuli5d0.

The performance test results in a narrow environment can be seen in table 5, while the standard deviations of the results are shown in table 6. The graph-building time results, and their standard deviation, were similar to the results from the open environment, as this mostly is a function of N_V and N_E. However, the planner produced much better path quality and path consistency with all parameter configurations due to the nature of the environment. The narrow hallways of the labyrinths are the only part of $V_{planner}$ that is in χ_{free}, which means that V_{free} is much smaller for the narrow environment than the open one, resulting in a much lower p_{free}. As the number of samples taken by the planner is not restricted directly, only the number of vertices added, the planner can create a much denser graph with few vertices inside the smaller $V_{planner}$ than in the

open field from the first tests. This ability could also lead to increased building time, as the random sampler has to provide more points to account for the lower p_{free}, but as the results in table 5 show, this seems to have had little impact on T_G. A smaller volume for the graph to span makes more vertices useful for path construction as there are much fewer directions for the units to move in.

The more restricted planning problem posed by the narrow environment allowed for generally lower N_V, giving a much lower T_G, as it still provided good path quality. As seen in table 5, $N_V = 50$ resulted in paths of almost the same quality as higher values for N_V while keeping T_G much lower. Increasing N_V above $N_V = 100$ only increased T_G without producing noticeably better-quality paths. The path consistency results in table 6 show that the configuration $N_V = 50$ and $N_{EO} = 80$ gave the best consistency, while average graph building time was kept consistently at under 100ms. The rate of pre-existing paths found was also generally higher than in the open environment because of the more restricted operating space. The planner was tested with $N_V = 25$ to limit-test how few vertices were needed in this particular environment. However, this configuration illustrated that there was a lower bound for N_V before the path quality worsened. The results for $N_V = 25$ are skewed due to the same oscillating phenomenon as the planner was prone to in the open environment with $N_V = 50$, only returning short straight paths close to V_T, but using more iterations than should be needed to reach the target area.

Multi-agent planner performance with varying N_V and N_{EO} in a narrow environment					
N_V	N_{EO}	Mean T_G[s]	Mean T_P [s]	Pre-existing path rate	Path optimality
25	10	**0.011**	**0**	30.4%	1.027
	20	0.128	**0**	12.5%	1.147
	40	0.019	**0**	31.6%	1.094
50	20	0.036	**0**	25.0%	1.056
	40	0.057	0.001	36.8%	1.024
	80	0.065	0.001	35.0%	1.019
100	20	0.064	**0**	38.9%	1.021
	40	0.141	**0**	18.6%	**1.016**
	80	0.238	0.004	**45.5**%	1.024
250	20	0.161	0.002	36.4%	1.112
	40	0.425	0.003	36.8%	1.051
	80	0.641	0.005	30.0%	1.035

Table 5: Performance of the planner with varying number of vertices N_V in tree. N_i denotes the number of iterations

σ of performance results in a narrow environment			
N_V	N_{EO}	σ of T_G[s]	σ of path optimality
25	10	**0**	0.193
	20	0.002	0.048
	40	0.505	0.255
50	20	0.017	0.100
	40	0.024	0.023
	80	0.031	**0.017**
100	20	0.009	0.041
	40	0.033	0.037
	80	0.083	0.045
250	20	0.052	0.191
	40	0.129	0.104
	80	0.276	0.069

Table 6: Standard deviation of planner performance with a varying number of vertices N_V and outgoing edges per vertex N_{EO} in the graph in a narrow environment

4.2 User experience

User experience is a complex metric to measure. However, in the context of this thesis, it will come down to how responsive the planner feels for an operator in addition to the visual tools presented in section 1.4.3. The responsiveness of the planner will be judged by the latency time t_L experienced from querying a path to a target until the path is returned and the unit starts to move. t_L was measured during the same tests as in section 4.1 to see how the parameters affected it. The results can be seen in table 7 and table 8. The latency measurements from planner operation in an open environment are expectedly directly correlated with the size of N_V and N_EO. In contrast, the results from the open environment have less correlation. These results likely originate from latency being strongly dependent on query input timing and existing queue size. When testing in a narrow environment, periods between queries had to be longer due to the restricted movement options for the units. Longer planner intervals shorten latency for cases with larger N_V as the planner gets more time to work through the queue before a new order arrives. The latency should be minimized for the planner to deliver an RTS-like experience. However, it could be wise not to use it as the primary tuning measurement due to its strong dependency on the operator.

Average t_L and σ_{t_L}			
N_V	N_{EO}	t_L[s]	σ_{t_L}[s]
50	20	**0.095**	**0.067**
	40	0.19	0.15
	80	0.12	0.077
100	20	0.21	0.34
	40	0.25	0.41
	80	0.24	0.45
250	20	0.23	0.27
	40	0.30	0.33
	80	0.43	0.23
500	20	0.27	0.41
	40	0.40	0.50
	80	0.87	1.2
	160	1.1	0.45

Table 7: Average latency time t_L and standard deviation σ_{t_L} of the planner when operating in an open environment

Average t_L and σ_{t_L}			
N_V	N_{EO}	t_L[s]	σ_{t_L}[s]
25	10	0.097	0.009
	20	0.132	0.205
	40	**0.090**	0.028
50	20	0.12	0.013
	40	**0.090**	0.030
	80	0.097	0.013
100	20	0.099	**0.002**
	40	0.091	0.024
	80	0.096	0.017
250	20	0.092	0.021
	40	0.172	0.076
	80	0.192	0.122

Table 8: Average latency time t_L and standard deviation σ_{t_L} of the planner when operating in a narrow environment

5 Conclusion

This thesis has presented a successful path and motion planner for medium-sized robotic teams, delivering robust collision avoidance of static obstacles and other known units. The planner's core is a rapidly-exploring random graph algorithm based on the RRG used in [1]. The algorithm has been modified with elements from a probabilistic roadmap approach to provide multi-query possibilities to utilize existing paths in previously explored areas. The planner functions in both mapped and unmapped areas as it utilizes an expanding mapping framework based on a combination of truncated signed distance fields for surface detection and euclidean signed distance fields for collision checking. The planner exhibits controller agnosticism, which means it works in conjunction with any MAV with a functioning position controller.

The planner is designed to fit the proposed more extensive RTS-like architecture presented in section 3.4.2 for controlling large robotic teams based on hierarchical AI systems designed for playing RTS games. The architecture allows for a relatively low autonomy level in each unit, making the robots prone to getting stuck in local optima due to the simple heuristic responsible for choosing waypoints when the target area is out of reach. The user experience has been elevated by adding simple visual tools and message-based feedback to keep the operator in the loop to handle stuck robots or other errors. The system employs a queue-based solution to multi-agent path planning to reduce the problem into a series of single-agent path planning problems. This approach gives an asynchronous behavior that mimics the player experience in RTS games. Planning speed has been prioritized in the tuning process to reduce planning latency and exhibit the planner's ability to adhere to strict real-time constraints. There is a general trade-off between path quality and planning time when configuring the planner, which should be tailored to the operating environment. The planner performs considerably worse in open spaces compared to narrow environments due to the random nature of the graph-building algorithm.

5.1 Further work

The proposed multi-agent planner yields satisfactory results; however, it could be further improved. Introducing online dynamic parameter adjusting could drastically enhance planner performance. The handling of the online parameter tuning could be approached by investigating neural-network-based learning methods that adjust parameters based on feedback from the units and environment. The user experience should be researched and tested further by conducting user tests to gather feedback as a part of an iterative implementation process.

The proposed multi-agent planner should be ready for simple control of medium-sized robotic teams, but more research needs to be conducted to achieve large-scale RTS-like control of robots. The interface of the proposed planner requires target points and unit identification numbers, which in this thesis are designed to fit a human operator. The operator could be exchanged with a neural network-based AI or even a finite state machine to further expand the system towards effective RTS-like control of large teams.

Further work includes the development of a cohesive swarming strategy to enable the planner to handle more robots by abstracting swarms as one unit. The units also need a local reactive collision avoidance and path recovery method to account for unknown dynamic obstacles. The system architecture presented in section 3.4.2 should be reviewed and can be used as a sketch for expanding the system to include more functionality, such as surveillance or patrolling. Many elements in the architecture still need to be developed to fully evaluate the whole system, where each part individually needs extensive testing and optimization.

Bibliography

[1] Tung Dang, Marco Tranzatto, Shehryar Khattak et al. 'Graph-based subterranean exploration path planning using aerial and legged robots.' In: *J Field Robotics* 37 (2020), pp. 1363–1388. DOI: https://doi.org/10.1002/rob.21993.

[2] DARPA. *DARPA Subterranean (SubT) Challenge*. https://www.darpa.mil/program/darpa-subterranean-challenge. Accessed: 2022-05-24.

[3] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Noetic. 25th May 2022. URL: https://www.ros.org.

[4] Donato Di Paola, Annalisa Milella, Grazia Cicirelli et al. 'An autonomous mobile robotic system for surveillance of indoor environments'. In: *International Journal of Advanced Robotic Systems* 7.1 (2010), p. 8.

[5] Ross A. Knepper, Todd Layton, John Romanishin et al. 'IkeaBot: An autonomous multi-robot coordinated furniture assembly system'. In: *2013 IEEE International Conference on Robotics and Automation*. 2013, pp. 855–862. DOI: 10.1109/ICRA.2013.6630673.

[6] Benjamin Pereira, Christian Andrew Griffiths, Benjamin Birch et al. 'Optimization of an autonomous robotic drilling system for the machining of aluminum aerospace alloys'. In: *The International Journal of Advanced Manufacturing Technology* 119.3 (2022), pp. 2429–2444. ISSN: 1433-3015. DOI: 10.1007/s00170-021-08483-4. URL: https://doi.org/10.1007/s00170-021-08483-4.

[7] Mihir Kulkarni, Mihir Dharmadhikari, Marco Tranzatto et al. 'Autonomous Teamed Exploration of Subterranean Environments using Legged and Aerial Robots'. In: *CoRR* abs/2111.06482 (2021). arXiv: 2111.06482. URL: https://arxiv.org/abs/2111.06482.

[8] A. Hong, O. Igharoro, Y. Liu et al. 'Investigating Human-Robot Teams for Learning-Based Semi-autonomous Control in Urban Search and Rescue Environments'. English. In: *Journal of Intelligent and Robotic Systems* 94.3-4 (June 2019). Copyright - Journal of Intelligent and Robotic Systems is a copyright of Springer, (2018). All Rights Reserved; Last updated - 2019-12-11, pp. 669–686. URL: https://www.proquest.com/scholarly-journals/investigating-human-robot-teams-learning-based/docview/2086032830/se-2?accountid=12870.

[9] Glen Robertson and Ian Watson. 'A Review of Real-Time Strategy Game AI'. In: *Ai Magazine* 35 (Dec. 2014), pp. 75–104. DOI: 10.1609/aimag.v35i4.2478.

[10] Helen Oleynikova, Zachary Taylor, Marius Fehr et al. *Voxblox: Incremental 3D Euclidean Signed Distance Fields for On-Board MAV Planning*. Autonomous Systems Lab, ETH Zürich. 21st Apr. 2017. arXiv: 1611.03631 [cs.RO].

[11] Mark H Overmars. 'Path planning for games'. In: *Proc. 3rd Int. Game Design and Technology Workshop*. 2005, pp. 29–33.

[12] Association for Advancing Automation (A3). *How Robots Are Taking on the Dirty, Dangerous, and Dull Jobs*. 15th Oct. 2019. URL: https://www.automate.org/blogs/how-robots-are-taking-on-the-dirty-dangerous-and-dull-jobs (visited on 27/01/2023).

[13] Renato Silveira, Leonardo Fischer, José Ferreira et al. 'Path-Planning for RTS Games Based on Potential Fields'. In: Nov. 2010, pp. 410–421. DOI: 10.1007/978-3-642-16958-8_38.

[14] Munir Naveed, D. Kitchin and Andrew Crampton. 'Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games'. In: (Dec. 2010).

[15] Yuxiao Chen, Andrew Singletary and Aaron D. Ames. 'Lidar-based exploration and discretization for mobile robot planning'. In: *CoRR* abs/2011.10066 (2020). arXiv: 2011.10066. URL: https://arxiv.org/abs/2011.10066.

[16] Fedor A. Kolushev and Alexander A. Bogdanov. 'Multi-agent Optimal Path Planning for Mobile Robots in Environment with Obstacles'. In: *Perspectives of System Informatics*. Ed. by Dines Bjøner, Manfred Broy and Alexandre V. Zamulin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 503–510. ISBN: 978-3-540-46562-1.

[17] Vishnu R. Desaraju and Jonathan P. How. 'Decentralized path planning for multi-agent teams with complex constraints'. In: *Autonomous Robots* 32.4 (May 2012), pp. 385–403. ISSN: 1573-7527. DOI: 10.1007/s10514-012-9275-2. URL: https://doi.org/10.1007/s10514-012-9275-2.

[18] Oktay Arslan, Karl Berntorp and Panagiotis Tsiotras. 'Sampling-based algorithms for optimal motion planning using closed-loop prediction'. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 4991–4996. DOI: 10.1109/ICRA.2017.7989581.

[19] Prasanna Velagapudi, Katia Sycara and Paul Scerri. 'Decentralized prioritized planning in large multirobot teams'. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2010, pp. 4603–4609.

[20] Jur van Den Berg, Jack Snoeyink, Ming C Lin et al. 'Centralized path planning for multiple robots: Optimal decoupling into sequential plans.' In: *Robotics: Science and systems*. Vol. 2. 2.5. 2009, pp. 2–3.

[21] Ping Wang and Weihua Zhuang. 'A token-based scheduling scheme for WLANs supporting voice/data traffic and its performance analysis'. In: *IEEE Transactions on Wireless Communications* 7.5 (2008), pp. 1708–1718.

[22] Khalid Al-Mutib, Mansour AlSulaiman, Muhammad Emaduddin et al. 'D* Lite Based Real-Time Multi-Agent Path Planning in Dynamic Environments'. In: *2011 Third International Conference on Computational Intelligence, Modelling & Simulation*. 2011, pp. 170–174. DOI: 10.1109/CIMSim.2011.38.

[23] Seifedine Kadry, Gennady Alferov, Viktor Fedorov et al. 'Path optimization for D-star algorithm modification'. In: *AIP Conference Proceedings*. Vol. 2425. 1. AIP Publishing LLC. 2022, p. 080002.

[24] Faiza Gul, Adnan Mir, Imran Mir et al. 'A Centralized Strategy for Multi-Agent Exploration'. In: *IEEE Access* 10 (2022), pp. 126871–126884. DOI: 10.1109/ACCESS.2022.3218653.

[25] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo et al. 'Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments'. In: *The International Journal of Robotics Research* 29.5 (2010), pp. 485–501. DOI: 10.1177/0278364909359210. eprint: https://doi.org/10.1177/0278364909359210. URL: https://doi.org/10.1177/0278364909359210.

[26] Craig Reynolds. 'Flocks, Herds and Schools: A distributed behavioral model'. In: *Computer Graphics, 21(4)*. 1987, pp. 25–34.

[27] Vinicius Graciano Santos, Mario F. M. Campos and Luiz Chaimowicz. 'On Segregative Behaviors Using Flocking and Velocity Obstacles'. In: *Distributed Autonomous Robotic Systems*. Ed. by M. Ani Hsieh and Gregory Chirikjian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 121–133. ISBN: 978-3-642-55146-8.

[28] Jur van den Berg, Ming C. Lin and Dinesh Manocha. 'Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation'. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2008.

[29] Heba Fathy, Osama Raouf and Hatem Abd elkader. 'Flocking behaviour of group movement in real strategy games'. In: *2014 9th International Conference on Informatics and Systems, INFOS 2014* (Feb. 2015), PDC64–PDC67. DOI: 10.1109/INFOS.2014.7036679.

[30] Peter E. Hart, Nils J. Nilsson and Bertram Raphael. 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[31] Johan Hagelbäck. 'Potential-field based navigation in StarCraft'. In: *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. 2012, pp. 388–393. DOI: 10.1109/CIG.2012.6374181.

[32] Hou Wenjun and Bai Xiudong. 'HCI in real-time strategy games: a study of principals and guidelines for designing 3D user interface'. In: *2006 7th International Conference on Computer-Aided Industrial Design and Conceptual Design*. 2006, pp. 1–6. DOI: 10.1109/CAIDCD.2006.329386.

[33] Jonathan Moizer, Jonathan Lean, Elena Dell'Aquila et al. 'An approach to evaluating the user experience of serious games'. In: *Computers & Education* 136 (2019), pp. 141–151. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2019.04.006. URL: https://www.sciencedirect.com/science/article/pii/S0360131519300855.

[34] Randy J Pagulayan, Kevin Keeker, Dennis Wixon et al. 'User-centered design in games'. In: *The human-computer interaction handbook*. CRC Press, 2002, pp. 915–938.

[35] Holly A Yanco and Jill L Drury. 'A taxonomy for human-robot interaction'. In: *Proceedings of the AAAI fall symposium on human-robot interaction*. 2002, pp. 111–119.

[36] Italo R. C. Barros, Luis F. Costa and Tiago P. Nascimento. 'TurtleUI: A Generic Graphical User Interface for Robot Control'. In: *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*. 2019, pp. 174–179. DOI: 10.1109/LARS-SBR-WRE48964.2019.00038.

[37] Sertac Karaman, Matthew R. Walter, Alejandro Perez et al. 'Anytime Motion Planning using the RRT*'. In: *2011 IEEE International Conference on Robotics and Automation* (2011).

[38] Sertac Karaman and Emilio Frazzoli. 'Sampling-based algorithms for optimal motion planning'. In: *The International Journal of Robotics Research* 30(7) (2011), pp. 846–894. DOI: 10.1177/0278364911406761.

[39] Sertac Karaman and Emilio Frazzoli. 'Incremental sampling-based algorithms for optimal motion planning'. In: *Robotics Science and Systems VI* 104.2 (2010).

[40] Michal Cáp, Peter Novák, Jiri Vokrinek et al. 'Multi-agent RRT*: Sampling-based Cooperative Pathfinding (Extended Abstract)'. In: *CoRR* abs/1302.2828 (2013). arXiv: 1302.2828. URL: http://arxiv.org/abs/1302.2828.

[41] Henri Farreny. 'Completeness and Admissability for General Heuristic Search Algorithms - A Theoretical Study: Basic Concepts and Proofs'. In: *Journal of Heuristics* 5 (1999), pp. 353–376.

[42] Helen Oleynikova, Alex Millane, Zachary Taylor et al. 'Signed Distance Fields: A Natural Representation for Both Mapping and Planning'. In: *RSS 2016 Workshop: Geometry and Beyond - Representations, Physics, and Scene Understanding for Robotics*. Autonomous Systems Lab, ETH Zürich. University of Michigan, 2016. DOI: https://doi.org/10.3929/ethz-a-010820134.

[43] A. Ladd and L.E. Kavraki. 'Generalizing the analysis of PRM'. In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*. Vol. 2. 2002, 2120–2125 vol.2. DOI: 10.1109/ROBOT.2002.1014853.

[44] Michal Kleinbort, Kiril Solovey, Zakary Littlefield et al. *Probabilistic completeness of RRT for geometric and kinodynamic planning with forward propagation*. 19th Sept. 2018. arXiv: 1809.0705v1 [cs.RO].

[45] Canny J. and Reif J.H. 'New lower bound techniques for robot motion planning problems'. In: *IEEE Symposium on Foundations of Computer Science (FoCS)* (1987), pp. 49–60.

[46] Kiril Solovey, Lucas Janson, Edward Schmerling et al. *Revisiting the Asymptotic Optimality of RRT**. 22nd Apr. 2020. arXiv: 1909.09688v2 [cs.RO].

[47] Kiril Solovey, Lucas Janson, Edward Schmerling et al. *Revisiting the Asymptotic Optimality of RRT**. 22nd Apr. 2020. arXiv: 1909.09688v2 [cs.RO].

[48] David Hsu, Jean-Claude Latombe and Hanna Kurniawati. 'On the Probabilistic Foundations of Probabilistic Roadmap Planning'. In: *The International Journal of Robotics Research* 25.7 (2006), pp. 627–643. DOI: 10.1177/0278364906067174. eprint: https://doi.org/10.1177/0278364906067174. URL: https://doi.org/10.1177/0278364906067174.

[49] R. Bohlin and L.E. Kavraki. 'Path planning using lazy PRM'. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 1. 2000, 521–528 vol.1. DOI: 10.1109/ROBOT.2000.844107.

[50] Edsger W Dijkstra. 'A note on two problems in connexion with graphs'. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[51] T. Chan and Wei Zhu. 'Level set based shape prior segmentation'. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. 2005, 1164–1170 vol. 2. DOI: 10.1109/CVPR.2005.212.

[52] Helen Oleynikova, Zachary Taylor, Marius Fehr et al. 'Voxblox: Building 3D Signed Distance Fields for Planning'. In: (Nov. 2016).

[53] Open Robotics. *Rviz*. Version ROS-noetic-Rviz. 25th May 2022. URL: http://wiki.ros.org/rviz/UserGuide.

[54] Stanford Artificial Intelligence Laboratory et al. *Documentation, ROS*. https://www.ros.org. Accessed: 2022-05-25.

[55] Tung Dang, Mihir Kulkarni and Mihir Dharmadhi. *pci mav*. https://github.com/ntnu-arl/pci_mav. 2021.

[56] Fadri Furrer, Michael Burri, Markus Achtelik et al. 'Robot Operating System (ROS): The Complete Reference (Volume 1)'. In: ed. by Anis Koubaa. Cham: Springer International Publishing, 2016. Chap. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_23. URL: http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[57] Berthold Horn. 'Closed-Form Solution of Absolute Orientation Using Unit Quaternions'. In: *Journal of the Optical Society A* 4 (Apr. 1987), pp. 629–642. DOI: 10.1364/JOSAA.4.000629.

[58] Jon Louis Bentley. 'Multidimensional Binary Search Trees Used for Associative Searching'. In: *Commun. ACM* 18.9 (1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[59] ETH Zürich Autonomous Systems Lab. *kdtree*. https://github.com/ethz-asl/nbvplanner/tree/master/kdtree. 2015.

[60] Armin Hornung, Kai M. Wurm, Maren Bennewitz et al. 'OctoMap: an efficient probabilistic 3D mapping framework based on octrees'. In: *Autonomous Robots* 34.3 (2013), pp. 189–206. DOI: 10.1007/s10514-012-9321-0. URL: https://doi.org/10.1007/s10514-012-9321-0.

[61] David Churchill and Michael Buro. 'Incorporating Search Algorithms into RTS Game Agents'. In: 2012.

[62] Weigui Jair Zhou, Budhitama Subagdja, Ah-Hwee Tan et al. 'Hierarchical control of multi-agent reinforcement learning team in real-time strategy (RTS) games'. In: *Expert Systems with Applications* 186 (2021), p. 115707. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2021.115707. URL: https://www.sciencedirect.com/science/article/pii/S0957417421010897.

[63] Chao Yao. *gazebo models worlds collection*. https://github.com/leonhartyao/gazebo_models_worlds_collection. 2020.

Appendix

A Planner function

```
Prm::GraphStatus Prm::planPath(geometry_msgs::Pose& target_pose,
↪  std::vector<geometry_msgs::Pose>& best_path){
  START_TIMER(ttime);
  // tuning parameters
  int loop_count(0);
  int num_vertices_added(0);
  int num_edges_added(0);
  int num_target_neighbours(0);

  std::vector<geometry_msgs::Pose> best_path_temp;

  StateVec target_state;
  convertPoseMsgToState(target_pose, target_state);

  units_[active_id_]->final_target_ = target_state;
  minimap_->setTarget(active_id_, &(units_[active_id_]->final_target_));
  minimap_->setTargetStatus(active_id_, true);


  units_[active_id_]->current_waypoint_ = units_[active_id_]->current_vertex_;


  Eigen::Vector3d dir_vec(units_[active_id_]->current_state_[0] -
↪  target_state[0],
                          units_[active_id_]->current_state_[1] -
↪  target_state[1],
                          units_[active_id_]->current_state_[2] -
↪  target_state[2]);



  double dir_dist = dir_vec.norm();
  double best_dist = 10000000; //inf

  // catch if robot is already at target
  if (abs(dir_vec.norm()) < random_sampling_params_->reached_target_radius) {
    units_[active_id_]->reached_final_target_ = true;
    ROS_INFO("UNIT %d: already at target given", active_id_);
    return Prm::GraphStatus::OK;
  }


  bool stop_sampling = false;
  detectTargetStatus(active_id_);
  if (units_[active_id_]->target_status_ == Prm::StateStatus::CONNECTED){
    ROS_INFO("UNIT %d: Target already connected! Returning path without expanding
↪  graph", active_id_);
    stop_sampling = true;
    num_target_neighbours++;
    units_[active_id_]->reached_final_target_ = true;
    total_already_exists_++;
    //minimap_->setTargetStatus(active_id_, false);
  }
```

```
  std::vector<Vertex*> target_neighbours;

  stat_->init(units_[active_id_]->current_vertex_->state);



  while ((!stop_sampling)&&(loop_count++ < planning_params_.num_loops_max) &&
    (num_vertices_added < planning_num_vertices_max_) &&
    (num_edges_added < planning_num_edges_max_)) {
    StateVec new_state;
    if (!sampleVertex(new_state)) {
      if ((loop_count >= planning_params_.num_loops_cutoff) &&
        (num_vertices_added <= 2)) {
      break;
      }
      continue; // skip invalid sample
    }

    ExpandGraphReport rep;
    expandGraph(roadmap_graph_, new_state, rep);

    if (rep.status == ExpandGraphStatus::kSuccess) {
      num_vertices_added += rep.num_vertices_added;
      num_edges_added += rep.num_edges_added;
      // Check if state is inside target area
      Eigen::Vector3d radius_vec(new_state[0] - target_state[0],
                                 new_state[1] - target_state[1],
                                 new_state[2] - target_state[2]);

      // Check if sampled vertex is close enough to target area
      if (radius_vec.norm() < random_sampling_params_->reached_target_radius) {
        target_neighbours.push_back(rep.vertex_added);
        num_target_neighbours++;
        units_[active_id_]->reached_final_target_ = true;
        units_[active_id_]->current_waypoint_ = target_neighbours[0];
        //
        if (num_target_neighbours >
↪   random_sampling_params_->num_paths_to_target_max){
          // stop samling if we have enough sampled points in target area
          stop_sampling = true;
        }
      }
      if ((num_target_neighbours < 1) && (radius_vec.norm() < dir_dist) &&
↪   (radius_vec.norm() < best_dist)) {
        // if no points in target area is sampled, we go to the point closest to
↪   the target in euclidean distance
        best_dist = radius_vec.norm();
        units_[active_id_]->current_waypoint_ = rep.vertex_added;
      }
    }
    if ((loop_count >= planning_params_.num_loops_cutoff) &&
        (num_vertices_added <= 2)) {
      break;
    }
  }
```

```
  ROS_INFO("Formed a graph with [%d] vertices and [%d] edges with [%d] loops,
↪  ntn[%d]",
          roadmap_graph_->getNumVertices(), roadmap_graph_->getNumEdges(),
↪  loop_count, num_target_neighbours);

  Prm::GraphStatus res = Prm::GraphStatus::NOT_OK;

  if(roadmap_graph_->getNumVertices() < 2){
    ROS_WARN("UNIT %d: Sampler failed, try again");
    return res;
  }
  stat_->build_graph_time = GET_ELAPSED_TIME(ttime);
  if(stat_->build_graph_time > 0.05){
    build_times_.push_back(stat_->build_graph_time);
  }
  total_build_time_ += stat_->build_graph_time;

  if (num_target_neighbours < 1) {
    Vertex* waypoint;
    roadmap_graph_->getNearestVertex(&target_state, &waypoint);
    units_[active_id_]->current_waypoint_ = waypoint;
    ROS_INFO("UNIT %d: Target not yet reached by roadmap, updated waypoint as
↪  best vertex", active_id_);
  }
  START_TIMER(ttime);
  std::vector<int> path_id_list;
  roadmap_graph_rep_.reset();
  roadmap_graph_->findShortestPaths(units_[active_id_]->current_vertex_->id,
↪  roadmap_graph_rep_);
  // Get the shortest path to current waypoint, and collision check the path if
↪  in lazy mode

  if(lazy_mode_){

    bool collision_free_path_found = false;
    while(!collision_free_path_found){
      roadmap_graph_->getShortestPath(units_[active_id_]->current_waypoint_->id,
↪  roadmap_graph_rep_, false, path_id_list);
      for(int i = 0; i < path_id_list.size()-1; i++){
        Vertex* v_start = roadmap_graph_->getVertex(path_id_list[i]);
        Vertex* v_end = roadmap_graph_->getVertex(path_id_list[i+1]);

        if (!checkCollisionBetweenVertices(v_start, v_end)){
          // edge is not collision free
          roadmap_graph_->removeEdge(v_start, v_end);
          ROS_WARN("UNIT %d: Collision found in path, replanning", active_id_);

↪  roadmap_graph_->findShortestPaths(units_[active_id_]->current_vertex_->id,
↪  roadmap_graph_rep_);
          break;
        }
        if (i == path_id_list.size()-1) {
          // We have iterated through the whole path without having a collision
          collision_free_path_found = true;
          res = Prm::GraphStatus::OK;
        }
      }
    }
```

```
    if (!collision_free_path_found) {
      res = Prm::GraphStatus::ERR_NO_FEASIBLE_PATH;
    }
  } else {
    // Get the shortest path to current waypoint
    double traverse_length = 0;
    double traverse_time = 0;
    std::vector<StateVec> best_path_states;
    roadmap_graph_->getShortestPath(units_[active_id_]->current_waypoint_->id,
↪   roadmap_graph_rep_, true, best_path_states);
    Eigen::Vector3d p0(best_path_states[0][0], best_path_states[0][1],
↪   best_path_states[0][2]);
    std::vector<Vertex*> best_path_vertices;
    roadmap_graph_->getShortestPath(units_[active_id_]->current_waypoint_->id,
↪   roadmap_graph_rep_, true,
                                    best_path_vertices);

    const double kLenMin = 0.5;
    std::vector<Eigen::Vector3d> path_vec;
    roadmap_graph_->getShortestPath(units_[active_id_]->current_waypoint_->id,
↪   roadmap_graph_rep_, true,
                                    path_vec);
    double total_len = Trajectory::getPathLength(path_vec);
    if (total_len <= kLenMin) {
      ROS_WARN("UNIT %d: Best path is too short, requeue different target");
      return Prm::GraphStatus::ERR_NO_FEASIBLE_PATH;
    }

    for (int i = 0; i < best_path_states.size(); ++i) {
      Eigen::Vector3d p1(best_path_states[i][0], best_path_states[i][1],
↪   best_path_states[i][2]);
      Eigen::Vector3d dir_vec = p1 - p0;
      tf::Quaternion quat;
      quat.setEuler(0.0, 0.0, best_path_states[i][3]);
      tf::Vector3 origin(best_path_states[i][0], best_path_states[i][1],
↪   best_path_states[i][2]);
      tf::Pose poseTF(quat, origin);
      geometry_msgs::Pose pose;
      tf::poseTFToMsg(poseTF, pose);
      best_path_temp.push_back(pose);
      double seg_length = (p1 - p0).norm();
      traverse_length += seg_length;
      if ((traverse_length > planning_params_.traverse_length_max)) {
        units_[active_id_]->reached_final_target_ = false;
        break;
      }
      p0 = p1;
    }

    roadmap_graph_->getShortestPath(units_[active_id_]->current_waypoint_->id,
↪   roadmap_graph_rep_, true, path_id_list);

    res = Prm::GraphStatus::OK;
  }
  stat_->shortest_path_time = GET_ELAPSED_TIME(ttime);
  total_path_extraction_ += stat_->shortest_path_time;
  ros::Time cctime;
  START_TIMER(cctime);
```

```
  // collision checking
  for (auto& u: units_){
    if ((u->id_ != active_id_)){

      double active_time = ros::Time::now().toSec();
      double active_time_segment = 0;
      // Iterate through segments in the current units path
      for(int v_active_id = 0; v_active_id < path_id_list.size() - 1;
↪  v_active_id++){
        // get first vertex in segment
        Vertex* v0 = roadmap_graph_->getVertex(path_id_list[v_active_id]);
        // get second vertex in segment
        Vertex* v1 = roadmap_graph_->getVertex(path_id_list[v_active_id+1]);
        // convert to vector
        Eigen::Vector3d p0(v0->state.x(), v0->state.y(), v0->state.z());
        // convert to vector
        Eigen::Vector3d p1(v1->state.x(), v1->state.y(), v1->state.z());
        // calculate segment vector
        Eigen::Vector3d active_segment = p1 - p0;
        active_time_segment = active_segment.norm()/planning_params_.v_max;
        // timestamp start of segment
        double t0 = active_time;
        active_time += active_time_segment;
        // timestamp end of segment
        double t1 = active_time;
        //----------------------Make segment-cuboid for
↪  collisionchecking-----------------
        Eigen::Vector3d active_center = (p1+p0)/2;
        Eigen::Vector3d active_half_dim = robot_box_size_/2;
        Eigen::Vector3d active_min_point = active_center - active_half_dim;
        Eigen::Vector3d active_max_point = active_center + active_half_dim;
        Eigen::AlignedBox3d active_cuboid(active_min_point, active_max_point);


        if((u->currently_moving_)&& (!u->current_path_id_list_.empty())){
        // Variable to keep tracked of checked time
        double checked_time = u->moving_time_start_.toSec();
        double checked_time_segment = 0;
        // iterate through segments in all other unit paths
        for(int v_check_id = 0; v_check_id < u->current_path_id_list_.size() - 1;
↪  v_check_id++){
          // get first vertex in segment
          Vertex* vc0 =
↪  roadmap_graph_->getVertex(u->current_path_id_list_[v_check_id]);
          // get second vertex in segment
          Vertex* vc1 =
↪  roadmap_graph_->getVertex(u->current_path_id_list_[v_check_id+1]);
          // convert to vector
          Eigen::Vector3d pc0(vc0->state.x(), vc0->state.y(), vc0->state.z());
          // convert to vector
          Eigen::Vector3d pc1(vc1->state.x(), vc1->state.y(), vc1->state.z());
          // calculate segment vector
          Eigen::Vector3d check_segment = pc1 - pc0;
          // get the duration the other unit has been moving
          //double time_diff = active_time - u_start_time;
          // predict where the unit would be if it only moves along this segment
          //Eigen::Vector3d predicted_segment = pc0 +
↪  (time_diff-checked_time)*planning_params_.v_max*check_segment.normalized();
```

```
        // Add the time the robot has taken to complete the segment
        checked_time_segment = check_segment.norm()/planning_params_.v_max;
        // timestamp start of segment
        double tc0 = checked_time;
        checked_time += checked_time_segment;
        // timestamp end of segment
        double tc1 = checked_time;
        //---------------------Make segment-cuboid for
↪  collisionchecking----------------
        Eigen::Vector3d check_center = (pc1+pc0)/2;
        Eigen::Vector3d check_half_dim = robot_box_size_/2;
        Eigen::Vector3d check_min_point = check_center - check_half_dim;
        Eigen::Vector3d check_max_point = check_center + check_half_dim;
        Eigen::AlignedBox3d check_cuboid(check_min_point, check_max_point);
        if(!doCuboidsIntersect(active_cuboid, check_cuboid)){
          //The segments with robot-size does not intersect, hence collision
↪  checking not required
          continue;
        }
        // If we go here, it means that a segment in the best path found by the
↪  planner potentially
        // crosses with a path that is currently being executed by another
↪  unit
        // and that the unit we may collide with has not executed the scary
↪  part yet
        // We finally checks if the timing seems scary to see if the active
↪  robot should wait or not
        if (!((t1 <= tc0)||(t0 >= tc1))){
          // The timing ranges are scary
          ROS_WARN("Planner detected possible collision, requeue at later
↪  time");
          return Prm::GraphStatus::ERR_NO_FEASIBLE_PATH;
        }

      }
    } else if (!(u->currently_moving_)){
    //check if path passes through idle robot
    //---------------------Make segment-cuboid for
↪  collisionchecking----------------
        Eigen::Vector3d check_center(u->current_state_.x(),
↪  u->current_state_.y(), u->current_state_.z());

        Eigen::Vector3d check_half_dim = robot_box_size_/2;
        Eigen::Vector3d check_min_point = check_center - check_half_dim;
        Eigen::Vector3d check_max_point = check_center + check_half_dim;
        Eigen::AlignedBox3d check_cuboid(check_min_point, check_max_point);
    //---------------------Make vertex-cuboid for
↪  collisionchecking----------------
      Eigen::Vector3d vertex_center = p1;
      Eigen::Vector3d vertex_half_dim = robot_box_size_/2;
      Eigen::Vector3d vertex_min_point = vertex_center - vertex_half_dim;
      Eigen::Vector3d vertex_max_point = vertex_center + vertex_half_dim;
      Eigen::AlignedBox3d vertex_cuboid(vertex_min_point, vertex_max_point);

        if(doCuboidsIntersect(active_cuboid, check_cuboid) ||
↪  doCuboidsIntersect(vertex_cuboid, check_cuboid)){
          //The path passes through an idle robot
          ROS_INFO("idle collision");
```

```
            return Prm::GraphStatus::ERR_NO_FEASIBLE_PATH;
          }
        }
      }
    }
  }

ROS_INFO("UNIT %d: No collisions detected", active_id_);

  stat_->collision_check_time = GET_ELAPSED_TIME(cctime);
  units_[active_id_]->current_path_id_list_ = path_id_list;
  if (!(best_path_temp.empty())){
    num_queries_++;
    double yawhalf = units_[active_id_]->current_state_[3] * 0.5;
    best_path_temp[0].orientation.x = 0.0;
    best_path_temp[0].orientation.y = 0.0;
    best_path_temp[0].orientation.z = sin(yawhalf);
    best_path_temp[0].orientation.w = cos(yawhalf);
  }
  for (int i = 0; i < (best_path_temp.size() - 1); ++i) {
      Eigen::Vector3d vec(best_path_temp[i + 1].position.x -
↪   best_path_temp[i].position.x,
                          best_path_temp[i + 1].position.y -
↪   best_path_temp[i].position.y,
                          best_path_temp[i + 1].position.z -
↪   best_path_temp[i].position.z);
      double yaw = std::atan2(vec[1], vec[0]);
      tf::Quaternion quat;
      quat.setEuler(0.0, 0.0, yaw);
      best_path_temp[i + 1].orientation.x = quat.x();
      best_path_temp[i + 1].orientation.y = quat.y();
      best_path_temp[i + 1].orientation.z = quat.z();
      best_path_temp[i + 1].orientation.w = quat.w();
  }

  best_path = best_path_temp;
  if (!(best_path.empty())){
    double total_len = Trajectory::getPathLength(best_path);
    Eigen::Vector3d opt_vec(units_[active_id_]->current_state_[0] -
↪   best_path[best_path.size()-1].position.x,
                            units_[active_id_]->current_state_[1] -
↪   best_path[best_path.size()-1].position.y,
                            units_[active_id_]->current_state_[2] -
↪   best_path[best_path.size()-1].position.z);
    double opt_dist = opt_vec.norm();
    double optimality = total_len/opt_dist;
    total_path_optimality_ += optimality;
    path_optimalities_.push_back(optimality);

  }




  visualization_[active_id_]->visualizeSampler(random_sampler_);
  random_sampler_.reset();
```

```
  visualization_[active_id_]->visualizeBestPaths(roadmap_graph_,
↪   roadmap_graph_rep_, 0, units_[active_id_]->current_waypoint_->id);
  if (roadmap_graph_->getNumVertices() > 1){
    visualization_[active_id_]->visualizeGraph(roadmap_graph_);
  } else {
    visualization_[active_id_]->visualizeFailedEdges(stat_);
    ROS_INFO("Number of failed samples: [%d] vertices and [%d] edges",
    stat_->num_vertices_fail, stat_->num_edges_fail);
    res = Prm::GraphStatus::ERR_KDTREE;
  }
  return res;
}
```

B Graph expansion function

```
void Prm::expandGraph(std::shared_ptr<GraphManager> graph,
                      StateVec& new_state, ExpandGraphReport& rep){
  // Find nearest neighbour
  Vertex* nearest_vertex = NULL;

  if (!roadmap_graph_->getNearestVertex(&new_state, &nearest_vertex)) {
    rep.status = ExpandGraphStatus::kErrorKdTree;
    return;
  }
  if (nearest_vertex == NULL) {
    rep.status = ExpandGraphStatus::kErrorKdTree;
    return;
  }
  // Check for collision of new connection plus some overshoot distance.
  Eigen::Vector3d origin(nearest_vertex->state[0], nearest_vertex->state[1],
                         nearest_vertex->state[2]);
  Eigen::Vector3d direction(new_state[0] - origin[0], new_state[1] - origin[1],
                            new_state[2] - origin[2]);
  double direction_norm = direction.norm();

  if (direction_norm > planning_params_.edge_length_max) {
    direction = planning_params_.edge_length_max * direction.normalized();
  } else if ((direction_norm <= planning_params_.edge_length_min)) {
    // Should not add short edge.
    rep.status = ExpandGraphStatus::kErrorShortEdge;
    return;
  }
  // Recalculate the distance.
  direction_norm = direction.norm();
  new_state[0] = origin[0] + direction[0];
  new_state[1] = origin[1] + direction[1];
  new_state[2] = origin[2] + direction[2];
  // Since we are buiding graph,
  // Consider to check the overshoot for both 2 directions except root node.
  Eigen::Vector3d overshoot_vec =
      planning_params_.edge_overshoot * direction.normalized();
  Eigen::Vector3d start_pos = origin + robot_params_.center_offset;
  if (nearest_vertex->id != 0) start_pos = start_pos - overshoot_vec;
  Eigen::Vector3d end_pos =
      origin + robot_params_.center_offset + direction + overshoot_vec;
  // Check collision if lazy mode is not activated
```

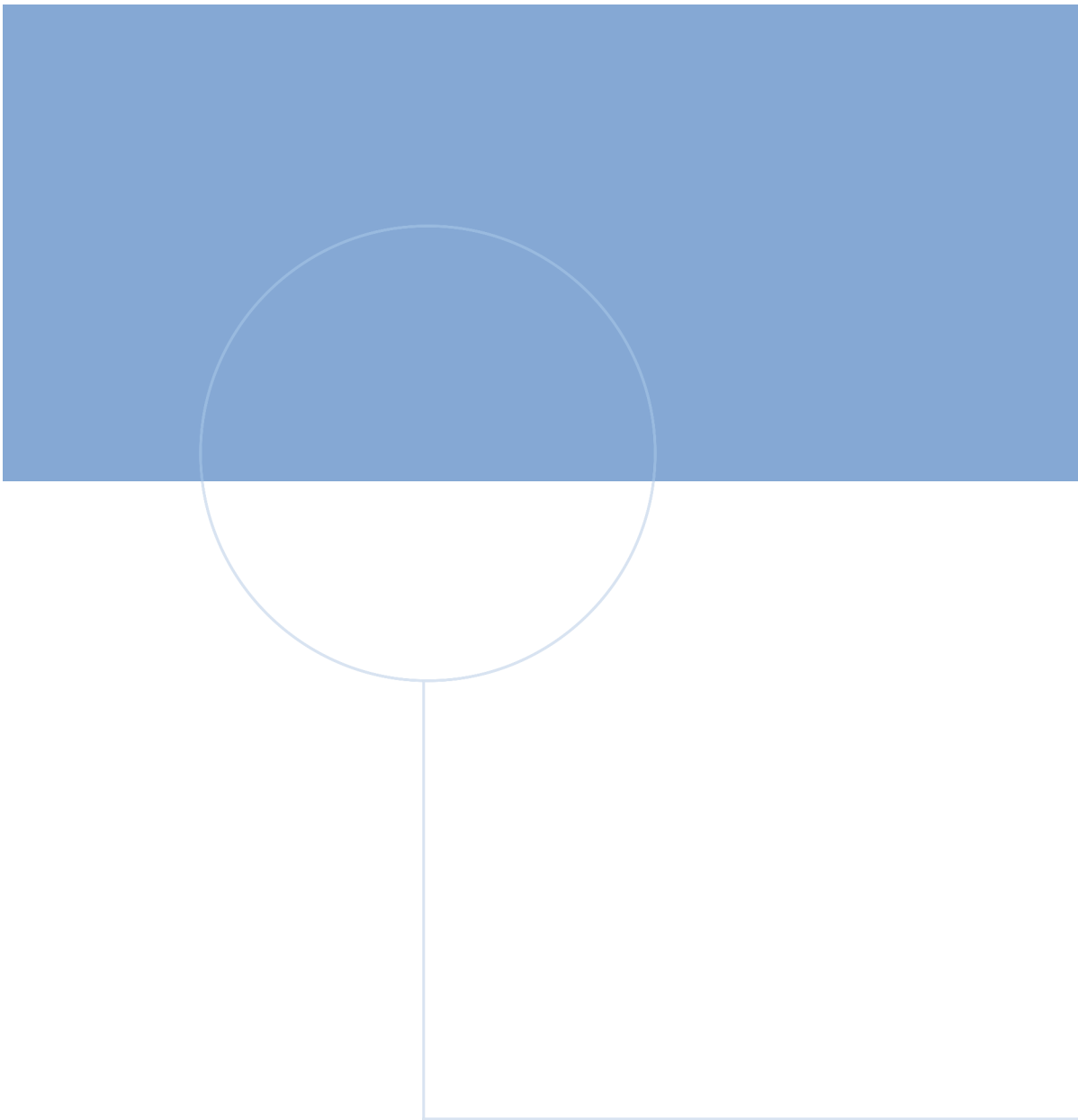
```
  if ( MapManager::VoxelStatus::kFree ==
       map_manager_->getPathStatus(start_pos, end_pos, robot_box_size_, false)
↪  || lazy_mode_) {
    Vertex* new_vertex =
        new Vertex(roadmap_graph_->generateVertexID(), new_state);

    // Form a tree as the first step.
    new_vertex->parent = nearest_vertex;
    new_vertex->distance = nearest_vertex->distance + direction_norm;
    nearest_vertex->children.push_back(new_vertex);
    roadmap_graph_->addVertex(new_vertex);
    ++rep.num_vertices_added;
    rep.vertex_added = new_vertex;
    roadmap_graph_->addEdge(new_vertex, nearest_vertex, direction_norm);
    ++rep.num_edges_added;

    // add more edges to create graph
    std::vector<Vertex*> nearest_vertices;
    if (!roadmap_graph_->getNearestVertices(
            &new_state, planning_params_.nearest_range, &nearest_vertices)) {
      rep.status = ExpandGraphStatus::kErrorKdTree;
      return;
    }
    origin << new_vertex->state[0],new_vertex->state[1],new_vertex->state[2];
    for (int i = 0; i < nearest_vertices.size(); ++i) {
        //ROS_WARN("noe: %d"
↪   ,roadmap_graph_->getNumOutgoingEdges(nearest_vertices[i]->id));
        if (roadmap_graph_->getNumOutgoingEdges(nearest_vertices[i]->id) >=
↪   planning_params_.max_num_outgoing){
          // Constraint the amount of outgoing edges per vertex
          continue;
        }
        direction << nearest_vertices[i]->state[0] - origin[0],
            nearest_vertices[i]->state[1] - origin[1],
            nearest_vertices[i]->state[2] - origin[2];
        double d_norm = direction.norm();

        if ((d_norm > planning_params_.nearest_range_min) &&
            (d_norm < planning_params_.nearest_range_max)) {
          Eigen::Vector3d p_overshoot =
              direction / d_norm * planning_params_.edge_overshoot;
          Eigen::Vector3d p_start =
              origin + robot_params_.center_offset - p_overshoot;
          Eigen::Vector3d p_end =
              origin + robot_params_.center_offset + direction;
          if (nearest_vertices[i]->id != 0) p_end = p_end + p_overshoot;
          // Check collision if lazy mode is not activated
          if (MapManager::VoxelStatus::kFree ==
                map_manager_->getPathStatus(p_start, p_end, robot_box_size_,
↪   true) || lazy_mode_) {
              roadmap_graph_->addEdge(new_vertex, nearest_vertices[i], d_norm);
              ++rep.num_edges_added;
          }
        }
    }

  } else {
```

```
    stat_->num_edges_fail++;
    if (stat_->num_edges_fail < 500) {
      std::vector<double> vtmp = {start_pos[0], start_pos[1], start_pos[2],
                                  end_pos[0],   end_pos[1],   end_pos[2]};
      stat_->edges_fail.push_back(vtmp);
    }
    rep.status = ExpandGraphStatus::kErrorCollisionEdge;
    return;
  }
  rep.status = ExpandGraphStatus::kSuccess;
}
```

C ROS libraries

This appendix presents the ROS packages used for this project that are not mentioned in the report. A link to the respective GitHub repositories are supplied. All packages listed with the exeption of `catkin_simple` are created by the Autonomous Systems Lab at ETH Zürich.

- `catkin_simple v0.1.1` https://github.com/catkin/catkin_simple
- `eigen_catkin v3.2.12` https://github.com/ethz-asl/eigen_catkin
- `eigen_checks v2.8.3` https://github.com/ethz-asl/eigen_checks
- `gflags_catkin v2.2.1` - https://github.com/ethz-asl/gflags_catkin
- `glog_catkin v0.3.5` - https://github.com/ethz-asl/glog_catkin
- `mav_comm` - https://github.com/ethz-asl/mav_comm
- `minkindr v0.0.1` - https://github.com/ethz-asl/minkindr
- `minkindr_conversions v0.0.0` - https://github.com/ethz-asl/minkindr_ros/tree/master/minkindr_conversions
- `yaml_cpp_catkin v0.5.90` - https://github.com/ethz-asl/yaml_cpp_catkin



NTNU

Norwegian University of
Science and Technology