3D Coverage Path Planning for Efficient Construction Progress Monitoring

3D Abdeckungspfadplanung zur effizienten Baufortschrittsüberwachung
Master thesis by Katrin Becker
Date of submission: 20.05.2022

1. Review: Prof. Dr. Oskar von Stryk
2. Review: M.Sc. Martin Sven Oehler
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Fachgebiet Simulation, Systemoptimierung und Robotik

3D Coverage Path Planning for Efficient Construction Progress Monitoring
3D Abdeckungspfadplanung zur effizienten Baufortschrittsüberwachung

Master thesis by Katrin Becker

1. Review: Prof. Dr. Oskar von Stryk
2. Review: M.Sc. Martin Sven Oehler

Date of submission: 20.05.2022

Darmstadt

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Katrin Becker, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 20.05.2022

K. Becker

Abstract

Autonomous robots have many applications, like in disaster scenarios or for inspection and monitoring. They can take on tasks that are very exhausting, tiring, repetitive or even dangerous for humans.
One of these applications, which is handled in this thesis, is the monitoring of construction site progress. This task is often done by humans using a camera or terrestrial laser scanner. As a result, this data is often incomplete and hardly comparable. Additionally, it is a very time consuming task especially if it is done periodically.
In this thesis, a novel 3D coverage path planner for efficient construction progress monitoring is proposed. It is based on next best view planners but instead of computing the next viewpoint online, the path is computed before the execution. An important difference to existing work is, that a 3D model of the building is given and based on this model the path should be planned. Additionally, a subset of this model can be given as target model, which forms the set of target points. This model contains the parts of the complete model, that should be covered when executing the path, for example when only special rooms or walls should be monitored or the resulting model should primarily cover pipes or cables.
For planning the path, first, candidate viewpoints are generated over the traversable space. Next, they are rated by the amount of visible target points. A set of candidates, from which the whole target model can be covered, is selected as viewpoints. These viewpoints, converted into waypoints, form the 3D coverage path. In order to be efficient, this set needs to be as small as possible and the waypoints need to be in a suitable order. Finding the best waypoint order so that the path is as short as possible and does not contain any path redundancies, is a Traveling Salesman Problem (TSP) type problem. In the thesis several approaches for the viewpoint selection and solving the TSP are used and compared. When executing the path, data is recorded at each viewpoint, e.g. as 360 degree images or point clouds. This data forms the resulting model.
The developed approach is evaluated on differently sized models including multiple levels and complex structures as well as on different target models. It is shown that the planning worked well on all these models and the high coverage of the environment after the execution proves the quality of the planned path.

Zusammenfassung

Autonome Roboter haben zahlreiche Anwendungsbereiche wie in Katastrophenszenarien oder zur Inspektion und Überwachung. Sie sind besonders geeignet zum Einsatz bei anstrengenden, ermüdenden, sich wiederholenden oder sogar für Menschen gefährlichen Aufgaben.
Eine dieser Aufgaben, die in dieser Arbeit betrachtet wird, ist das Überwachen des Fortschritts auf Baustellen. Häufig wird diese Aufgabe von einem Menschen mit einer Kamera oder einem terrestrischen Laser Scanner übernommen. Das führt allerdings dazu, dass die erhaltenen Daten oft unvollständig und schwer vergleichbar sind. Außerdem ist der Zeitaufwand sehr groß, gerade wenn regelmäßig Daten aufgenommen werden sollen.
In dieser Arbeit wird ein neuartiges Verfahren für die 3D Abdeckungspfadplanung zur effizienten Baufortschrittsüberwachung vorgeschlagen. Es basiert auf Next Best View Berechnungen aber statt den nächsten Blickpunkt online zu berechnen, wird der Pfad vor der Ausführung berechnet. Ein wichtiger Unterschied zu bestehenden Arbeiten ist, dass ein 3D Modell gegeben ist und der Pfad basierend auf diesem Modell berechnet wird. Zusätzlich kann eine Teilmenge dieses Modells als Zielmodell vorgegeben werden, welche die Menge von Zielpunkten bildet. Dieses Modell beinhaltet die Teile des kompletten Modells, die abgedeckt werden sollen wenn der Pfad abgefahren wird, beispielsweise wenn nur spezielle Räume oder Wände überwacht werden sollen oder das resultierende Modell hauptsächlich Rohre und Kabel abdecken soll.
Um einen solchen Pfad zu planen werden zuerst Kandidatenposen für Blickpunkte über dem befahrbarem Teil des Modells erzeugt. Diese werden anschließend anhand der Anzahl der sichtbaren Zielpunkte bewertet. Eine Menge von Kandidaten, von denen aus das vollständige Modell gesehen werden kann, wird als Blickpunkte ausgewählt. Diese Blickpunkte, umgewandelt in Wegpunkte, bilden den 3D Abdeckungspfad. Um möglichst effizient zu sein, muss diese Menge so klein wie möglich gewählt werden und die Wegpunkte in eine geeignete Reihenfolge gebracht werden. Die beste Reihenfolge zu finden, so dass die Pfadlänge minimal ist und keine Redundanzen beinhaltet, ist ein Problem des Typs Problem eines Handlungsreisenden (Traveling Salesman Problem (TSP)). In dieser Arbeit werden verschiedene Ansätze für die Auswahl der Blickpunkte und zur Lösung des TSP verglichen. Während der Pfad abgefahren wird, werden Daten an jedem Blickpunkt aufgenommen, beispielsweise als 360 Grad Bilder oder Punktwolken. Diese Daten bilden das resultierende Modell.
Der entwickelte Ansatz wurde auf verschieden großen Modellen mit mehreren Ebenen und komplexen Strukturen sowie unterschiedlichen Zielmodellen ausgewertet. Es wird gezeigt, dass der Planer auf allen Modellen funktioniert und nach der Durchführung eine hohe Abdeckung der Umgebung gegeben ist, was die gute Qualität des geplanten Pfads zeigt.

Contents

1. Introduction

1.1. Motivation

On construction sites, progress needs to be monitored continuously in order to ensure, that the current state corresponds with the desired state. It is also important to know the exact location of objects such as pipes and cables for further construction. So it is desired to obtain a model, e.g. a 3D model or a set of images, that contains data of the complete construction site.

Data for such a model can all be obtained manually by humans taking photos and scans of the environment. The downside is that this is a very time-consuming task which is prone to errors and can result in incomplete data.

In order to improve data quality, it would be helpful to have a plan showing the exact locations of points from which to take the required data. This would result in a more complete model and in more consistent and comparable data. This last point is especially important if data is recorded regularly. This leads to the first task, which is computing such a plan containing all the required viewpoints. For determining the best viewpoints, a model of the desired state of the construction site is required. This type of model can be obtained from the process of building information modeling (BIM).

But even with such a plan, a human would still need to place a camera or other sensors as accurately as possible at each of the points which can be hard and will mostly take a lot of time. Hence, this task can be automated using ground robots. Since the main focus of this task is construction sites of buildings, ground robots are best suited as they can maneuver in the inside of buildings and can carry several sensors which are required for obtaining the desired data. Such an environment is not well suited for drones, for example, due to narrow hallways and many potential obstacles, such as cables or covering sheets hanging from the ceiling or wall. Another point speaking for the use of robots is accuracy. Humans tend to make errors when working on long repetitive tasks. Additionally, if the robot completes the task autonomously faster than a human with a scan plan and a sensor can, no human resources are blocked and the time consumption can be lowered drastically.

For autonomous monitoring, it is important that the construction site is monitored as completely as possible in the shortest possible time. Therefore, it is required to compute a time-optimal path through the construction site from which the complete environment can be covered (Figure 1.1).

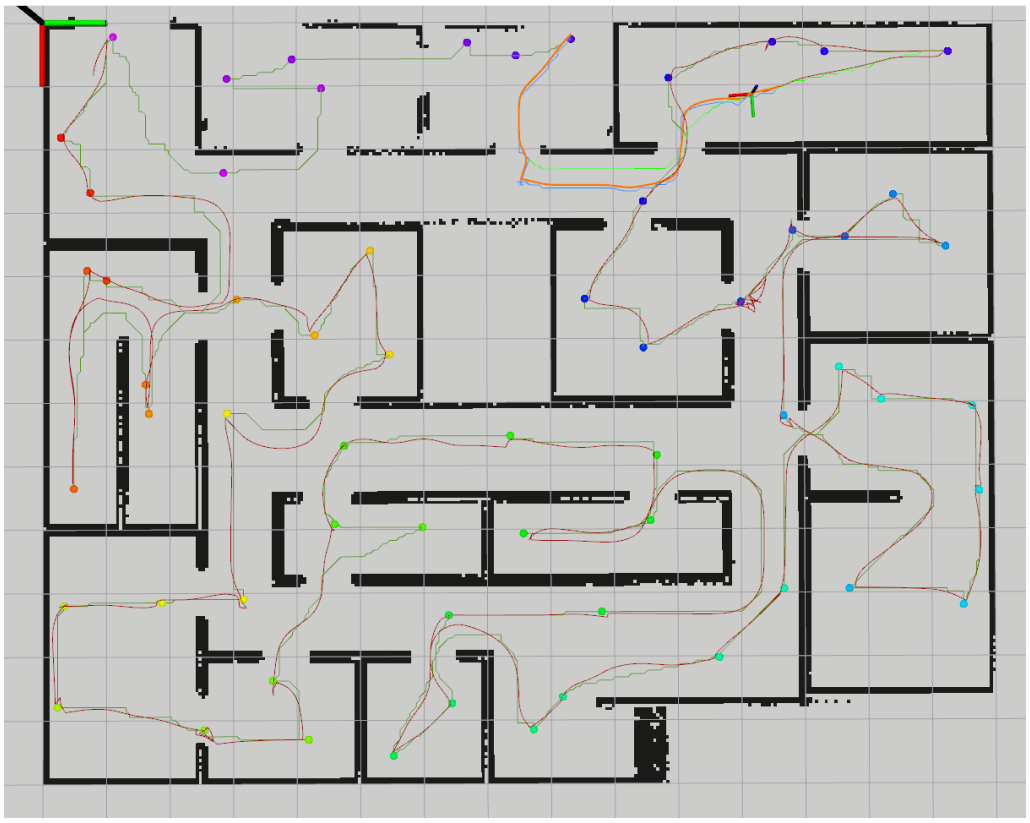


Figure 1.1.: Planned viewpoints (points) and path (dark green) during the execution. The already driven path (dark red) and the current robot position (as axes, in the upper right) are shown as well as a 2D map which was recorded during the execution.

1.2. Overview

This thesis is structured as follows: Section 2 explains foundations and general principles used in this thesis. Then the related work follows in Section 3. Here the research about existing approaches is presented. In Section 4 the methods and proposed algorithms are explained. The implementation details of the algorithms in this work are described in Section 5. This is followed by the evaluation of the implemented approaches displayed in Section 6. Finally, the developed concepts and the results are summarized in the Conclusion in Section 7. Furthermore, an outlook on how the work can be further improved in the future and which follow-up work results from it is given here.

2. Foundations

In the following section, several basic concepts used in this work are explained. First, some mathematical foundations of probability distributions, spherical point set generation and two problems from the field of combinatorial optimization are described. Afterwards the transformation notation is introduced and finally different model representations are presented.

2.1. Probability Distributions

In this thesis, two probability distributions have been used to retrieve random values for several probabilistic decisions.

2.1.1. Uniform distribution

A uniform distribution gets two boundary parameters, a and b with $a \leq b$. The probability density function f of a value x is given by

$$f(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & x < a \text{ or } x > b \end{cases} \tag{2.1}$$

So each value $x \in [a, b]$ is equally probable.

2.1.2. Exponential distribution

In an exponential distribution the probability density function f of a value x is given by

$$f(x, \lambda) = \begin{cases} \lambda e^{(-\lambda x)} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{2.2}$$

where λ is called the *rate parameter*. The mean of an exponential distribution is $1/\lambda$.

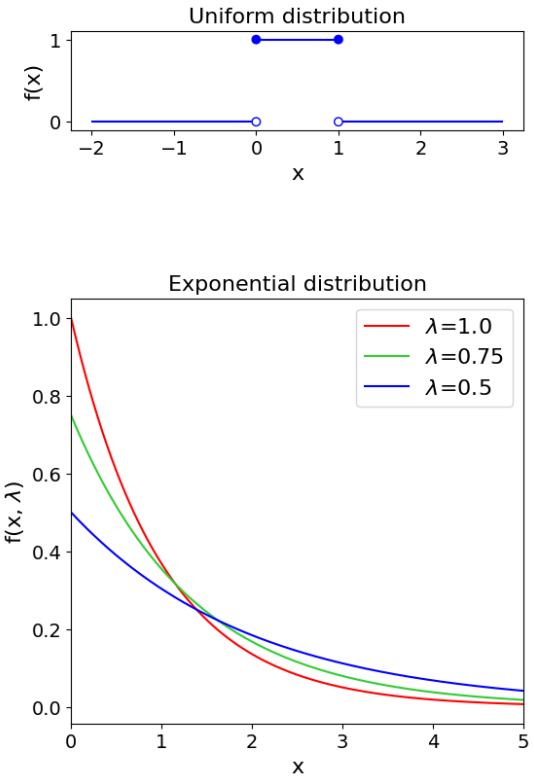
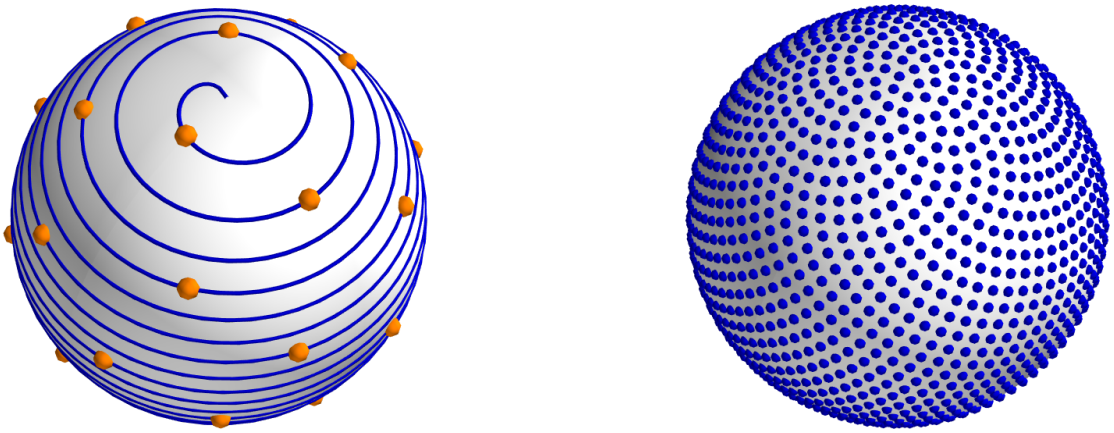


Figure 2.1.: Probability distributions

2.2. Spherical Fibonacci Point Sets



(a) Construction of a spherical Fibonacci Point Set with 32 points

(b) Resulting point set with 2000 points

Figure 2.2.: Spherical Fibonacci Point Sets

Spherical Fibonacci Point Sets [1, 2], also called Fibonacci lattice or Fibonacci Sphere, are points that are evenly distributed on a sphere (Figure 2.2b). In order to construct the point sets, a given number of points is generated along a tightly wound generative spiral, with each point placed in the largest gap between previous points, as shown in Figure 2.2a.

For the construction, the golden ratio Φ is used, which is the positive solution of the following equation:

$$\Phi^{-1} = \Phi - 1 \tag{2.3}$$

$$\Leftrightarrow \quad \Phi = \frac{1+\sqrt{5}}{2} \tag{2.4}$$

The points are constructed in spherical coordinates and then converted into the Cartesian coordinate system using:

$$\mathbf{P}(\phi,\theta) = \begin{pmatrix} \cos(\phi)\sin(\theta) \\ \sin(\phi)\sin(\theta) \\ \cos(\theta) \end{pmatrix} \tag{2.5}$$

The construction rules for the i-th point ($i \in \{0,...,n-1\}$) of the point set $\mathbf{SF}^n$ with n points are:

$$\mathbf{SF}_i^n = \mathbf{P}(\phi_i,\ \cos^{-1}(z_i)) \tag{2.6}$$

with:

$$\phi_i = 2\pi * \left[\frac{i}{\Phi}\right] \tag{2.7}$$

$$z_i = 1 - \frac{2i+1}{n} \tag{2.8}$$

where $[x]$ is the fractional part of x:

$$[x] = x - \lfloor x \rfloor \tag{2.9}$$

2.3. Set cover problem

Given a set of elements X and a family $\mathcal{F}$ of subsets of X, the set cover problem describes the problem of finding a minimal subset $\mathcal{C} \subseteq \mathcal{F}$, whose members cover all elements of X (cf. [3], chap. 35.3). An example can be seen in Figure 2.3, the 16 points are the elements of X and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5\}$. An optimal solution for the set cover problem is $\mathcal{C} = \{S_2, S_3, S_5\}$.

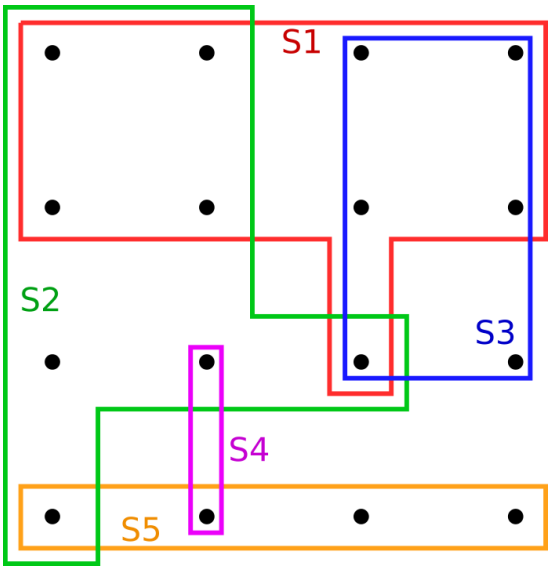


Figure 2.3.: Set cover problem

The set cover problem is NP-complete (cf. [3], chap. 35.3). This means, that there is no algorithm, that can solve this problem in polynomial time. But there exist heuristic solvers for the problem, that can solve it approximately in polynomial time with different guarantees about their solution.

One example for such a heuristic solver is the greedy algorithm (cf. Section 4.2.4). For a set cover problem with $|X| = n$, its solution is maximum $\rho(n)$ times worse than the optimal solution, with

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \tag{2.10}$$

with $H(d)$ is the d-th harmonic number:

$$H(d) = \sum_{i=1}^{d} \frac{1}{i} \tag{2.11}$$

(cf. [3], Theorem 35.4). So for the example shown in Figure 2.3, a solution gained with the greedy algorithm is a maximum of $\rho(16) = H(9) \approx 2.83$ times worse than the optimal solution. The actual solution here is $\mathcal{C} = \{S_1, S_5, S_2, S_3\}$.

2.4. Travelling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) describes the problem of finding the shortest possible route for visiting each of n vertices exactly once and returning to the start vertex, given a list of vertices and the distances or costs between each pair of vertices. The list of vertices and the costs between them are often represented as a weighted graph, as in Figure 2.4.

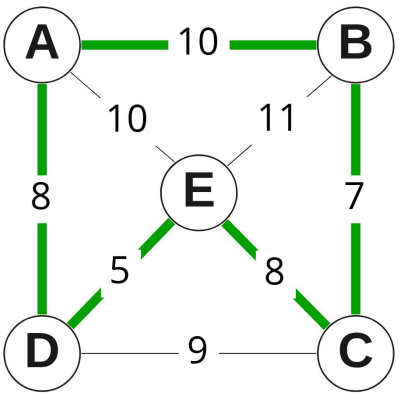


Figure 2.4.: Graph with the solution for the TSP

If the costs are the same in both directions, the TSP is called *symmetric* and the graph is undirected. Additionally, the TSP is called *metric* if the distances or costs between the vertices satisfy the triangle inequality.

The TSP is NP-complete (cf. [3], chap. 34.5.4), as explained above in Section 2.3. But there also exist several heuristic solvers for the TSP, that can find an approximate solution in polynomial time with different guarantees about their solution.

An example for a heuristic solver for symmetric and metric TSPs uses a minimum spanning tree (MST) (cf. [3], chap. 35.2.1 and Section 4.2.5). Here, the worst route returned by this approximation algorithm is maximum two times longer than the optimal route (cf. [3], Theorem 35.2).

2.5. Transformation

Each part of the robot and each object in the environment has an assigned coordinate system at whose origin the object is located. Likewise, a world coordinate system is defined. The coordinate systems are also called frames, e.g. world frame, the robot's base link frame or a sensor frame. The relations between these frames are defined by homogeneous transformations. A transformation from source frame s to target frame t is defined by a rotation matrix ${}^{t}R_{s} \in \mathbb{R}^{3x3}$ and a translation vector ${}^{t}r_{s} \in \mathbb{R}^{3x1}$. They can be written together as the 4x4 matrix ${}^{t}T_{s} \in \mathbb{R}^{4x4}$:

$$ {}^{t}T_{s} = \begin{pmatrix} & {}^{t}R_{s} & & {}^{t}r_{s} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.12} $$

Transformations can also be chained by using:

$$ {}^{t2}T_{s} = {}^{t2}T_{t} * {}^{t}T_{s} \tag{2.13} $$

If a pose P is represented in frame s, it can be written as transformation matrix ${}^{P}T_{s}$, as each pose in a frame s can also be seen as coordinate system relative to frame s where the orientation describes the rotation matrix ${}^{P}R_{s}$ and the position the translation ${}^{P}r_{s}$. The transformation to another frame then works as shown in equation 2.13.

2.6. Model representations

A model of an environment can be represented in different ways. Each type of representation has different advantages and disadvantages, which often lead to specific use cases. In this work, various representations of the input model are needed. While the robot follows the path to record the data, it also creates several representations of its environment.

2.6.1. Mesh

A mesh (Figure 2.5a), or more precisely a polygon mesh, is a model representation that contains vertices, edges and faces. Typically, the surfaces of objects are represented here, the volumes are only implicitly defined.

A mesh can be manifold. In this work, 2-manifold meshes are used, since they are described by 2-dimensional polygons. A 2-manifold mesh fulfills multiple properties. The most important one in this case is its continuity, i.e. it wraps the object or volume completely without any beginning or end. This is the case if it is a surface of a volume. Additionally it holds that an edge needs to be connected to exactly 2 faces which means that inner surfaces and boundary edges are not allowed. Further properties and conditions for manifold meshes exist, but will not be described here, as they are not relevant for this work.

2.6.2. Point cloud

A point cloud is an unordered set of points. Each point has a position and can have different properties like intensity or color. The points are unordered and not clustered, they do not belong to any kind of object representations and there are no continuous surfaces. This makes them unsuitable for several tasks, e.g. object detection or path planning. But they can often be used to create other environment representations, e.g. an occupancy map.

2.6.3. Occupancy grid map

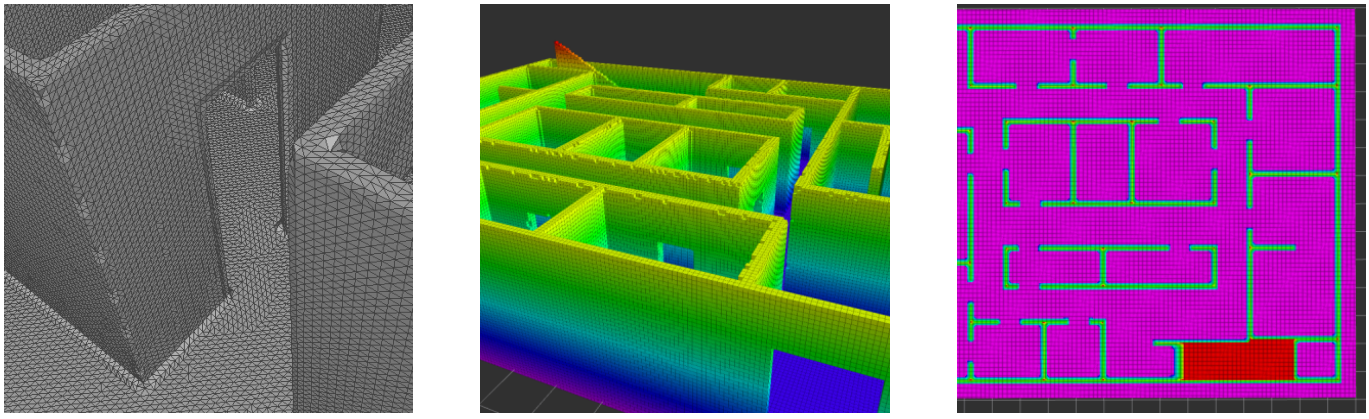
An occupancy grid map is a grid which contains the probabilities that each grid cell contains an obstacle (cf. [4]). Often a threshold is used for these probabilities to decide if a cell is seen as occupied or free.

Occupancy grid maps occur as both 2-dimensional and 3-dimensional maps (Figure 2.5b).

2.6.4. Signed Distance Field (SDF)

Signed distance fields (SDFs) (Figure 2.5c) are also a grid representation, but here each cell contains a signed value that describes the distance to the closest surface. If the value is positive, the cell is in the free space, if it is negative the cell is part of an object. Therefore, the surfaces are located at the zero-crossing. The distance is calculated starting from the center of each cell or voxel, what cells in 3-dimensional grids are often called. On SDFs the gradients can also be computed, which always point away from the object according to their signed distance function.

Two subtypes of SDFs are truncated signed distance fields (TSDFs) and Euclidean signed distance fields (ESDFs) (cf. [5]), which differ mainly in the distance value computation. A TSDF uses the projective distance to the surface along the sensor ray. Furthermore, a TSDF is truncated to contain only values near surfaces, with any distance value greater than the truncation distance set to the truncation distance. An ESDF on the other hand uses the Euclidean distance to the nearest occupied voxel and does not perform any truncation.



(a) Mesh (b) 3D occupancy grid map (c) 2D slice of a 3D SDF

Figure 2.5.: Model representations

3. Related Work

Coverage planning combines aspects from various research on different tasks. The first type of task is the object scanning, where a complete 3D model should be created of a previously unknown object. The second one is the exploration task, where an unknown environment should be explored completely. The last type focuses more on buildings, as here buildings are to be scanned completely, but often not automated with robots, but with e.g. terrestrial laser scanners.

All the above mentioned tasks are often solved using next best view (NBV) planner. A common approach for NBV planning is to first generate a set of candidate viewpoints based on different criteria. Afterwards, these candidates are ranked by their reward. Finally, either the best candidate is chosen as NBV or a set of candidates is selected as viewpoints. This concept can be found in each of the following works. But for each of the steps there are very different approaches. They also differ in the information that is known previously and thus in the amount of precomputations and online computations.

3.1. Object Scanning

The first type of task is the scanning of an object in order to create a complete 3D model from a previously unknown object.

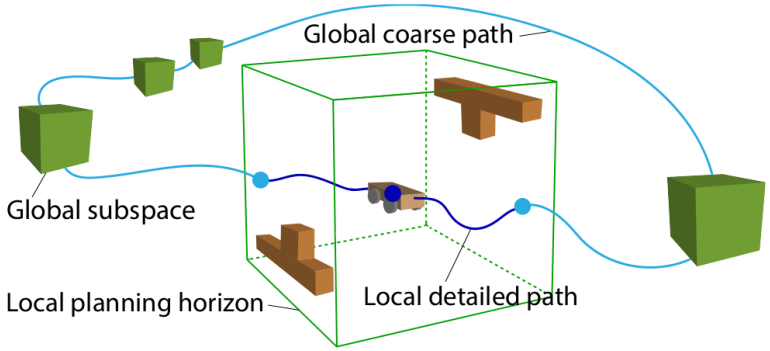
The approach proposed by Cunningham-Nelson et al. [6] starts with no information about the scene except the target position. First, a prior model is estimated by generating waypoints in a cone shape around the target position and executing them. The information that is extracted from the prior model is stored as clusters. For the candidate viewpoints, the workspace is discretized, and for each candidate it is checked whether it fulfills the constraints and if so, how many clusters can be seen from it. The constraints require, that the candidate is inside the robot's workspace and the sensor's reach and at least one cluster is within the field of view of the sensor. A subset of candidate viewpoints is selected by using a greedy approach which always selects the candidate with the most seen clusters. Then the candidates' rewards are updated according to the seen clusters. This is repeated until all clusters have been seen. Then, one or more viewing angles are selected from which all associated clusters are visible. Finally, the selected viewpoints are passed to the path planner, which decides the order in which the poses will be executed.

Daudelin and Campbell [7] focus more on the scanning of objects or scenes of unknown size. Many other approaches need information about the position and the extent of the object, e.g. in form of a known bounding box. This is not required here, the only condition is that when the algorithm starts, a part of the target is in the field of view. This framework computes everything online and needs no precomputation. A 3-dimensional occupancy grid map is used to store the probability occupancy information for each voxel, for the object reconstruction a point cloud is used. In the occupancy grid, two probabilities are stored. The probability that the cell receives a measurement from the next viewpoint, i.e. when it is within the sensor's range and there is a

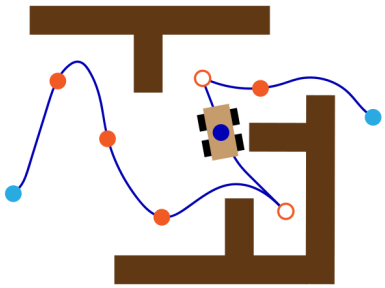
clear line of sight, and the probability that the cell belongs to the object. Frontier cells are unknown cells that border empty and occupied cells and therefore are frontiers of the currently known representation of the object. The probability, that a cell belongs to an object decays with the distance from frontier cells. Frontier cells have the highest probability of belonging to the object. The algorithm dynamically generates a search space of candidate viewpoints based on the current knowledge. Afterwards, the view quality function is evaluated for as many orientations as required to cover all directions of view at the candidate viewpoint position. The view quality function estimates the information gain that could be obtained from the candidate viewpoint. Afterwards, the best candidate viewpoint with the most information gain is selected. While navigating to this viewpoint, the map is updated. The algorithm terminates, if the expected information gain of the next best view is below a threshold. Otherwise, the next candidate viewpoints are generated and evaluated according to the updated map.

3.2. Exploration

The next type of task is the exploration of an unknown environment using NBV algorithms or coverage planning.



(a) Illustration of the exploration framework.

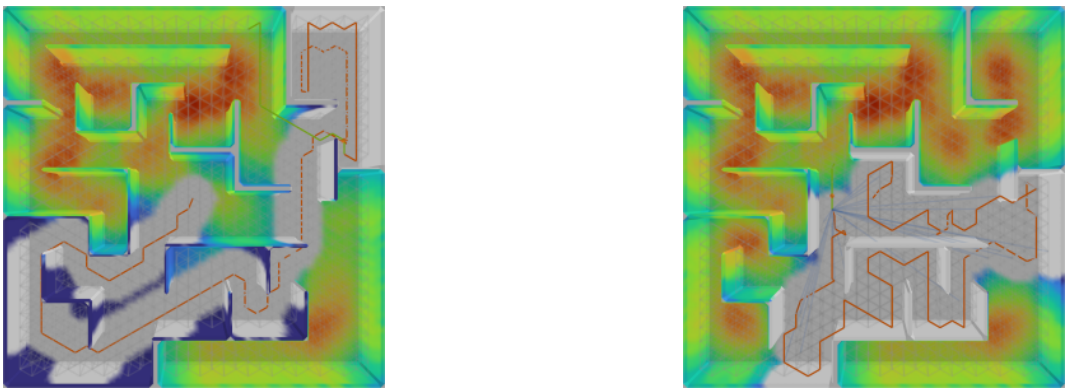


(b) Path in the local planning horizon. The light-blue dots connect the local path to the global one. The orange dots are the selected viewpoints.

Figure 3.1.: Illustrations to the TARE framework [8]

Cao et al. [8] propose an exploration algorithm that uses two representations of the environment, a high-resolution representation for the local planning and a low-resolution one for the global planning (cf. Figure 3.1a). The goal is to find the shortest path which, when followed by the robot, covers all uncovered surfaces, i.e. boundaries between free and non-free space. As here the task is the exploration of an unknown environment, the steps to achieve this goal are repeated as more and more of the environment is explored. This path is always planned in a local area (cf. Figure 3.1b). First, viewpoints are uniformly sampled in this local space. Second, the reward of each viewpoint is computed as the area of surfaces, that can be covered from it. Afterwards, a minimum set of viewpoints needs to be chosen and a path computed using the selected viewpoints. The viewpoint selection is performed probabilistically according to the reward of the viewpoints and the reward of the remaining viewpoints is reduced by the surfaces that are covered by the currently selected viewpoint. After enough viewpoints have been chosen to cover all surfaces, the path is computed. For

this purpose, a Traveling Salesman Problem (TSP) is solved approximately. These two steps, the viewpoint selection and the path planning, are repeated a given number of times and the overall best solution, i.e. the one with the lowest path costs, is used. Therefore, the method optimized the entire exploration path, instead of only maximizing the direct reward for example by using a greedy algorithm.



(a) Traditional (b) Non-greedy

Figure 3.2.: Traditional vs non-greedy exploration planners after 150m travel. ©2021 IEEE [9]

Ericson et al. [9] use an adapted version of greedy. This is to address the issue that greedy in exploration often leads to small unexplored corners that need to be explored later, leading to path redundancies. Therefore, a weight is introduced that weights each surface inversely according to its covisibility. This means that small regions are weighted higher since there are not many other surfaces visible from points where these regions can be seen, and thus the covisibility of these surfaces is low. This weighting ensures that the greedy algorithm does not always choose the viewpoints that can see the most surfaces, but also those that can see the small corners. This is also shown in Figure 3.2. The first image shows the exploration with a traditional planner, here, many small corners are left unexplored. But with the non-greedy plannner, everything is explored immediately. However, since the problem discussed here mainly occurs in the exploration of unknown environments, it is not that relevant for this thesis as here the environment is already known when planning the path. But it might be interesting for possible extensions or future work when combining this work with an exploration task e.g. on only partially known environments.

Another approach is proposed by Steinbrink et al. [10]. Here, a rapidly-exploring random graph is used for sampling-based autonomous exploration of unknown environments. Nodes are sampled locally and globally for improving the exploration efficiency. For each node, the traversability is estimated and is then integrated into the graph. Additionally, a gain-cost ratio is derived from the assumed 3D map coverage at the respective node and the distance to it from the current robot position. The node with the best gain-cost ration is selected as next exploration goal. The graph is continuously built with decoupled calculations of the node gains. The gains are computed using ray tracing methods. As these calculations are decoupled, a better goal can be found while driving to the current one. Then the current goal is interrupted and replaced by this better goal. This way, there is no need to stop the robot after reaching the goal to perform the computations to find the next goal. As soon as no unexplored goal is available anymore, the exploration is terminated.

3.3. Scan of existing buildings

The last type of task is in the context of scanning buildings. Here 3D scanners are often used, but not necessarily mounted on a robot.

Chen et al. [11] plan viewpoints where a terrestrial laser scanner can later be placed in order to create a 3D model of an existing building. Here, a 2D model, i.e. a 2D floor plan, is given as prior information. The coverage is also only planned for the 2D model and not for the final 3D model. Hence, instead of using sampled points or a 3D model for checking the coverage, the lines that form the walls in the floor plan are used. First, the free spaces need to be identified, which is done using image processing tools. Afterwards, the planning starts. Therefore, first candidate positions are sampled within the free spaces. Then the visibility check begins, which is also illustrated in Figure 3.3. Here, a sweep-ray algorithm is used to compute the visible line-segments with respect to a minimum and maximum viewing distance. Then the minimum number of scanning positions need to be found that can capture the building completely. Here, an additional constraint is that a defined overlap between the scans should be present. Chen et al. present 3 algorithms to solve this problem: the Greedy best-first search, Greedy search algorithm with a backtracking process and the Simulated Annealing algorithm. They showed, that for a lot of buildings the solutions using backtracking is better and for a few problems this solution could still be refined using simulated annealing.

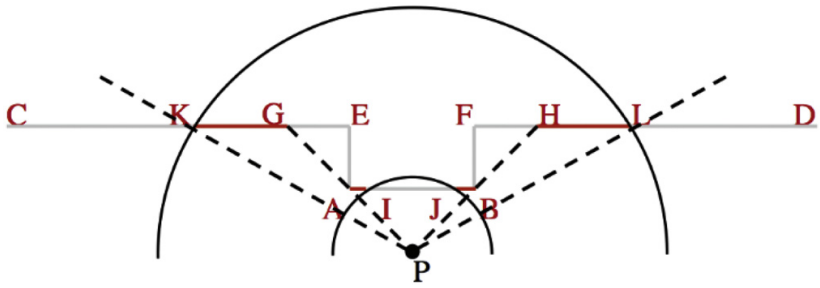


Figure 3.3.: Visibility checking [11]. The circles illustrate the minimum and maximum viewing distance. The candidate position is located at point P. The letters A to L describe the endpoints of the visible (red) and not visible line segments.

3.4. Current system

The following subsection describes some parts of the system currently used by Team Hector[1]. These are the exploration algorithm and some other NBV planners used for inspection tasks.

3.4.1. Exploration

The used exploration algorithm is based on the work proposed by Wirth and Pellenz [12] and uses a 2-dimensional occupancy grid. Here, the next exploration target is selected based on distance from frontiers. Frontier cells separate the known from the unknown regions as they are free cells but have a neighbor cell that is marked as unknown. In order to get new information, these frontier cells might be considered as new

[1] https://www.teamhector.de/

target if there are enough adjacent frontier cells so that the robot can pass it. The costs of a path to a close frontier are computed, which not only takes the distance into account, but also the safety of the route. If more than one potential target is detected, the one with the lowest costs is chosen.

3.4.2. Next Best View planning for one POI

On the current system there is also a next best view planner proposed by Sigg [13] for inspection tasks of one specific point of interest (POI) with the manipulator, e.g. when inspecting manometers. Here, a robot base pose and manipulator position are required in such a way, that the camera can be used to inspect the POI. First, a grid is laid over the known traversable space and all positions where the robot can be placed are evaluated in terms of reachability and distance to any structure or obstacles. Afterwards, for a subset of base poses several camera poses are sampled in a uniform distribution relative to the POIs and the respective base position. Then the camera poses are validated in terms of visibility and reachability in order to ensure that the POI is observable from it. Finally the best base position and the corresponding best camera pose are chosen.

Some other approaches for generating the inspection poses and selecting the best one were presented and compared by Schmidt [14]. The first approach samples a few base positions and checks if these can be used to place the camera in a valid position. For valid base positions several camera poses are sampled and evaluated. The second approach is the other way round. Here first valid camera poses are sampled. Then base positions are searched for each of the camera poses. The third approach is similar to the second one, but here from the POI outgoing rays are used instead of sampling camera positions. Then the ray closest to the current manipulator position on the proposed base position is chosen. All three approaches use basically the same structure as the previously proposed works by generating candidates and evaluating them.

4. Method

The proposed method is based on next best view (NBV) planner as described in the related work in Section 3. An important difference between some of the works and the proposed method is the amount of previously known information and thus in the amount of precomputations and online computations. In this work, a model of the construction site is given as prior information, therefore, the whole path planning process is performed prior to the execution. A typical NBV planner only computes one viewpoint at a time as the next best one. However, when prior information is available, as in this work, it is advantageous to select an optimal set of viewpoints that covers the entire environment rather than just optimizing the instant reward of the next best view.

There are several steps that need to be performed in order to compute the path and execute it to get the required data from the environment. Some of the steps are similar to the ones presented in Section 3, but the different approaches are transferred, combined and extended to be suitable for solving problems presented in this thesis. Most of the steps are almost completely separated from each other with only a small amount of data which needs to be passed via the interfaces. The following section will explain the functionality of each step as well as how they all work together.

4.1. Overview

The Figures 4.1 and 4.2 show the individual steps that are necessary for the computation and execution of the path. The computations are based on a building model obtained from the process of building information modeling (BIM).

Figure 4.1 contains the steps that are required to compute a path on a given building model as well as the data that is transferred between the individual steps. Only the models are used by multiple steps, which is represented by the dashed lines. The individual steps are explained in Section 4.2 in greater detail. First, the building model needs to be converted into suitable models. Afterwards, the candidate viewpoints are generated. They are passed to the reward process and then the best set is selected. These are the viewpoints where data for the final model is recorded during the execution. The viewpoints are positions for the sensor. The corresponding pose for the base link is a waypoint. As the resulting path consists of waypoints, the viewpoints need to be converted to waypoints before computing the best order for them and the final path.

Figure 4.2 illustrates the execution loop. First, the path that comes from the precomputations is loaded. Then, for each waypoint on the path, the robot moves to the corresponding pose and records the data after reaching it. After all waypoints have been visited, the process is terminated and the recorded data is processed or saved.

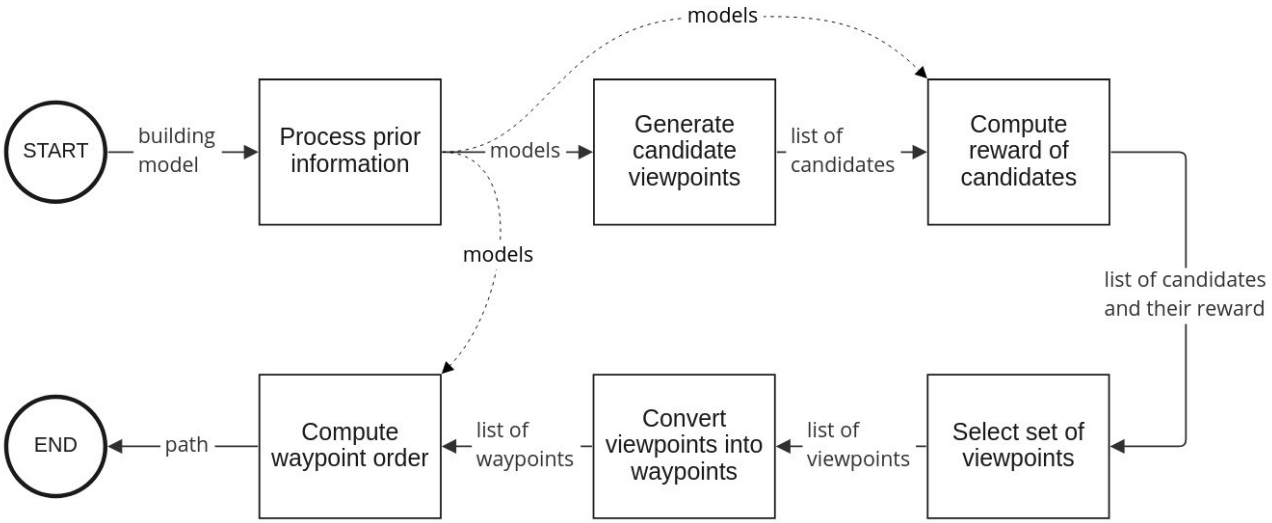


Figure 4.1.: Overview: Precomputations. The passed data is shown on the arrows.

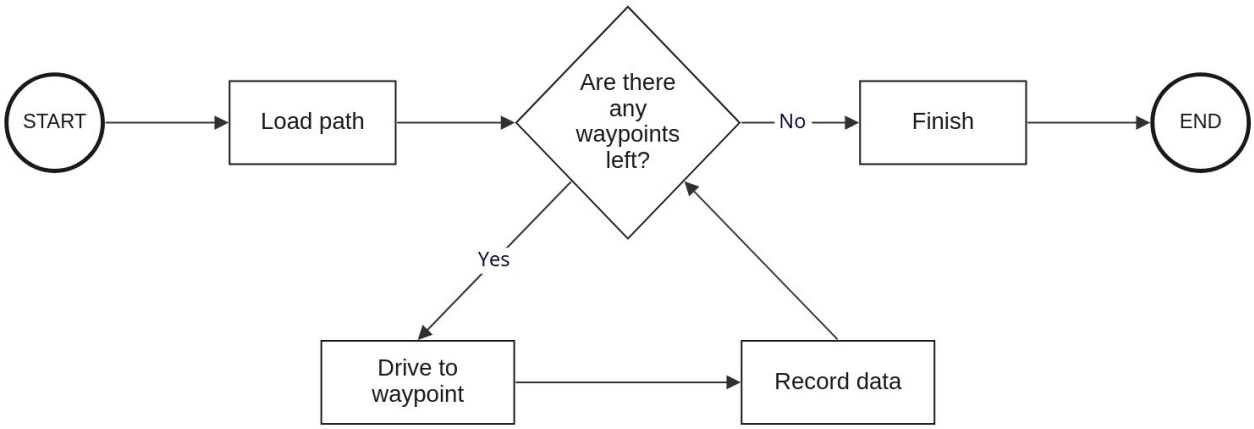


Figure 4.2.: Overview: Execution

4.2. Precomputations

The precomputations include the processing of prior information, the computation of the view- and waypoints and determining their order. These precomputations are meant to be independent from the execution on the real robot or a simulation. It is also required, that the results of the precomputations, e.g. the path, can be stored and loaded for multiple executions.

4.2.1. Process prior information

The prior information, i. e. the building mesh, needs to be present in two ways. The first one is a complete model of the environment, the second one is a subset of the complete model that only contains the parts that

are required to be part of the resulting data, e.g. only pipes, special walls or rooms etc. This subset of the model is called the target model in the following. Both models need to be processed in order to be usable.

The complete model will be used for the candidate generation, the reward process and the waypoint order computation. Therefore, it needs to be converted into several different model representations (cf. Section 2.6). The candidate generation and the waypoint order computation are performed using a mesh representation. The reward process requires a SDF and a 3-dimensional occupancy grid map.

The target model will be used as point cloud to form the target set, a set of points that is used to quantify the coverage. The elements of the target set will be called target points.

4.2.2. Generate candidate viewpoints

In order to compute the required viewpoints, first, candidate viewpoints are generated. This is split into the position and the orientation generation.

Position generation

The candidate positions can be generated in several ways. One would be to lay a grid over the model and use each grid point as a candidate or a number of candidates is randomly generated in each grid cell. But here the problem of setting the height, i.e. the z coordinate, occurs. If z would be set to a fixed value, different floor heights would cause problems, e.g. due to stairs, ramps or multiple levels.

So the candidates need to be generated over the traversable space. This way, stairs, ramps and other uneven floors will be handled. The actual method used depends on the model and the corresponding implementation and is described in Section 5.2.2.

Orientation generation

The candidate viewpoints also need an orientation. In this case, the same position is used for multiple candidates with different orientations.

For the orientation generation, a factor n is given that defines how often 360° will be covered by the generated orientations according to their field of view. The number of generated orientations N is then

$$N = \frac{n * 360^\circ}{fov_h} \tag{4.14}$$

with fov_h as the horizontal field of view of the used sensor. The candidate orientations will be generated equally distributed, starting with identity as orientation.

4.2.3. Compute reward of candidates

In order to choose the final set of viewpoints out of the list of candidates, each candidate is ranked by its reward. The reward of each candidate is the number of target points that are expected to be seen from its pose according to prior information.

For each combination of candidate and target point the visibility is checked. Here, the different representations of the complete model are used. The general assumption is, that a target point is visible if it is within the sensor's range and a clear line of sight exists between candidate and target point. In order to check, if such a clear line of sight exists, ray casting can be used. However, since this is an expensive operation, several other checks are performed beforehand to exclude invisible targets with as little computational effort as possible.

Sensor check

In the sensor check, sensor specific properties are handled. It is examined whether the target point is between the minimal and maximal range of the sensor, as well as if it is in the sensor's field of view.

For the minimal and maximal range, the Euclidean distance between the candidate and the target point is used.

The field of view is defined as horizontal and vertical minimum and maximum angles to the x-axis, in a range of $[-180°, 180°]$. For the check, the candidate pose is used as the origin of the sensor frame. In order to check whether the target point is in the sensor's field of view, first the target point is transformed into the sensor frame. Then the angle between the transformed pose and the x-z-plane for the horizontal and the x-y-plane for the vertical field of view is computed. These angles can be compared with the specified minima and maxima.

The angle between the vector and the plane is always the smaller angle in the range of $[-90°, 90°]$. If the x-value is negative, i.e. the target point is behind the sensor, the angle comparison would always mark the target as inside the field of view, when the smaller angle is in the field of view range. However, this is only true if the range of the field of view for the other, horizontal or vertical, is outside $[-90°, 90°]$, i.e. if the area behind the sensor is also at least partially included in the field of view. If this is not the case, the supplementary angle to $180°$ (or to $-180°$ for negative angles) must be used to mark the correct target points as blocked. An example for this case is a sensor, whose vertical field of view is $[-30°, 30°]$ and the horizontal field of view is $[-135°, 135°]$. For a target point with a negative x-value and the vertical angle $\alpha_v = 15°$ and the horizontal angle $\alpha_h = -20°$, the vertical angle can be used directly for the check, but for the horizontal angle the supplementary angle needs to be used with $\alpha'_h = -180° - (-20°) = -160°$. Hence, the target point is not visible.

Self filter check

The self filter check should remove all points that are not visible because the view is blocked by the robot itself, for example the base or the manipulator.

In order to perform this check efficiently, a mask is precomputed and during the check for each target point, this point is mapped to the nearest point of the mask and checked, whether this point on the mask is visible or marked as blocked. For the mask, a sampled unit sphere is used. The points will be generated as spherical Fibonacci point sets (see Section 2.2) and form a point cloud.

SDF check

For this check, a SDF representation of the complete model is used, either as TSDF or ESDF (cf. Section 2.6.4). The first check is based on the distances to the next occupied cell that are stored in the SDF. If the distance between the candidate and the target point is smaller than the distance to the next occupied cell at one of these two points, there cannot be any obstacle between them and hence, there must exist a clear line of sight.

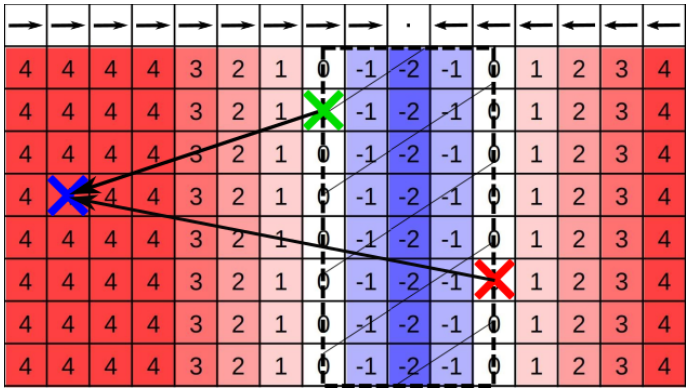


Figure 4.3.: SDF check

The second check is based on the gradients stored in the SDF, but here only the one at the target point is used as well as the direction from the target point to the candidate. If both, the direction and the gradient, have the same sign in each component, the ray starting from the end point will hit an obstacle quickly.

The second check is also illustrated in Figure 4.3. The grid shows the SDF (in this case a TSDF), the top row contains the gradients of the cells in the columns. The blue cross marks the candidate position, the green and red ones mark target points. Also the direction from the targets to the candidate is displayed. If the direction has the same sign in each component as the gradient, like for the red target point, the view is blocked. Otherwise, the target is visible according to the SDF check, like the green target in the figure.

Occupancy grid map check

In the last check the ray casting is performed on a 3-dimensional occupancy grid map (cf. Section 2.6.3). Here a ray is started at the candidate in the direction towards the target point. If the ray hits an occupied voxel before reaching the end point, there is no clear line of sight and the target is not visible from the candidate viewpoint.

4.2.4. Select set of viewpoints

After all candidates have been ranked, the best candidates have to be selected. This task corresponds to the "Set cover problem" (cf. Section 2.3). Solving the set cover problem is NP complete, but there are heuristic solvers.

Several heuristics have been examined in this work and are explained below. All approaches make use of a specified minimum reward. This minimum reward is a regulator of how many viewpoints are selected, which is a trade-off between the number of viewpoints, on which the length of the path depends, and the coverage rate. The minimum reward is especially necessary in this work because, due to the nature of the sampled target set and the visibility tests, the sets of covered targets often have fuzzy edges. So without a minimum reward, many viewpoints would be selected with very few new targets to see.

This also leads to the termination conditions that are used in all selectors. The selection is considered as sufficient and the process is terminated, when there are either no candidates or uncovered target points left, or when there is no candidate left whose reward is greater than the minimum reward.

Greedy approach

A simple approach to solve this problem is the greedy algorithm. Here, the list of candidates is sorted by reward and the candidate with the highest reward is added to the set of selected viewpoints. Then, the rewards of the remaining candidates are updated. Their reward is reduced by the number of target points visible from both, them and the newly selected viewpoint. This is repeated until the selection is sufficient and at least one of the termination conditions described above is met.

This approach is fast as it only consists of sorting the candidate list by reward and select the best. But it might not be optimal due to redundancies. These redundancies occur when multiple selected viewpoints also cover the complete or nearly complete target set of a previously selected viewpoint that had the greatest reward at the time it was selected.

Greedy approach without redundancies

In order to remove the redundancies that might occur in the greedy algorithm, the current approach introduces a backtracking step after each selection. In this backtracking step, for each selected viewpoint it is tested whether all or almost all of its targets are also covered by other selected viewpoints. If this is the case, the viewpoint is removed.

With the backtracking step, the greedy algorithm has no more redundancies and therefore leads to a local optimum.

Probabilistic approach using an exponential distribution

Similar to the greedy algorithm, the probabilistic approach with usage of an exponential distribution uses the idea of selecting the best or one of the best candidates. Here, the candidates are also sorted by reward and then all candidates with a reward smaller than the minimum reward are removed. Afterwards, a random value, obtained from an exponential distribution (see Section 2.1.2), determines the index of the candidate to be selected. This is repeated until the selection is sufficient. In this use case, how often the best candidate is selected depends on the choice of λ.

Since this approach does not always select the best candidate, but sometimes the second or third best, this can lead to better results than the greedy algorithms. For improving the results of probabilistic approaches, they are often executed repeatedly and the best result is stored and returned at the end. This is also done in this work.

Probabilistic approach based on reward

Another probabilistic approach uses an uniform distribution (see Section 2.1.1). Each candidate's probability to be selected is based on its reward. As in the other probabilistic approach, first, all candidates with reward smaller than the minimum reward are removed in order to not distort the probabilities. Afterwards, the rewards of all remaining candidates are summed up. In order to select a candidate depending on a random value, ranges for when to select which candidate need to be specified. For this purpose, the remaining candidates are iterated and for each candidate the upper limit of the range is computed by dividing its reward r_i by the sum of all rewards S computed earlier. Additionally, the upper limit of the last candidate is added to the current limit, which results in a range of $[l_{i-1}, l_i)$ for the i-th candidate with $l_i = \sum_{k=0}^{i} r_k / S$. So if for the random value x obtained from the uniform distribution holds $x \in [l_{i-1}, l_i)$, the i-th candidate will be chosen. This is repeated until the selection is sufficient as described earlier. Similar to the first probabilistic approach, the complete selection process will also be executed repeatedly.

4.2.5. Compute waypoint order

After the viewpoints have been selected, they are converted into waypoints by transforming them from the sensor frame to the robot's base link frame.

Subsequently, the most cost-optimal order for the waypoints must be calculated. First, the costs need to be computed. In this work, the lengths of the paths between every two waypoints are used as costs. The paths are computed on the given model.

Using these costs, the waypoint order is computed. Choosing the best waypoint order with the lowest costs is a Traveling Salesman Problem (TSP) type problem (see Section 2.4). Additionally, it can be assumed as symmetric and metric. Solving the TSP is NP complete, but there exist heuristic algorithms. In this work, several solvers and heuristics for the TSP have been examined and are explained in the following. The heuristics used here all require a fully connected graph.

Brute-force search

The Brute-force search is not a heuristic but an exact solver. Here, all permutations of the given waypoint list will be tested as path, its costs computed and the current shortest one will be stored. In this way, the result is the optimal solution as each possible solution has been tested.

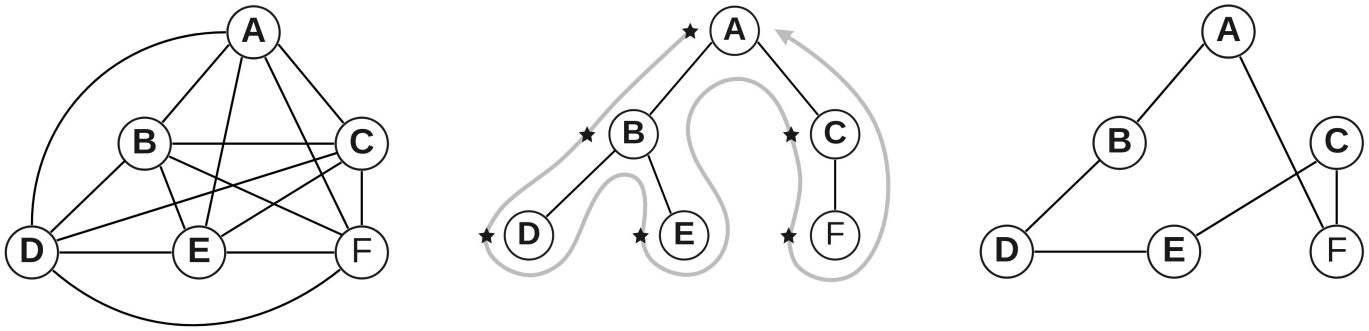
However, this approach can only be used for small TSPs, as with n waypoints or nodes in the graph, $n!$ possible permutations exist. For a TSP, the path can often be rotated later or the start node is given. Due to the fixed first node, the number of permutations reduces to $(n-1)!$, which is still a lot for larger problems. In general, it can be said that it takes a lot of time to test all permutations, which is why this method is usually not practical.

Greedy approach

The greedy approach for solving TSPs is a very simple and fast heuristic. First, a point is selected as the starting point. From this, the next point in the path will always be the one to which the costs are lowest and which is not in the path yet. The greedy algorithm often leads to omitting vertices that are a bit further out until these are the only ones left at the end, which can lead to path redundancies and thus to poor solutions.

Minimum spanning tree approach

For metric and symmetric TSPs, another heuristic exists which is based on minimum spanning trees (MSTs) (cf. [3], chap. 35.2.1). An illustration of the algorithm can be seen in Fig. 4.4 for a metric and symmetric TSP instance with a fully connected graph as shown in (a). First, as can be seen in (b), a vertex is selected as root and then the MST is computed. Afterwards, it is traversed and each node of the tree is added to a list when it is visited first. This list of nodes forms the path as can be seen in (c) as solution of the TSP. As paths for TSPs are circular, the connection between the last and the first node is added.



(a) TSP instance (b) MST with traversal (c) Resulting path

Figure 4.4.: MST solver for the TSP

Simulated annealing

Simulated annealing (SA) is a probabilistic algorithm to approximate a solution for which the cost function is in a global optimum. In the case of the TSP, this would be a solution with minimal path costs. The algorithm tries to improve a given initial solution without getting stuck in local optima. The name comes from annealing in metallurgy. Therefore, the parameter of the algorithm is called *temperature* and the adjustment of this parameter *cooling schedule* or *cooling rate*.

The initial solution can be any valid solution, in this work random initial solutions are used as well as the solutions generated by the greedy and the minimum spanning tree approach. But depending on the quality of the initial solution, i.e. how close the initial solution is to the global optimum, the simulated annealing algorithm requires more or less steps to find an optimal solution. Additionally, simulated annealing is not guaranteed to result in the global optimum, so depending on the initial solution it might also happen, that a local optimum which is too deep cannot be left and the initial solution leads to a worse final solution than a random one that required more steps.

The algorithm starts with an initial solution and an initial temperature. In each iteration, a neighbor, i.e. a slightly mutated version of the current solution, is tested and the cost function computed. If this neighbor is a better solution, it is accepted as new current solution. But with a computed acceptance probability, a worse solution might also be accepted in order to avoid being stuck in local optima. The temperature is adjusted at the end of each iteration according to the cooling schedule.

Any monotonically decreasing sequence can be used for the cooling schedule, and the best cooling rate and initial temperature highly depend on the given problem.

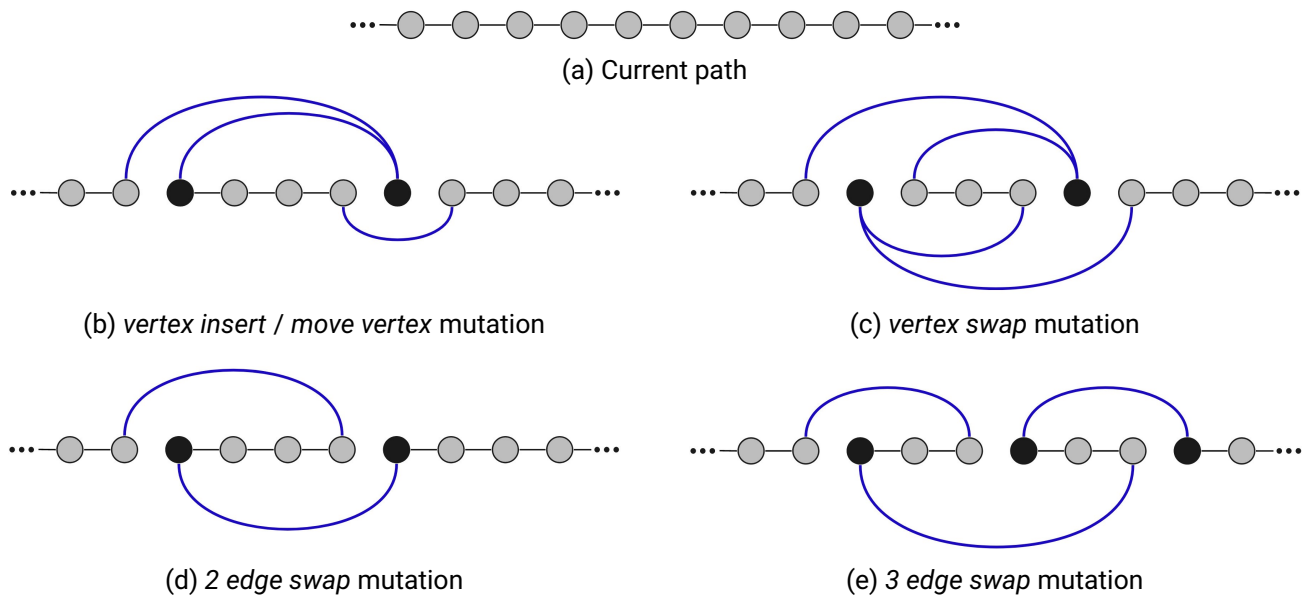


Figure 4.5.: Neighbors of a current path shown in (a) generated with different mutations.

The acceptance probability depends on the temperature T, the costs of the current solution $C_{current}$ and the costs of the neighbor solution that is tested $C_{neighbor}$. The acceptance probability P_{accept} using the so called "Metropolis acceptance criterion" (cf. [15]) is

$$P_{accept} = \exp\left(-\frac{C_{neighbor} - C_{current}}{T}\right) \tag{4.15}$$

With decreasing temperature T, the acceptance probability for a worse neighbor decreases. Due to this, worse solutions are accepted more often at the beginning and as the algorithm progresses they are accepted less and less often. For example, a neighbor path that is $1\,\text{m}$ longer than the current path is accepted for a temperature $T = 5$ with a probability of $P_{accept} \approx 81,9\%$ but for a temperature of $T = 0.5$ only with a probability of $P_{accept} \approx 13,5\%$.

For the neighbor generation, different mutators can be used. They can also be combined and in each iteration one of the mutators is selected with a specified probability (cf. [16]). Another possibility is to execute all mutators in each step and select the best solution as neighbor (cf. [17]).

In Figure 4.5 different neighbors with one mutator applied are illustrated. All mutators use randomly selected vertices to perform the mutation. The *vertex insert / move vertex* mutation moves a selected vertex in front of another vertex. In the *vertex swap* mutation two vertices are swapped. The *2* and *3 swap edge* mutations work similar. Here, 2 or 3 vertices are randomly selected and the edges, that end in these vertices, are swapped. While doing this, the path between these vertices is reversed.

4.3. Execution of the path

After the path was computed and stored, it needs to be executed. Here, the robot drives along the path and records data at each waypoint.

4.3.1. Drive to waypoint

In order to navigate the robot between waypoints, different methods can be used.

One is to use an existing path planner that only receives the goal, i.e. the next waypoint, and plans a path to it based on different maps, e.g. obstacle maps. The advantage of the usage of such a planner is that here the paths between the waypoints are planned on execution time and can therefore take dynamic obstacles and the like into account. But the planning can result in different problems, as it is only based on the currently known map. Every time the next waypoint is in a previously unknown part of the map, unfavorable paths may be planned. This can lead to large detours, because the robot moves along the path towards the waypoint, but as soon as the map is updated because for example when new obstacles are detected, a new path has to be calculated. In the worst case, the previously planned path is completely blocked and the robot has to turn around and search for a new path until a free path is found and enough of the unknown part of the map is detected.

Another method comes from the fact that for the costs the paths have already been computed on the model (see Section 4.2.5). These computed paths can be stored and used for the execution. This has the advantage, that there is no additional planning required and it does not matter, how much the map was already explored. But a large disadvantage is, that they cannot adapt to dynamic obstacles, that are not in the model but in the real environment. So here the precomputed paths need to be monitored while being executed and adapted if required. For this, only the blocked path parts are replanned while all valid parts of the paths remain.

4.3.2. Record data

When a waypoint is reached, the data recording starts. Here, for example point clouds from a lidar or images from a 360 degree camera can be taken and stored. For several types of data recording it might be important to wait at the waypoints for a predefined time, e.g. in order to ensure that the 360 degree camera is not moving anymore or a Lidar, that is mounted on a rotating base, can perform a full turn in order to scan the whole environment.

5. Implementation

The following section describes the implementation details and the used libraries. The software implemented in the context of this work can be found mainly in the package `three_dimensional_coverage_path_planning`[1].

5.1. Used frameworks and libraries

5.1.1. ROS

The work is based on Robot Operating System (ROS)[2]. ROS is a framework and collection of libraries for robot software development and execution. The processes are organized in so-called nodes. Nodes can communicate with each other in several ways, e.g. via topics, actions or services. ROS also provides a parameter server. Here, parameters can be specified in configuration files and are loaded automatically on the parameter server. Nodes then can retrieve parameters from the server instead of managing them themselves.

For the visualization, *rviz* can be used. In this work, the models, visibility checks and paths are visualized as well as the recorded data.

5.1.2. ROS pluginlib

Using the library `pluginlib`[3], ROS packages can load and unload plugins dynamically from a runtime library. For each type of plugin, a base class is required and the plugins themselves are subclasses. Using plugins, a software can easily be adapted or extended without necessarily changing the base itself.

5.1.3. PCL

The Point Cloud Library (PCL)[4] is a standalone library which contains several classes and algorithms to create, manage and process point clouds. Each point consists of a position and can have additional data, e.g. an intensity or a color.

One filter algorithm that is used in this work is the `pcl::VoxelGrid` filter. Here, the cloud is split into voxels with a given leaf size. All points in a voxel are replaced with the centroid of the voxel.

[1] `https://git.sim.informatik.tu-darmstadt.de/hector/3d_coverage_path_planning`
[2] `https://www.ros.org/`
[3] `http://wiki.ros.org/pluginlib`
[4] `https://pointclouds.org/`

Another filter that is used in this work is the `pcl::RandomSample`. A given number of random sampled points are selected using a uniform probability, all other ones are discarded. An important difference to the voxel grid filter is that a selection of original points is kept, whereas in the voxel grid filter the original points are replaced by the centroid.

5.1.4. Grid map

The grid map [18] is an implementation of 2-dimensional grid maps that can contain different map types in different layers. In this thesis, only the 2-dimensional occupancy grid map layer (cf. Section 2.6.3) is used.

5.1.5. OctoMap

An OctoMap is a 3D occupancy grid (cf. Section 2.6.3) based on an octree [19]. In this thesis it is used to perform ray casts in the map.

5.1.6. Voxblox and Voxblox Ground Truth

Voxblox is a library that can be used to build TSDFs and ESDFs (cf. Section 2.6.4). It also provides a volumetric mapping.

SDFs are normally created from sensor data. But the library `voxblox_ground_truth`[5] provides the possibility to create TSDFs from Polygon File Format (.ply) meshes.

5.1.7. Mesh navigation

The mesh navigation [20] mainly provides a navigation server for *Move Base Flex* [21]. But it also contains several libraries that can be used separately, as it is the case in this work.

The first library is the `mesh_map`. Here, a mesh stored in a Hierarchical Data Format (HDF5) file is loaded and managed. It provides methods to iterate the vertices and to get the neighbor vertices of each vertex. A map consists of several layers. In this work, two layers are used, the `HeightDiffLayer` and the `InflationLayer`. The first one contains costs for the height differences in a local range, the second one contains costs defined by their distance to a lethal vertex in another layer, in this case the `HeightDiffLayer`.

The second library used in this work is the core package, named `mbf_mesh_core`. It contains the `MeshPlanner`, the base class for plugin based path planning on a `MeshMap`. The plugins contain different planner implementations, e.g. the `dijkstra_mesh_planner` and the `cvp_mesh_planner`.

5.1.8. FlexBE

FlexBE [22] stands for flexible behavior engine. Each behavior is represented by a state machine. There are several built-in states that can be used as well as self developed states. States can communicate with other nodes using for example actions.

[5] `https://github.com/ethz-asl/voxblox_ground_truth`

5.2. Precomputations

For the precomputations, a FlexBE behavior is designed which handles their execution and afterwards saves the path to a file so that it can be loaded for the execution of the path (cf. Section 5.3). It communicates with the software using actions.

5.2.1. Process prior information

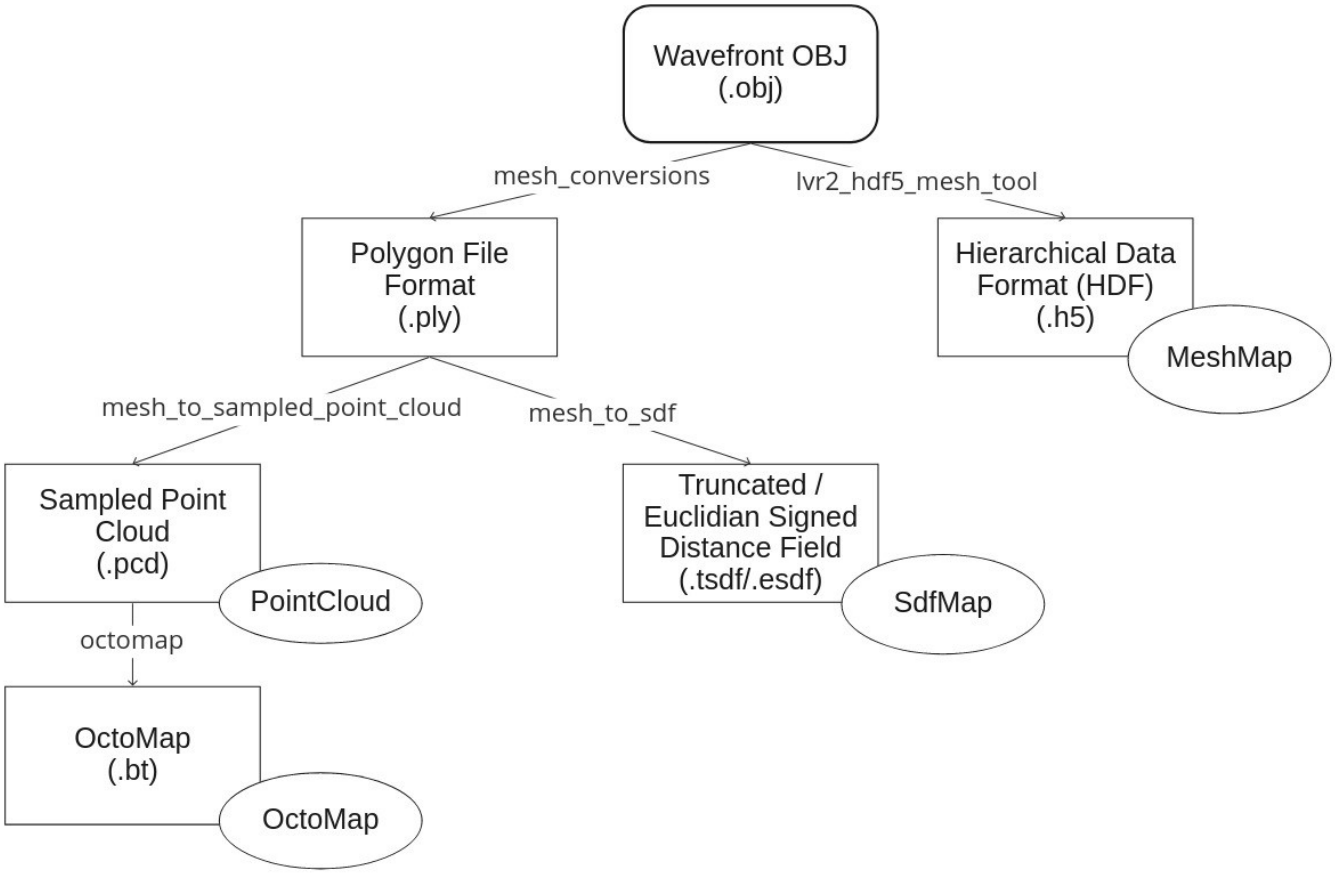


Figure 5.1.: Model conversions: file and object types; the connections contain the names of the libraries or tools that are used to convert the model

The building model is given as an object file (.obj) and needs to be converted into several other model representations. The conversions between the different file types are displayed in Figure 5.1 and explained below.

For the conversions, a package named `hector_model_conversions`[6] was implemented. It contains three libraries, that can convert a mesh into other model representations.

[6] `https://github.com/tu-darmstadt-ros-pkg/hector_model_conversions`

The first library named `mesh_conversions` is used to convert the input mesh file into another mesh file type, the Polygon File Format (.ply) using the functionality of the library `pymeshlab` [23, 24].

The next model type that is required is a point cloud. Here it is important that the points are sampled when the mesh is converted, since otherwise the resulting cloud will only contain the vertices of the mesh. Depending on the mesh, these may be far too few points for the point cloud to be meaningfully used further. How many points are sampled can be chosen using a parameter. If necessary, a `pcl::VoxelGrid` filter (cf. Section 5.1.3) can be applied to prevent too many of the sampled points from being too close together. The whole conversion process is implemented in a second library named `mesh_to_sampled_point_cloud`, that uses the functionality of the `pcl_mesh_sampling` tool (cf. Section 5.1.3). Using the resulting point cloud, the OctoMap is generated.

Another model that is generated from the .ply mesh file, is the SDF. For this, the library `mesh_to_sdf` was implemented. Here, the library `voxblox_ground_truth` (cf. Section 5.1.6) is used to compute the TSDF from the mesh. If necessary, the ESDF is then calculated from the TSDF. In the main package, a class named `SdfMap` is implemented in order to handle both, TSDFs and ESDFs. This way it is configurable whether a TSDF or ESDF should be used for the computations.

The last model used is a representation of the mesh used by the `mesh_navigation` (cf. Section 5.1.7). Here, the data is stored in a Hierarchical Data Format (HDF5) file which later also contains the layer information of the mesh map. Using the tool `lvr2_hdf5_mesh_tool` provided by the `mesh_navigation`, the HDF5 file is generated from an .obj or .ply file. However, for the HDF5 file to be usable, the source mesh must be manifold (see Section 2.6.1). The `mesh_navigation` uses the edges of the faces of the mesh for planning, so additionally it is useful, when the faces are not too large. Hence, depending on the mesh, it is required to recompute the mesh with a specific face size or vertex distance. In this recomputation, the mesh can also be made manifold if this was not the case before.

5.2.2. Generate candidate viewpoints

For the candidate generation, the positions and orientations are generated separately. For each candidate position, a number of orientations are generated, hence, there are several candidates with the same position but different orientations.

The candidate generation is implemented as plugin system (cf. Section 5.1.2). In this thesis only one plugin is implemented, but this concept allows other candidate generation methods to be implemented easily in the future. For example, when a waypoint is blocked during the execution of the path and needs to be replaced, using a local grid candidate generator (cf. Section 4.2.2) for a small area around the blocked waypoint can be reasonable.

Position generation

As described in Section 4.2.2, the candidates need to be generated above the traversable space. In this thesis, two methods are implemented. Both use the `MeshMap` as provided by the `mesh_navigation` package (cf. Section 5.1.7), which is loaded from the mesh file in the HDF5 format.

For the first method, the list of vertices in the `MeshMap` is iterated and each vertex, whose costs are below a specified threshold, is used as potential candidate. But as the mesh also has flat surfaces on the ceilings and not only on the floor, the normal of the vertex is also checked for a non-negative z-value. However, this

method can lead to unreachable candidates, e.g. in inaccessible rooms or, if the mesh contains no ceiling, on top of the walls.

In order to avoid this, a second method has been implemented. Here, a reachable start point needs to be given as parameter. If it is not possible, to provide such a reachable point or the point is invalid, the first method is still used as a fall back solution. Starting from the given point, all neighbors of the current vertex, that have not already been examined, are tested. If their costs are lower than a specified threshold, they are added to a queue and to the list of potential candidates. Then, the next vertex is the first from the queue. This way, only vertices that are reachable from the start point are iterated and added as potential candidates.

Depending on the mesh, there can be a large amount of potential candidate positions. This is the case for example when after recomputation the mesh has vertices every 0,05 m, as described in Section 5.2.1. In order to reduce the number of candidate positions, a filter is applied. The first approach was to use a `pcl::VoxelGrid` filter (cf. Section 5.1.3). But as described earlier, here all points in a voxel are replaced by the centroid of the voxel, so a new point is used instead of an existing one. If the voxels contain a wall or other invalid areas, e.g. where the costs are too high, depending on the used leaf size it can happen, that the centroid lays inside a wall or another invalid area. So in this use case, the voxel filter cannot be used since a filter is required, that preserves original points in order to ensure their validity. The approach implemented in this thesis uses the `pcl::RandomSample` (cf. Section 5.1.3). As here the number of samples is required which is difficult to estimate for a new model, it was chosen to use the percentage of candidates to sample, which is specified in the parameters.

For the position generation the mesh vertices are used. However, since positions for viewpoints are to be generated here and the mesh vertices are on the ground and thus are waypoints rather than viewpoints, they must be transformed from the robot's base link to the specified sensor frame.

Orientation generation

The orientation generation is implemented in the base class and can be used by all plugins, but they can also use their own orientation generation methods.

The base class version implements the concept described in Section 4.2.2, with the factor n given as a parameter. The sensor specifications like the field of view are also retrieved from the parameter server. The orientations are first computed as Euler angles and later converted to quaternions. Here, only a rotation around the z-axis is required. Starting with 0°, the rotation angle is incremented for each new orientation by $360°/N$ with N as number of orientations to generate.

5.2.3. Compute reward of candidates

In the candidate reward process, for each candidate all visible targets are identified. For this purpose, the visibility of all target points for the current candidate is determined using the different visibility checks. Only if all checks are successful, a viewpoint counts as visible.

The visibility checks are implemented in a plugin system (cf. Section 5.1.2). This way, it is possible to specify in a configuration file which visibility checks are to be used and in which order the checks are performed. Depending on the model, sometimes it is not reasonable or possible to use all checks described below. Additionally, with the plugin system new checks can be easily added for example for a special type of model.

Sensor check

The sensor checker is implemented as described in Section 4.2.3. The sensor specific minimum and maximum range is retrieved from the parameter server as well as the values for the field of view.

Self filter check

In the initialization of the self filter check, the mask is computed as described in Section 4.2.3. The generated spherical point cloud is used as input cloud for the `robot_body_filter` [25]. Here, all points that are blocked by the robot itself are filtered out. Afterwards the original cloud and the filtered cloud are compared. Any point that is in both will be marked as visible, the points that are only in the original cloud will be marked as blocked in the mask (Figure 5.2).

In the visibility check itself, the target point is transformed to the candidate pose frame. After normalization it can be used as direction from the sensor to the target point. This direction is mapped to the nearest point on the mask using an algorithm proposed by Keinert et al. [1].

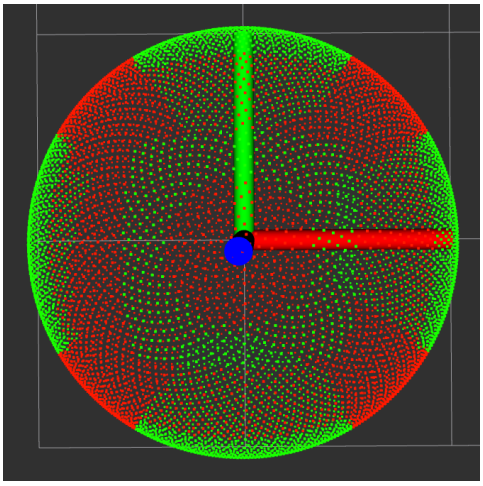


Figure 5.2.: Self filter mask. The visible points are green, the others red.

SDF check

The SDF check is implemented as explained in Section 4.2.3. Since the distances and gradients are needed for each candidate and for each target point, they are stored in a map in order to avoid having to recalculate them each time.

Occupancy grid map check

The OctoMap (cf. Section 5.1.5) provides the possibility to perform a ray cast. Here, a start point and a direction need to be passed, as well as a maximum length of the ray. For the maximum length the distance between the candidate and the target point is used. If the ray hits an occupied cell, the center of this voxel is returned as end of the ray. Otherwise, the center of the voxel where the maximum length of the ray is reached is returned. This end needs to be compared with the target point. However, since the target point can be anywhere within the voxel, the distance between the end and the target point must be checked. If this distance is below a specified threshold, the ray cast is considered successful and there exists a clear line of sight, the target point is visible. Otherwise an occupied voxel was hit before the target was reached, the target is not visible. Since the ray cast returns the center of the voxel, the target point can be at most $(voxel\_size * \sqrt{3})/2$ distant from the returned end to still be in the same voxel, which is the value used for the threshold. This threshold is additionally increased by a configurable tolerance to compensate for inaccuracies.

5.2.4. Select set of viewpoints

For the selector implementation a plugin system is also used (cf. Section 5.1.2) as here different selectors are implemented and it also allows to easily implement other selectors in the future.

Base class

The base class provides functionality that is used in all or almost all selectors. Apart from some general methods like an initialization and some accessors, the base class contains a method to check if a selection is sufficient according to Section 4.2.4 and can be terminated.

Additionally, it contains a method that is called when a viewpoint is selected. In this method, first, the viewpoint is added to the list of selected viewpoints and removed from the remaining viewpoints. Then the rewards of all remaining viewpoints need to be updated. Each viewpoint instance keeps track of all visible targets, all targets that can be newly seen from this viewpoint if it gets selected and all targets that are visible but have been covered by other viewpoints. In the process of updating the reward, the intersection of the targets sets, that are visible from the newly selected viewpoint and from the current one, is formed in order to find all target points that are visible from the current viewpoint but now have been covered by the newly selected one. The points from this intersection set are moved to the list that contains all viewpoints that have been covered by other viewpoints and the reward is decremented by the size of the intersection set. Finally, the targets that could been seen from the newly selected viewpoint are removed from the list of uncovered target points.

Greedy approach

The greedy approach is implemented as described in Section 4.2.4. The list to be sorted is the one containing the remaining candidates which is implemented as a `std::vector`. However, the list is sorted by reward in ascending instead of descending order and then the last element is selected as best viewpoint. Otherwise, the first element would be selected and removed after selection. But removing the first element of a `std::vector` is computationally much more expensive than removing the last element, because all other elements of the list would be moved one place forward.

Greedy approach without redundancies

For most selectors, mapping viewpoints to their visible targets is sufficient. However, to remove redundancies, mapping in the reverse direction is also required, i.e. mapping each target point to all selected viewpoints that can cover this target. This is required in order to decide which selected viewpoint might be redundant. The map is populated during the selection process.

A selected viewpoint is redundant, if all of its covered target points can also be covered by other selected viewpoints. But due to inaccuracies in the ranking process, the edges of the covered target point sets are not clearly delimited, but blurred. So, similar to the minimum reward, a parameter is introduced to compensate for these inaccuracies. This parameter describes the maximum number of covered targets that may be lost when removing a selected viewpoint. *Lost covered targets* are the targets, that are no longer covered after the almost redundant viewpoint was removed. These are lost after the removal, but might be covered by later

selected viewpoints. A viewpoint that was once removed as redundant is not added to the list of remaining candidates as it would not be selected due to small reward or if it would be selected, be removed again.

For the removal process, all selected viewpoints are iterated after each selection of a viewpoint. All targets that are marked as seen from the viewpoint are checked in the mapping, if they are also covered by another viewpoint. If this is not the case, this target point is counted as a lost covered target point. If the number of lost covered target points is greater than a given maximum, the viewpoint is assumed to not be redundant and the next selected viewpoint is examined.

But if the number of lost covered target points is below the threshold, the viewpoint is assumed to be redundant as almost all of its covered targets can also be covered by another viewpoint. In this case, all covered targets of the redundant viewpoint are iterated and assigned to the first of the other viewpoints, that can also cover it. In this process, the reward of the other viewpoint is also increased. The redundant viewpoint is removed from the mapping. If the target is one of the lost covered target points, it is added again to the list of uncovered target points and for each remaining candidate it is checked, whether they can cover it. If this is the case, the lost target is added to their list of targets to be covered and their reward is increased.

Probabilistic approaches

The two probabilistic approaches are implemented as described in Section 4.2.4. When they are executed repeatedly, a new solution is not only accepted and stored as best when it contains less selected viewpoints than the old solution but also when the number of selected viewpoints is the same but the number of uncovered target points is smaller than in the old solution.

5.2.5. Compute waypoint order

In order to allow the implementation for calculating an order for a set of waypoints to be used outside of this work, it was developed in a separate package with the name `hector_waypoint_order`[7]. The package provides both, classes for computing the costs between the waypoints as well as classes for solving the TSP (cf. Section 4.2.5).

Using the cost computer, a cost map can be computed with start and end point as key and the costs as values. This cost map as well as a list of all waypoints is passed to the TSP solver.

Cost computer

The cost computer is implemented as plugin system (cf. Section 5.1.2). Therefore, the `hector_waypoint_order` package contains a base class for the cost computer which provides methods for initialization and computing the costs.

In this work a subclass was implemented, which computes the costs as the length of the paths between the waypoints (Figure 5.3). It uses a path planner, that is specified using a parameter, and computes the paths between all pairs of waypoints. If the used path planner has not already computed the length, this is done using the Euclidean distance between the points of the planned path. The paths are also stored in a map and

[7] `https://github.com/tu-darmstadt-ros-pkg/hector_waypoint_order`

can be retrieved from the cost computer, in order to allow the usage of the already planned paths instead of having to recompute them later, for example if they are to be used for navigation.

The path planner is also implemented as plugin system and a base class is provided in the `hector_waypoint_order` package. In this work, the path planner of the `mesh_navigation` package was used (see Section 5.1.7 and Figure 5.3). However, since a mesh and navigation within it are a very special use case, the path planner plugin `MeshNavigationPathPlanner`, that uses the `mesh_navigation`, was not implemented in the same package as the base class, but in the main package of this thesis. Otherwise it would result in much more dependencies of the `hector_waypoint_order` package that are used only in very few cases. The plugin allows to use different planners provided in the `mesh_navigation` as they have also been implemented as a plugin system.

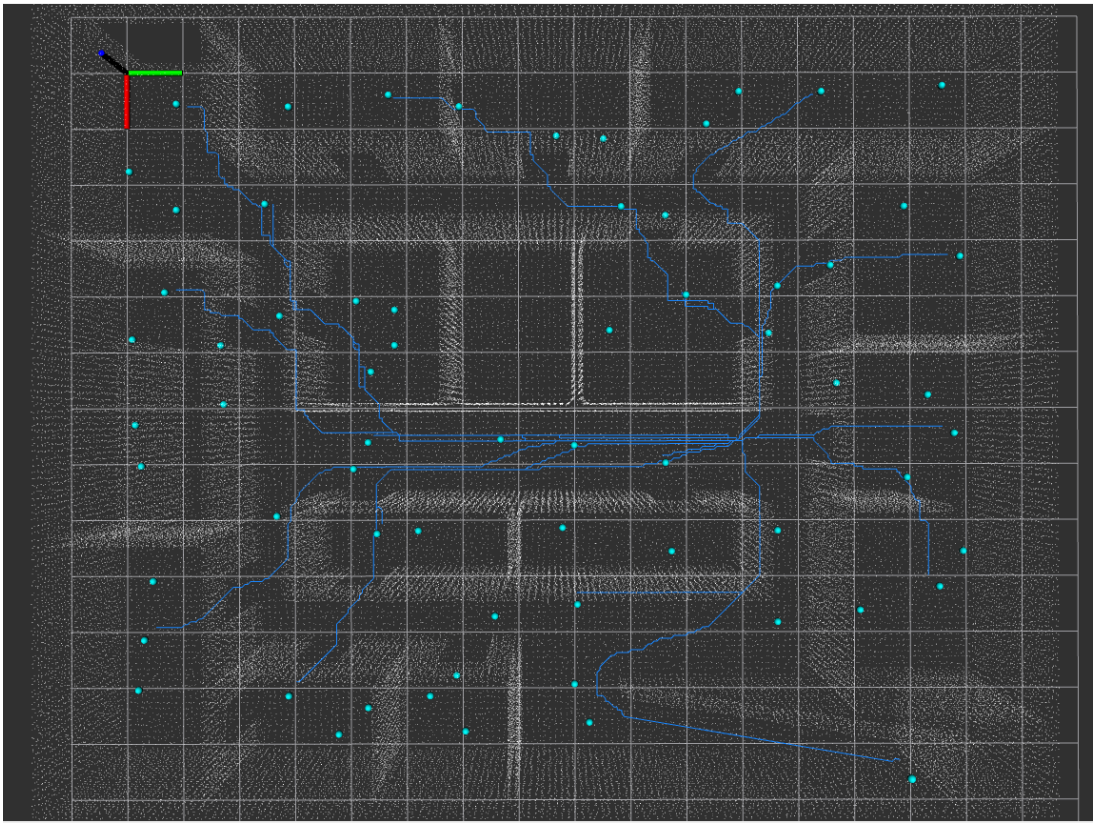


Figure 5.3.: Subset of the paths that are calculated in the cost computer. The dots are the selected viewpoints, furthermore the target model can be seen.

TSP solver

With the `WaypointOrderComputerBase` a base class for using the subclasses as plugins is provided. This way, the TSP solver to use for computing the waypoint order can be specified in the parameters and loaded on execution time. The base class provides a method for the initialization and a pure virtual method for computing the waypoint order. It also contains a method to compute the costs using the stored cost map for a path given as list of waypoints, as the objective function of the TSP contains the complete path costs and hence is used in all implementations of TSP solvers.

Brute-force search This solver is implemented using the C++ Standard Library method `std::next_permutation`[8]. In order to reduce the required number of permutations at least a bit, the fact that a circular path is searched is exploited here. Therefore only the path from the second to the second last waypoint is permuted, assuming that the first and last waypoint are the same. Another small improvement is, that the method for computing the costs of a complete path has an argument that describes the maximum allowed costs. If this argument is passed, the cost computation is aborted as soon as these costs are exceeded, which saves a few lookups in the cost map.

Greedy search The greedy search is implemented as described in Section 4.2.5. The starting point is selected randomly. A list of all unvisited waypoints is held and in this list the waypoint with the lowest cost, according to the cost map when starting from the current waypoint, is selected as next waypoint. Finally, after all waypoints have been visited, the first waypoint is added to the end in order to make the path circular again.

Minimum spanning tree search For the minimum spanning tree (MST) search, the method `metric_tsp_approx_tour` from the Boost Graph Library [26] (BGL) is used.

Simulated annealing The simulated annealing algorithm is implemented as described in Section 4.2.5.

The default initial solution for the simulated annealing algorithm is the unordered list of waypoints that is passed to it as an argument. Here the waypoints have been added in the order they have been selected, so this is considered to be a nearly random order. Using a method, this initial solution can be overwritten by another given solution, e.g. generated by MST or greedy search.

As also described earlier, the mutators use randomly selected vertices for the mutation. In order to ensure that only valid mutations are generated, vertices are selected repeatedly until all are valid.

It might happen that the last solution is not the best one, e.g. if after a good intermediate solution due to a badly chosen initial temperature and cooling schedule, this solution is left and the algorithm gets stuck in another local optimum which is worse than the solution already seen. In order to avoid this, the best solution ever seen is saved and returned at the end, instead of always using the last solution.

Simulated annealing with MST or Greedy search As described in Section 4.2.5, the solution of another heuristic solver can be used as initial solution for the simulated annealing algorithm. In this thesis this was implemented for the greedy and the MST search. The solver is instantiated and the solution for the TSP computed. This solution is then set as initial solution for the instance of the simulated annealing solver.

[8] `https://en.cppreference.com/w/cpp/algorithm/next_permutation`

5.3. Execution of the path

For the execution of the path, a FlexBE behavior (cf. Section 5.1.8) is designed. In this, first the path is loaded from a file and rotated so that the first pose is as close as possible to the current robot position. Afterwards, the action to initialize the path execution in the `three_dimensional_coverage_path_planning` package is called, which also loads the path into the software. The next part in the behavior is repeatedly retrieving the next waypoint from the path and moving to it until all waypoints have been visited. For this purpose, a sub-behavior is designed. It first checks, whether the next waypoint has a path or not. This is important, as there is no precomputed path between the robot's start pose and the first pose of the path. If there is no path given, it plans a path to the waypoint and follows it, otherwise the action to move to a given waypoint of the developed software is called (cf. Section 5.3.1). After reaching the waypoint, regardless of whether a path was given or not, the corresponding action is also called here, which handles the data recording (cf. Section 5.3.2). Finally, after all waypoints have been visited, the execution is finished by calling the action that performs the post-processing of the data (cf. Section 5.3.2).

5.3.1. Drive to waypoint

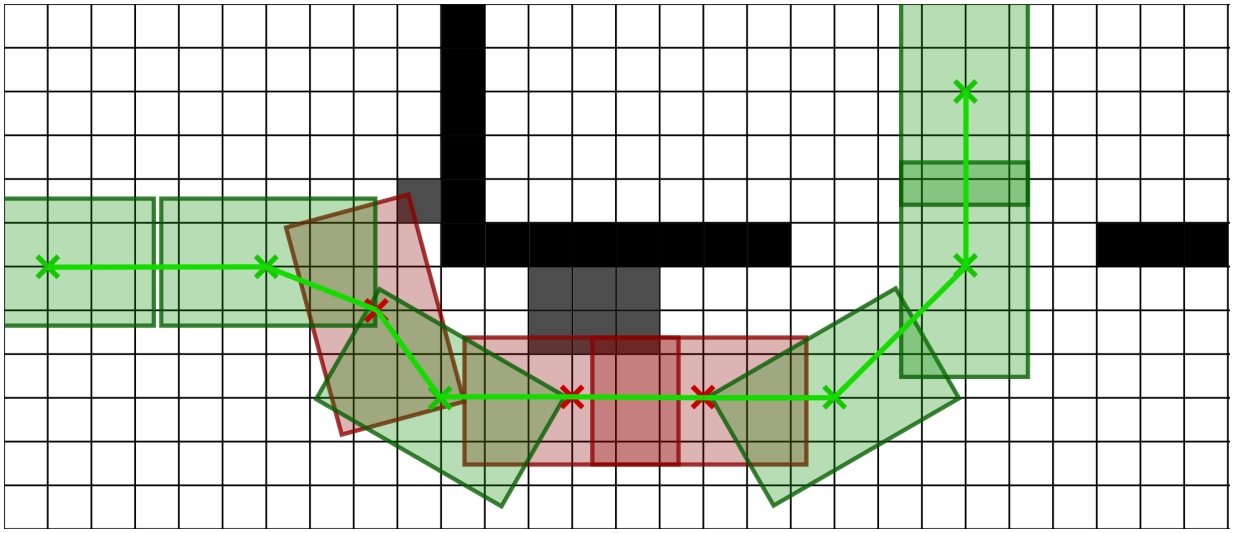
As discussed in Section 4.3.1, the previously computed paths are used in this thesis. Each time, only the path to the next waypoint is considered. As also already described, these paths must be updated during execution to avoid dynamic obstacles.

Each time a new grid map (cf. Section 5.1.4) is received, the current path is checked and updated if necessary, which is also illustrated in Figure 5.4. Only the part of the path that is in front of the robot is checked, since points that have already been passed are no longer relevant. For the detection, if a path is blocked, a polygon iterator is used. Here, a polygon using the robot's length and width is defined and transformed to each pose in the path. Then the occupancy value of each cell inside the polygon is used to check, whether the pose is blocked or not. Several blocked path parts with only a few free poses in between are joined together to form one large path part, which is later replanned. Otherwise, this would lead to zig-zag paths as the blocked parts are replanned and it returns to the original path for only a few poses in between, only to be led away again onto a second replanned path part. For the replanning, the `grid_map_navigation_planner`[9] is used.

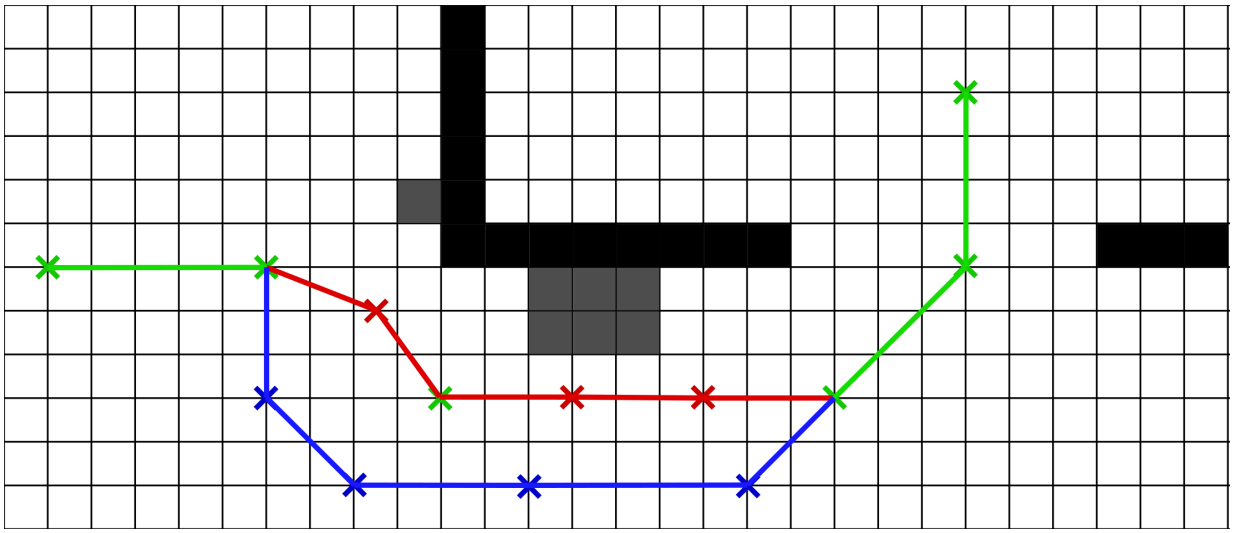
Finally, the updated path is sent to the `vehicle_controller`[10], which first smoothes the path and then manages the path following. For this work, the Dynamic Arc Fitting (DAF) controller (cf. [27, 28]) is used.

When the robot follows the path, it sometimes has problems with narrow places like doors and can get stuck. Often this is only a misalignment problem of a few centimeters. This can also happen if the robot is rotating on the spot when it aligns to the next path section after reaching a waypoint and is too close to the next wall or other obstacle. In these cases it is stuck and needs to try to free itself from the situation. A simple stuck recovery is to drive backwards a short distance. Afterwards, the movement can be replanned based on the new position of the robot and is then often successful.

[9] `https://github.com/tu-darmstadt-ros-pkg/grid_map_navigation_planner`
[10] `https://github.com/tu-darmstadt-ros-pkg/vehicle_controller/tree/new_controller`



(a) Originally planned path with footprint polygons. The green ones are free, the red ones are blocked.



(b) Complete updated path. The green sections are the ones kept from the original planned path, the red is the blocked and therefore discarded one, and the blue section is the updated path.

Figure 5.4.: Path update during execution

5.3.2. Record data

For the data recording, also a plugin system is used. This allows easy adaption to different sensors, for example for the usage of a lidar for a 3D model or a 360 degree camera for a virtual room tour. The base class implementation contains an initialization method that retrieves a list of topics as data topics, a recording duration and a directory for the recorded data from the parameter server. In the record method, the base implementation just sleeps for the given recording duration after setting a flag, that the recording has started. In this way, subclasses that use callbacks for the data topics only have to implement these callbacks and check the recording flag. The base class also contains an empty finish method, that is called after all waypoints have been visited. It can be overwritten by subclasses for example for saving all collected data or perform some post-processing.

In this thesis a subclass is implemented, that receives the point cloud data of the lidar. When the recording flag is set, each received point cloud is transformed to a fixed frame and added to an accumulated cloud. In the finish method, the accumulated point cloud is saved to a file.

6. Evaluation

In this section, the proposed method and implementation is evaluated using different models. The main objectives, which are evaluated using various metrics, are covering as much of the target model as possible in the smallest amount of time.

The experiments were performed on an *Intel Core i7-8565U CPU @ 1.80GHz*.

6.1. Robots

This work was evaluated with two robots, “Asterix” and “Spot” (Figure 6.1).

Asterix is a tracked robot developed in a student project at Team Hector[1]. It is a highly-mobile platform with movable flippers, a manipulator and many sensors, including a 360 degree camera and a rotating Velodyne VLP-16 lidar.

Spot is a legged robot developed by Boston Dynamics[2]. The one used for evaluation is part of the research project “AICO - AI and Robotics in Construction”. In the context of this project, the robot was equipped with additional sensors, including a 360 degree camera and a Velodyne VLP-16 lidar.

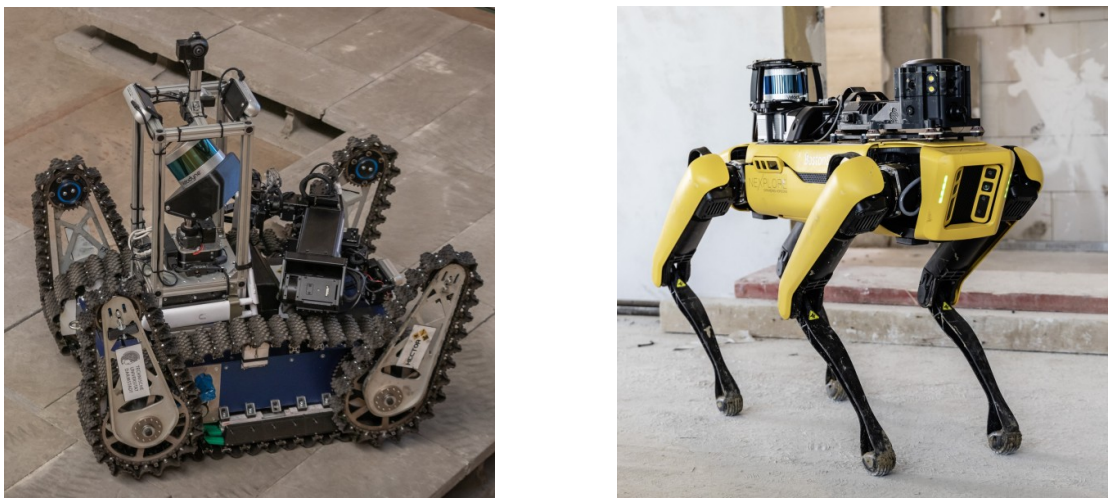


Figure 6.1.: Asterix (left, Photo: Bastian Hirschel) and Spot (right, Photo: Nexplore[3])

[1] https://www.teamhector.de/robots#asterix
[2] https://www.bostondynamics.com/products/spot
[3] https://www.nexplore.com/

6.2. Evaluation models

For the evaluation, two different models have been used. They have both been remeshed using Blender[4], the smaller one with a voxel size of 5 cm, and the larger ones with 10 cm due to memory reasons. Afterwards, the faces were triangulated. The specifications of the models are provided in Table 6.1.

Model	Length $[m]$	Width $[m]$	Height $[m]$	#Vertices	#Faces
Small	12.1	15.1	4.14	481,021	962,114
Large (part)	56.4	24.2	8.42	1,116,373	2,233,836
Large (complete)	56.4	72.4	8.42	3,294,654	6,592,384

Table 6.1.: Model specifications

The first one (Figure 6.2) is a rather small model which was only constructed for this work and is no real building. Nevertheless, the doors have a standardized width of 0,875 m and all corridors have a width of at least 1,2 m. It is constructed to contain difficulties that may also occur in real buildings, for example rooms with interior walls and multiple doors and an unreachable room in the middle. It also contains a ramp in order to evaluate traversable height differences.

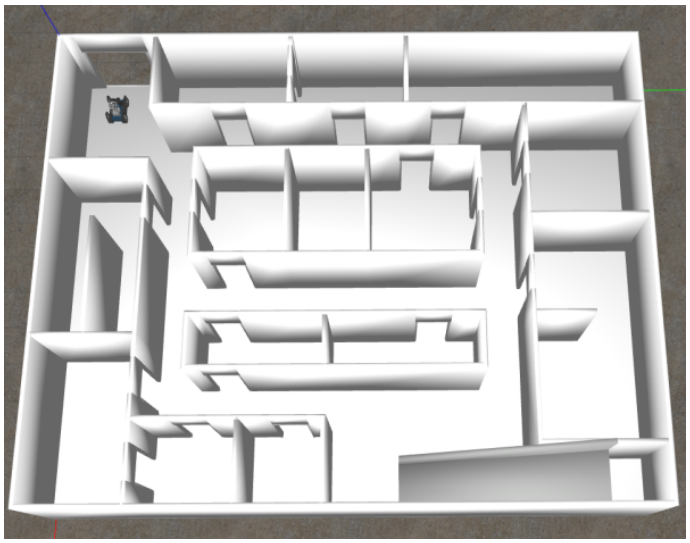


Figure 6.2.: Small model

The second one (Figure 6.3) is a large model of a real construction site. In this evaluation, only about a third of the complete model is used. The reason for this is the large memory consumption of the different model representations during the precomputations with a sufficiently high resolution of the SDF and 3-dimensional occupancy grid. The model has two floors and multiple stairs.

[4]https://www.blender.org/

Figure 6.3.: Large model (complete and part)

6.3. Precomputations

6.3.1. Select set of viewpoints

Coverage and time efficiency are both directly related to the choice of viewpoints, since as few viewpoints as possible should be selected that cover as much as possible. Different approaches for selecting the viewpoints have been presented in Section 4.2.4. The following evaluations were performed on the small model (cf. Section 6.2) with two different target models. The first one is the same as the complete model and the second one is a much smaller part of the small model, shown in Figure 6.4.

First, it was investigated how to choose the minimum reward to ensure good coverage, but still have a low number of viewpoints. Therefore, the greedy approach was executed with a minimum reward of 1, so each point that covers at least one new target point, is selected. In Figure 6.5, the reward of each newly selected viewpoint is shown as well as the number of covered target points. As can be seen, for both target models the reward of the newly selected viewpoints decreases very fast. Hence, the usage of a minimum reward is required. For the following experiments, it was set to 100, since there is a good balance between coverage and number of viewpoints.

Then the different approaches have been compared in terms of number of selected viewpoints, number of uncovered target points and computation time. The results are shown in Table 6.2. The coverage seems to be very low for each of the approaches, but here it must be considered that the target points are not only sampled on the inside of the model but also on the outer walls, below the floor and in the unreachable room in the middle. So there are a lot of target points that can never be covered.

As can be seen in the table, the greedy approach is the fastest one and has pretty good results. The greedy approach without redundancies is sometimes slightly better, for example with the complete model, but takes up to 6 times longer. The two probabilistic approaches require much more time as they are executed repeatedly, in this case 100 times. The one with the exponential distribution can provide better results but as the mean

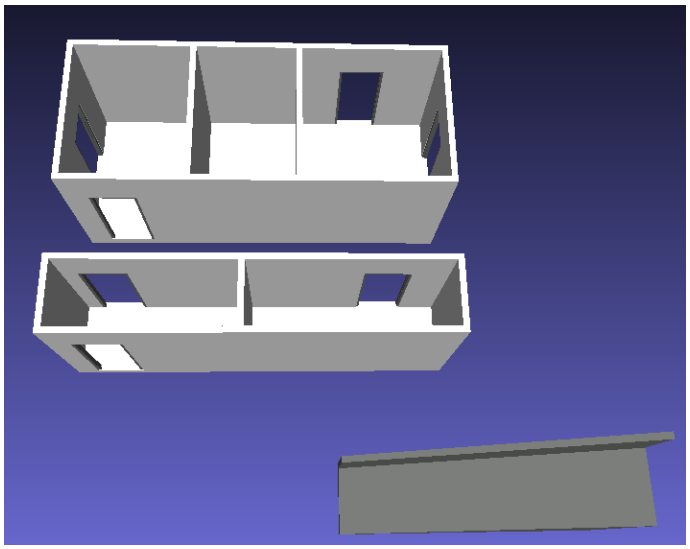


Figure 6.4.: Smaller target model for usage with the small model. It only contains the inner rooms and the ramp.

and the standard deviation indicate, this is not necessarily the case. Additionally, its coverage is slightly worse. The worst one is the probabilistic approach based on rewards. Here, much more viewpoints are selected but the coverage is only a little better. This is, because the rewards of many candidates are very similar. Then the probability, to select a viewpoint with a medium good reward is very high which results in many selected viewpoints as the reward of the remaining candidates decreases slower.

For the following experiments, the greedy approach with a minimum reward of 100 is used.

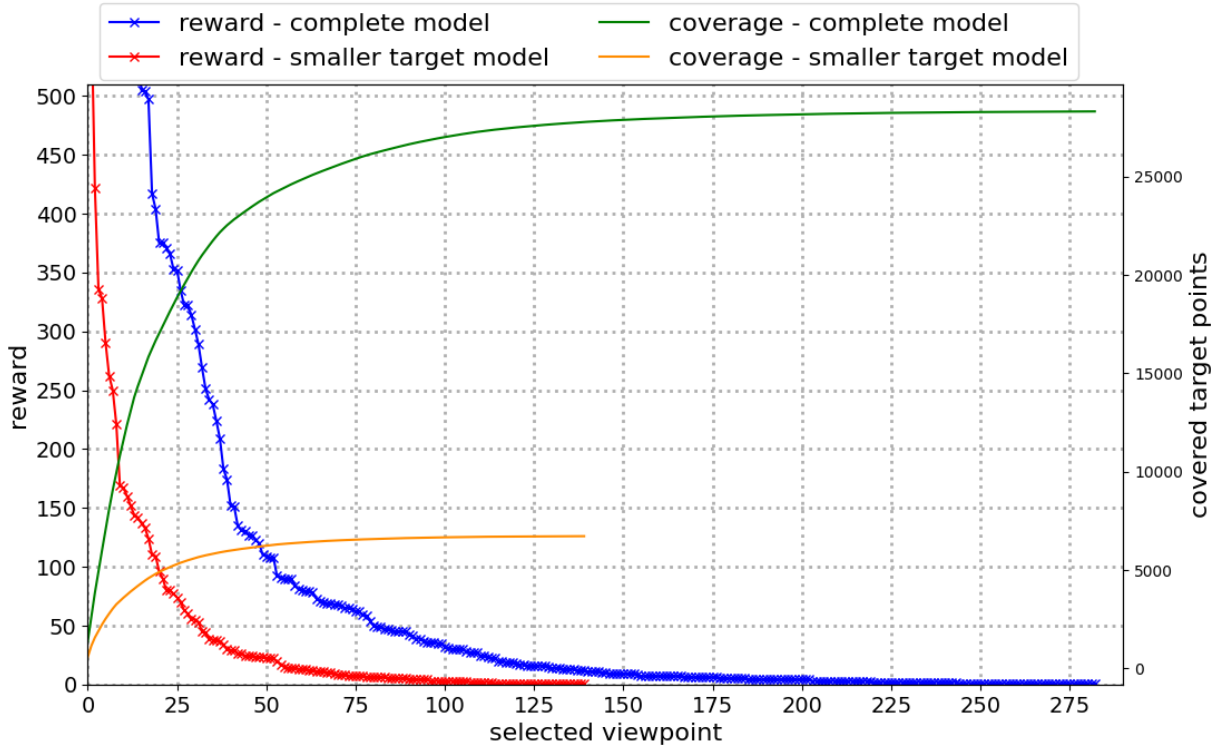


Figure 6.5.: Reward of all viewpoints in order of selection using the greedy approach with a minimum reward of 1

Target model		Total	Greedy	Greedy w/o redundancies	Probabilistic (exp. dist.)	Probabilistic (reward)
Complete model	# Selected Viewpoints	327	53	53	$\mu = 55.44$ $\sigma = 1.44$ best = **52**	$\mu = 74.18$ $\sigma = 2.8$ best = 69
	# Uncovered Targets	114,771	90,612	90,554	$\mu = 90{,}474.44$ $\sigma = 120.73$ best = 90,645	$\mu = 90{,}130.52$ $\sigma = 193.46$ best = **90,182**
	Computation time [µs]		**1,350,373**	9,003,365	146,971,815	208,473,413
Smaller target model	# Selected Viewpoints	327	20	20	$\mu = 20.49$ $\sigma = 1.01$ best = **19**	$\mu = 25.06$ $\sigma = 1.39$ best = 22
	# Uncovered Targets	25,960	**21,153**	**21,153**	$\mu = 21{,}168.48$ $\sigma = 103.25$ best = 21,239	$\mu = 21{,}113.79$ $\sigma = 116.95$ best = 21,380
	Computation time [µs]		**110,994**	468,105	10,140,930	11,763,659

Table 6.2.: Comparison of the selector approaches. The total number of candidates as well as the size of the target set are in the "Total" column. The minimum reward was set to 100 and the maximum number of lost targets to 99. The two probabilistic approaches have been repeated 100 times, μ is the mean and σ the standard deviation. The best was selected as described in Section 5.2.4, the computation time is for 100 repetitions.

6.3.2. TSP solvers

The length of the path containing all selected viewpoints also directly affects the time efficiency. In Section 4.2.5 different solvers for the TSP have been presented. The evaluations were performed on data sets with varying numbers of waypoints and are based on the small model.

For the simulated annealing parameters, initial temperature and cooling rate, several different values have been tested in order to find the best one. The tests were performed 25 times for each setting and on TSP instances with 23, 66 and 149 nodes. Not each parameter combination was performed on each TSP instance, because it became apparent quite quickly, as can also be seen in Figure 6.6, that the various parameters do not have a great impact and the resulting path length is very similar. The mean is shown in the figure, the standard deviation is below 6 for each of them. This shows, that the approach is robust to the parameter settings, the best setting depends on the problem instance.

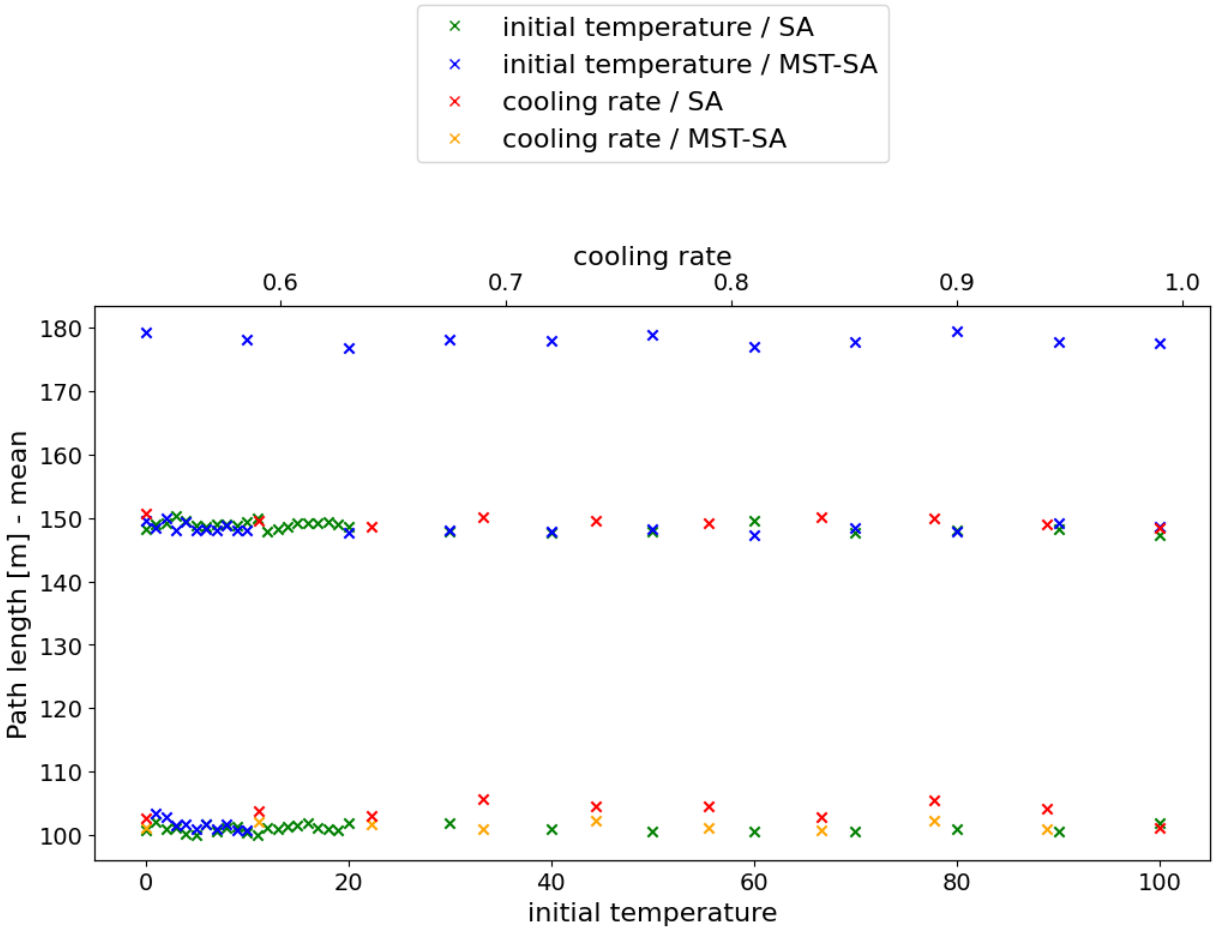


Figure 6.6.: Comparison of different values for initial temperature and cooling rate for SA and MST-SA, performed on TSP instances with 23, 66 and 149 nodes.

The different solvers were compared in terms of path length and computation time. The results are shown in Table 6.3. Four instances of the TSP with various numbers of waypoints have been solved using the presented approaches. But only the smallest one was also executed with the Brute-Force search, here the path has a length of 59,5 m. The results show that for each problem size, the simulated annealing (SA) approaches give

better results than Greedy and MST. The standard deviations also show that they are sufficiently robust. The disadvantage of these approaches, however, is that significantly more computation time is required. The three simulated annealing approaches themselves differ not that much. They have quite similar results and computation times. This means, that with a fixed initial temperature, cooling rate and termination condition, the better initial solutions do not really improve the results of the simulated annealing.

#Way-points		Greedy	MST	SA	MST-SA	Greedy-SA
9	Path [m]	$\mu = 66.20$ $\sigma = 5.95$	66.71	$\mu =$ **59.50** $\sigma = 0.0$	$\mu =$ **59.50** $\sigma = 0.0$	$\mu =$ **59.50** $\sigma = 0.0$
	Time [µs]	$\mu = 494$ $\sigma = 71$	$\mu =$ **482** $\sigma = 90$	$\mu = 1{,}934{,}584$ $\sigma = 51{,}336$	$\mu = 1{,}822{,}710$ $\sigma = 101{,}567$	$\mu = 1{,}991{,}268$ $\sigma = 72{,}887$
23	Path [m]	$\mu = 125.88$ $\sigma = 4.26$	125.14	$\mu = 101.01$ $\sigma = 3.21$	$\mu = 101.49$ $\sigma = 3.49$	$\mu =$ **100.35** $\sigma = 2.31$
	Time [µs]	$\mu =$ **612** $\sigma = 119$	$\mu = 796$ $\sigma = 162$	$\mu = 5{,}484{,}290$ $\sigma = 141{,}015$	$\mu = 5{,}339{,}074$ $\sigma = 191{,}055$	$\mu = 5{,}699{,}300$ $\sigma = 204{,}944$
66	Path [m]	$\mu = 177.34$ $\sigma = 6.83$	174.85	$\mu = 149{,}11$ $\sigma = 2.7$	$\mu = 148.05$ $\sigma = 2.02$	$\mu =$ **147.92** $\sigma = 1.75$
	Time [µs]	$\mu = 2{,}285$ $\sigma = 259$	$\mu =$ **2,096** $\sigma = 390$	$\mu = 22{,}583{,}077$ $\sigma = 1{,}128{,}284$	$\mu = 22{,}238{,}323$ $\sigma = 976{,}110$	$\mu =$22,950,444 $\sigma = 1{,}049{,}590$
149	Path [m]	$\mu = 213.92$ $\sigma = 4.16$	215.92	$\mu =$ **177.43** $\sigma = 3.3$	$\mu = 178.16$ $\sigma = 3.91$	$\mu = 177.48$ $\sigma = 4.49$
	Time [µs]	$\mu =$ **10,228** $\sigma = 593$	$\mu = 17{,}660$ $\sigma = 1{,}494$	$\mu = 60{,}262{,}204$ $\sigma = 3{,}174{,}638$	$\mu = 60{,}598{,}970$ $\sigma = 3{,}322{,}087$	$\mu = 60{,}912{,}301$ $\sigma = 1{,}951{,}424$

Table 6.3.: Comparison of the TSP solvers. The “Path” rows contain the path length, the “Time” rows the computation time. Simulated annealing uses always the best mutator, an initial temperature of 5 and a cooling rate of 0.99. It performs 10^5 iterations. All solvers have been executed 25 times. μ is the mean value, σ is the standard deviation.

6.3.3. Results

The final results of the precomputations, i.e. the selected viewpoints and the complete path, are evaluated on both models with two target models each.

First, the results of the planning on the small model with the complete model as target model using Asterix can be seen in Figure 6.7. For the planning, the parameters as described in Appendix A.1 were used. The resulting path contains 62 waypoints and is 155,9 m. As can be seen, each reachable area contains viewpoints and the path has no redundancies.

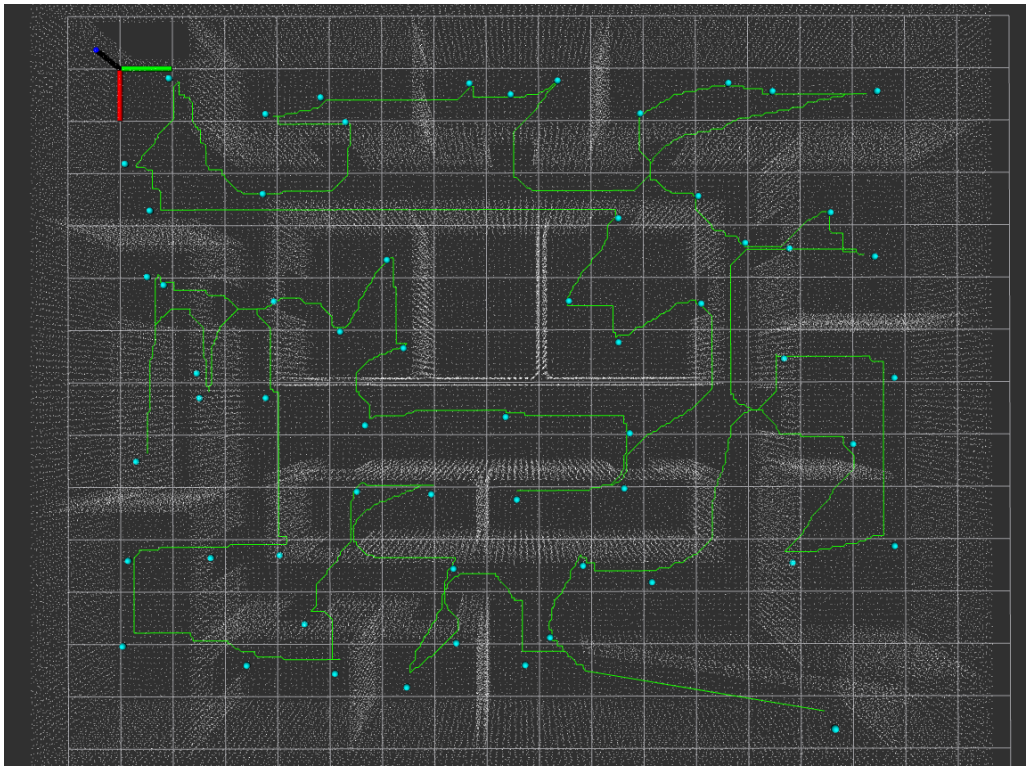


Figure 6.7.: Path planned for Asterix on small model with complete model as target model (white point cloud). The dots are the selected viewpoints, the green line is the planned path.

Figure 6.8 shows the planned path on the small model with the smaller target. The image does not show all selected viewpoints due to visualization problems when the points are published too fast. It becomes clear, that the selection of viewpoints is focused on the target model. As a result, the path is much shorter. It has a length of 85,75 m and contains 25 waypoints.

The next experiments were performed on the large model using Spot. First, the precomputations were executed with the parameters as described in Appendix A.1. Here, the costs on the stairs are too high, so that only candidates in the upper level are generated. The results can be seen in Figure 6.9 and 6.10. The first image shows the planned path for the complete model as target model. Here, 111 viewpoints have been selected and the path is 456,3 m long. The second image shows the usage of the smaller target model. The path has 34 waypoints and a length of 137,35 m. In order to test the path planning with multiple levels and height differences, the radius of the height differences layer of the mesh map was decreased to 0.15. With this, the stairs could be passed and the path was planned using both levels (Figure 6.10b). It has 84 waypoints

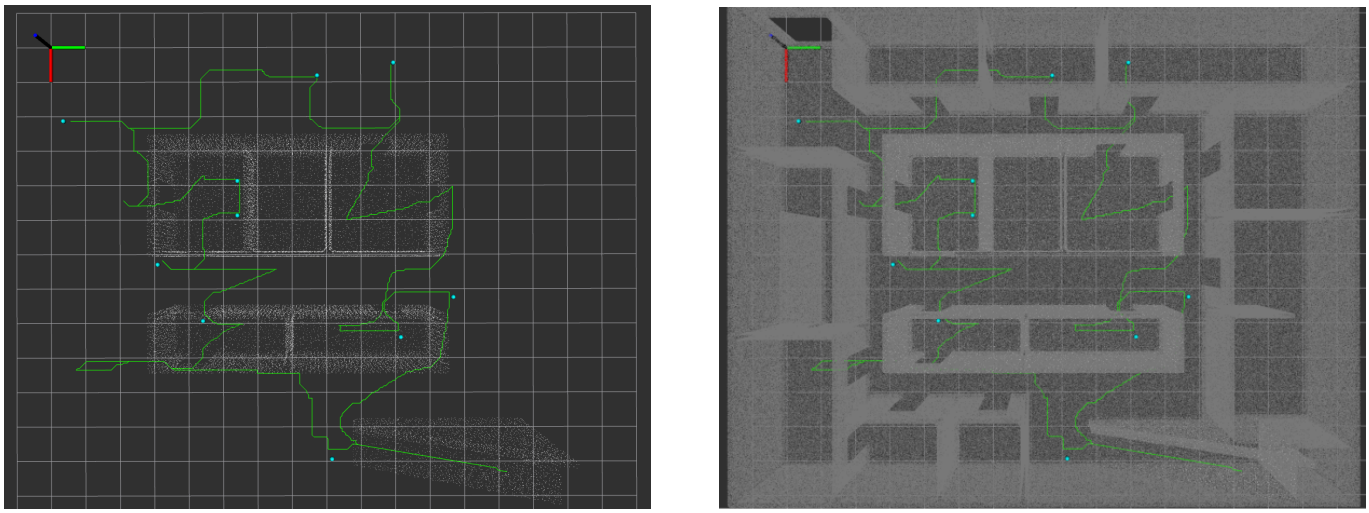


Figure 6.8.: Path planned for Asterix on small model with the smaller target model. The dots are the selected viewpoints, the green line is the planned path. The left image shows the target model, the right image the complete model.

and is 520,77 m long. There is one selected viewpoint in the right part of the model, which has been cropped in the image for better visibility.

These experiments show, that the developed path planner works on different problem sizes with different target models and can also plan a path on multiple levels.

6.3.4. Computation time

The computation time required for the precomputations is highly dependent on the used models as well as on chosen parameters. For example, the voxel size of the model representations, the number of target points that are sampled and the used selector and TSP solver impact the computation time.

For the small model with the complete model as target and with the parameters specified in Appendix A.1, the precomputations required 260,2 s. In this case, the models have been generated from the mesh. When loading the model representations, that have been computed earlier, this time reduces to 220,1 s.

Another way to reduce the required time is the parallelization of the visibility checks using OpenMP[5]. Without parallelization, the visibility checks for all 327 candidates require 44,4 s. With the parallelization, this reduces to 16,0 s.

[5] `https://www.openmp.org/`

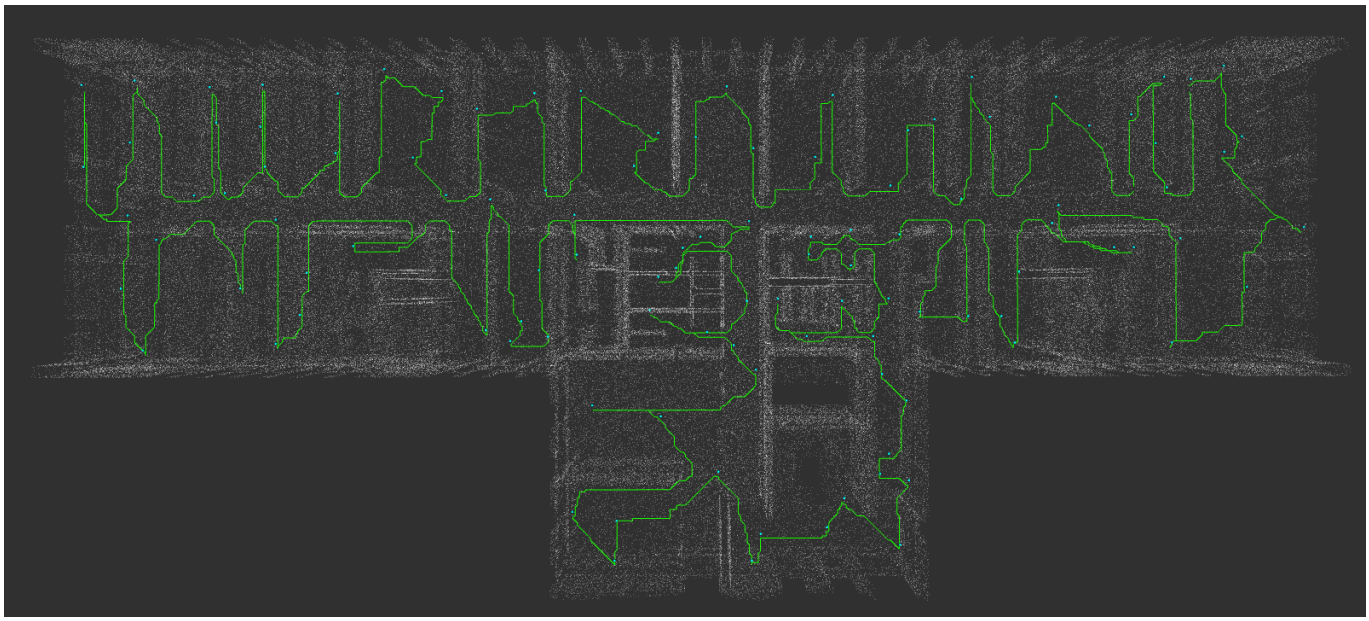
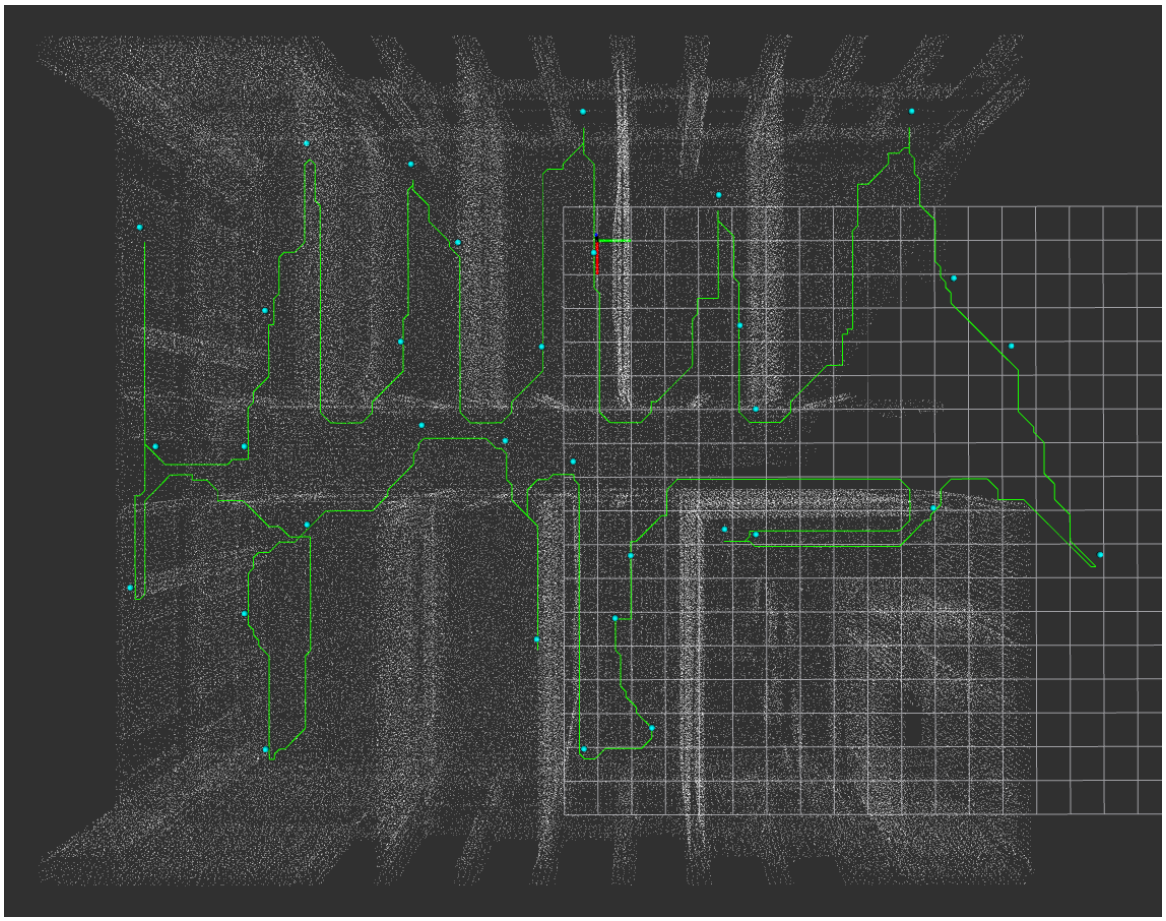
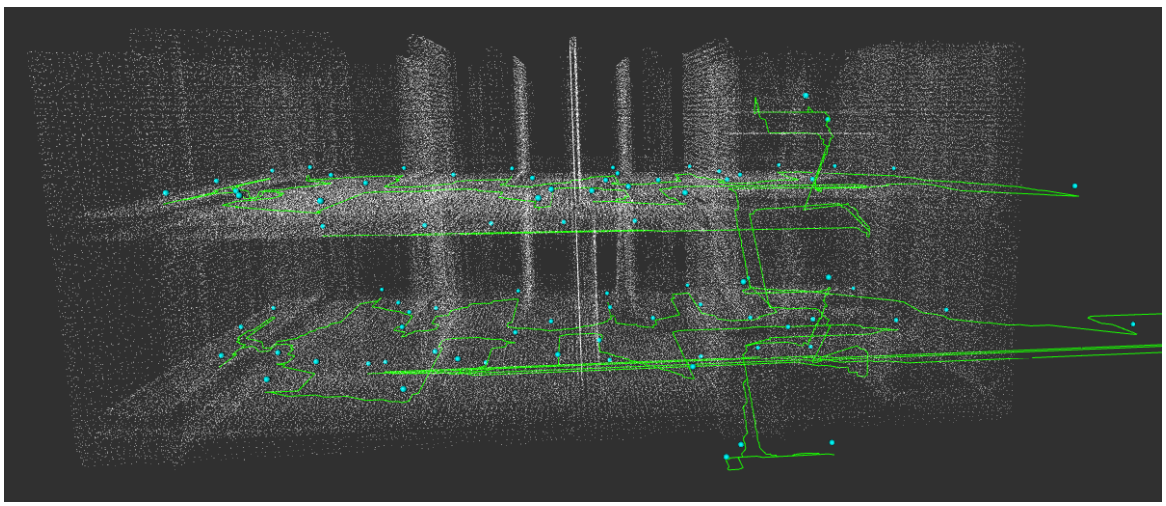


Figure 6.9.: Path planned for Spot on large model with complete model as target model (white point cloud). The dots are the selected viewpoints, the green line is the planned path.



(a) Single level



(b) Multi level with connection via stairs

Figure 6.10.: Path planned for Spot on large model with smaller target model (white point cloud). The dots are the selected viewpoints, the green line is the planned path.

6.4. Execution of path

When executing the planned path, several problems occurred. The path and the viewpoints were all planned in the building frame, but the execution happens in the world frame. Therefore, the transformation between the building frame and the world frame is required. In simulation, this is a static transformation since the model and the robot are always spawned at the same position. But in real world, this is not possible.

Another problem is, that the path or viewpoints sometimes are close to the wall. The robot can reach this pose, if it has the right orientation or arrives from the right direction. But since the vehicle controller often only approaches a given waypoint and rotates at the end to the desired orientation, the waypoint sometimes cannot be accessed. However, with the currently used path planner, the paths cannot be planned further away from the walls, because then the doors would also be counted as not traversable, since here the distance to the door frame on both sides would be smaller than required. This stems from the fact, that the currently used planner approximates the robot using a circular shape by inflating all obstacles. The current solution is, as described in Section 5.3.1, the updating of blocked paths when the blocked pose is not at the viewpoints, and the stuck behavior for both, when the robot is stuck in the middle of the path but also when it cannot approach the viewpoint. But the stuck behavior only takes effect when the robot is already in contact with the obstacles, which is not great on construction sites and in buildings.

6.4.1. Simulation

The execution of the path was tested in simulation using Gazebo[6], an open-source robotics simulator which is integrated with ROS. As described above, a static transformation between the world and the building was used.

For the planning of the paths, the parameters as described in Section A.1 were used except the cost threshold for the candidate generation, which was set to 0.1. The reason for this is the used grid map planner (cf. Section 5.3.1), which is a 2D planner and hence has problems with the ramp in the small model. With the adapted cost threshold, no candidates are generated on the ramp.

The first image (Figure 6.11) shows the path update. The light green path is the original planned path which is too close to the wall with the used robot length and width. Hence, the path is updated (blue line). Here it is also shown that the updated path contains only the poses in front of the robot. The orange path is the smoothed one provided by the vehicle controller, which the robot follows.

[6] https://classic.gazebosim.org/

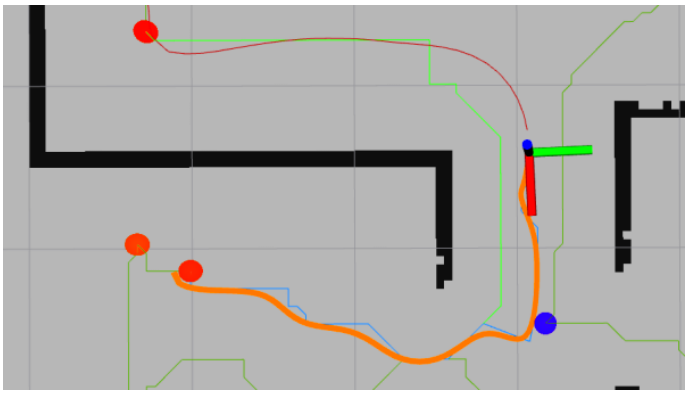


Figure 6.11.: Updated path during execution. The axes show the current position of the robot. The light green line is the original planned path, the blue one is the updated path. The thick orange line is the smoothed one. The dark red line is the trajectory already driven.

The Figures 6.12 and 6.13 show the results of the path execution using Asterix on the small model with the complete model used as target model. The first image shows the planned and the actually driven path as well as the waypoints and the 2D grid map. Here it can be seen, that the robot could follow the path and reach nearly all waypoints. One waypoint was skipped, as it was too close to a door frame, which can be seen in more detail in Figure 6.14. The same image also shows the behavior of the stuck recovery, where the robot drives backwards a short distance. Figure 6.13 shows the resulting point clouds. In the first image, the point clouds recorded at each waypoint have different colors, in order to show which viewpoint covers which area. This visualization is also very helpful for debugging. The second image contains the same cloud but without colors which allows for a better overview of the coverage. The density of the resulting cloud is different, as it contains the raw data of the scans at the waypoints, which are more dense closer to the sensor, and no filtering has been performed. The coverage of the model is very high, there are only a few areas left uncovered. One reason for some of the uncovered areas is, that the robot did not always align correctly with the desired orientation. Figure 6.15 shows such a case, where the desired orientation (green arrow) and the actual orientation (red arrow) are shown. Another reason for uncovered areas are the narrow aisles and rooms of the small model, for example, in Figure 6.13 on the left and the top edge. Here, there is not enough space to place the robot far enough away from the wall so that these areas can be seen.

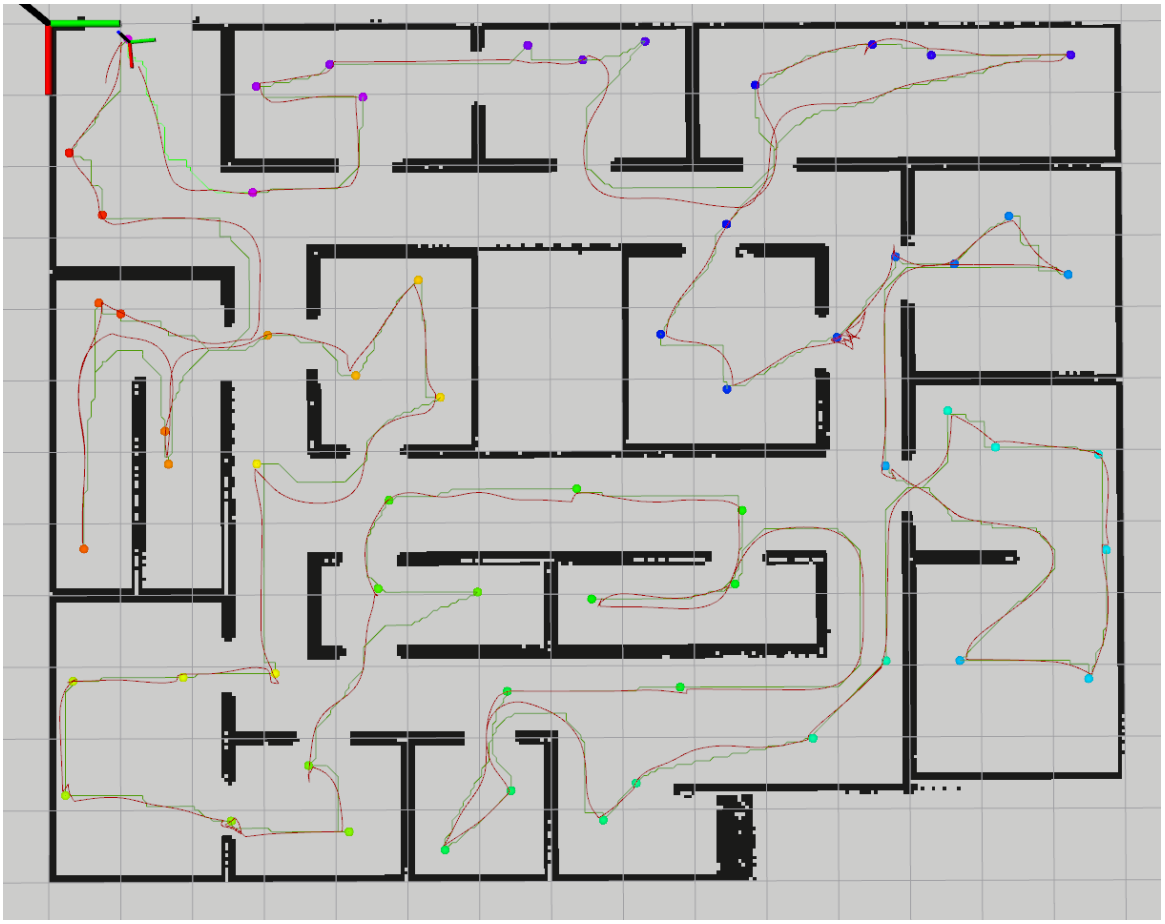


Figure 6.12.: Finished execution of path with Asterix on small model for complete model as target model. The image contains the 2D occupancy grid map, the waypoints, the planned path (green), the driven trajectory (dark red) and the current robot pose as small axes in the upper left.

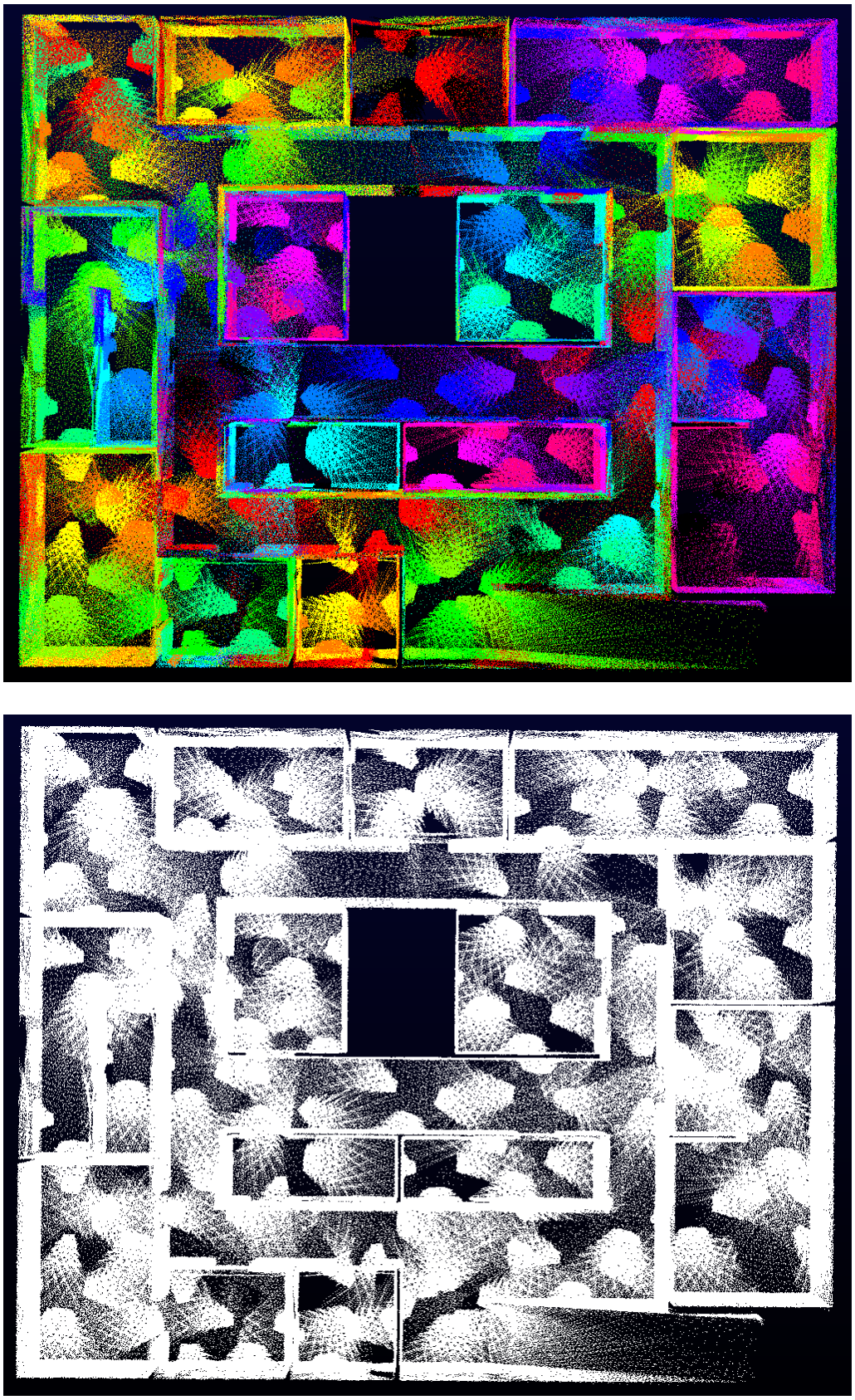


Figure 6.13.: The resulting model of the execution shown in Figure 6.12. In the upper image, the data recorded at each waypoint has different colors. The lower one is the same without colors.

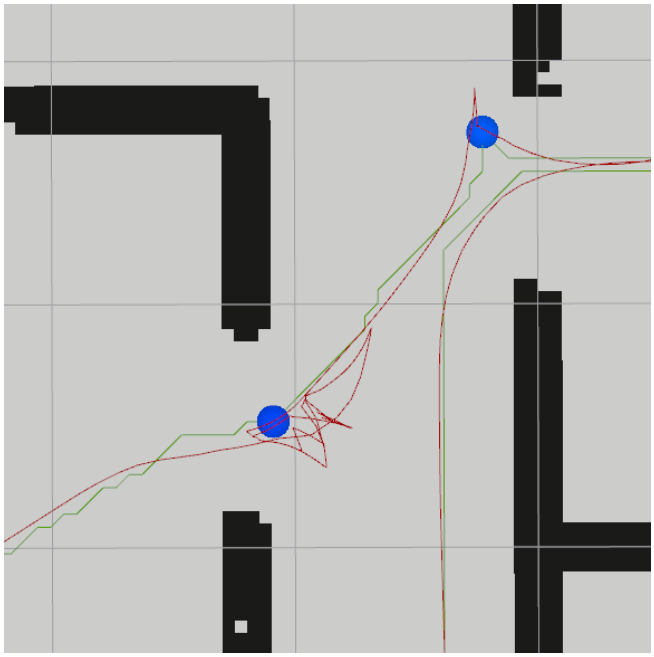


Figure 6.14.: Stuck recovery.
Upper right point: recovery successful.
Lower left point: several tries, none successful, point skipped.

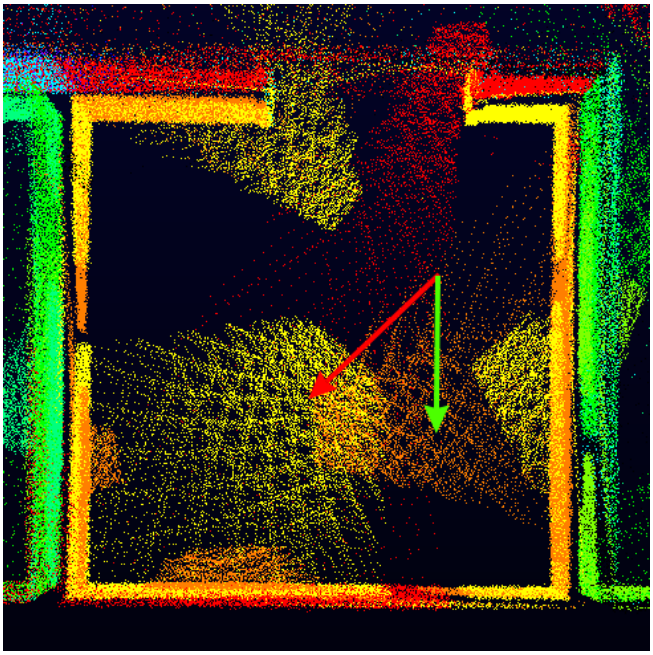


Figure 6.15.: Uncovered areas due to wrong orientation. The robot should have aligned with the green arrow, but had an offset indicated by the red arrow. Hence, the yellow points do not cover the expected area.

Comparison with Exploration

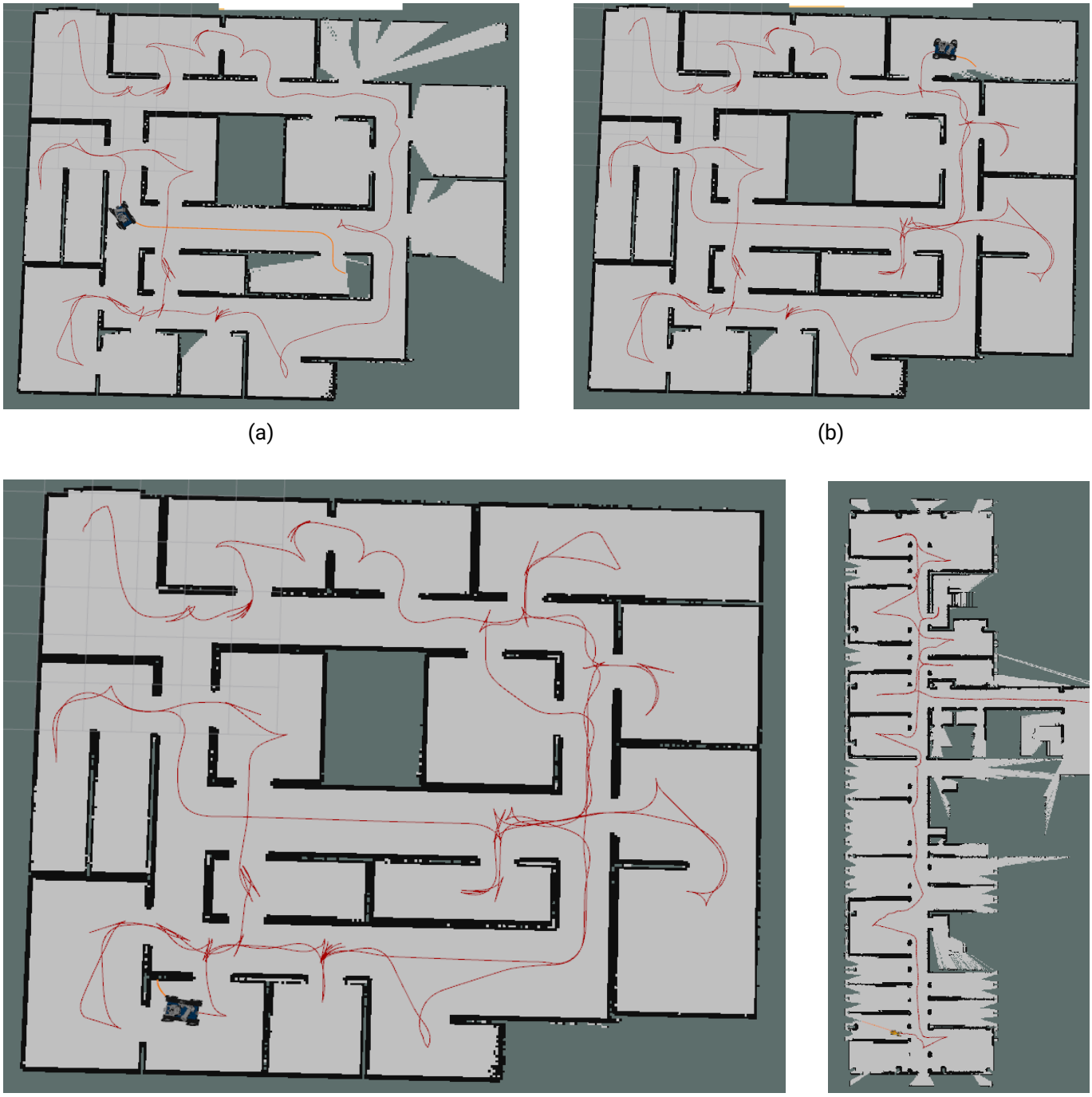
Exploration algorithms try to explore and map an unknown environment with a high coverage. Therefore, the developed planner is compared with an existing exploration algorithm implementation (cf. Section 3.4.1).

The path of the exploration with Asterix on the small model can be seen in Figure 6.16 (a) - (c). The first two images show the progress, the third one the finished exploration. Here, the used greedy approach in the exploration can be seen clearly, as there are many small uncovered areas remaining where the robot does not expect to see a large amount of new data and therefore does not explore them immediately. This leads to path redundancies. However, for the small model, the path does not contain too many redundancies. Quite in contrast to the exploration in the large model, performed with Spot. Figure 6.16d only shows the first part of the exploration, but it already shows that here are many unexplored corners left which need to be explored later. Additionally, here the areas and distances are significantly greater which leads to problems during the exploration since then sometimes the computation of the next exploration goal and the path to it takes too much time and the next planning cycle is started before the robot could move towards its current goal.

Another problem is the stuck detection and recovery during the exploration. As can be seen in the images, the robot has been stuck more often than during the execution of the path shown in Figure 6.12.

A point cloud is recorded and accumulated continuously during exploration. In order to make the result of the developed planner comparable to it, a point cloud was also recorded continuously instead of only at the viewpoints. The results are shown in Figure 6.17. It can be seen that the point cloud of exploration does have several uncovered areas whereas the accumulated point cloud of the path of the developed planner is nearly complete.

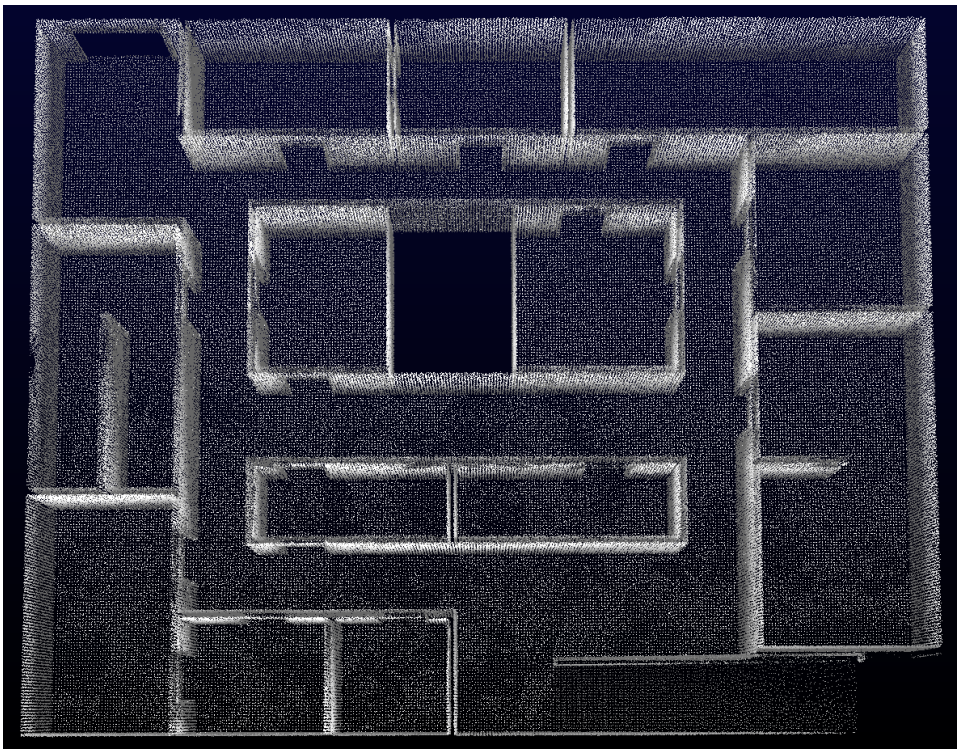
The problem with the accumulated point clouds occur during the execution on the real robot. In simulation, the localization is exact, hence, there is no noise in the clouds. But in the real world, the quality of the localization directly affects the quality of the accumulated point cloud. The proposed method to only record the data at the viewpoints offers the possibility to reduce the noise, e.g. by combining the localization and the desired viewpoint position or apply some kind of post processing that corrects the position of the recorded point clouds afterwards.



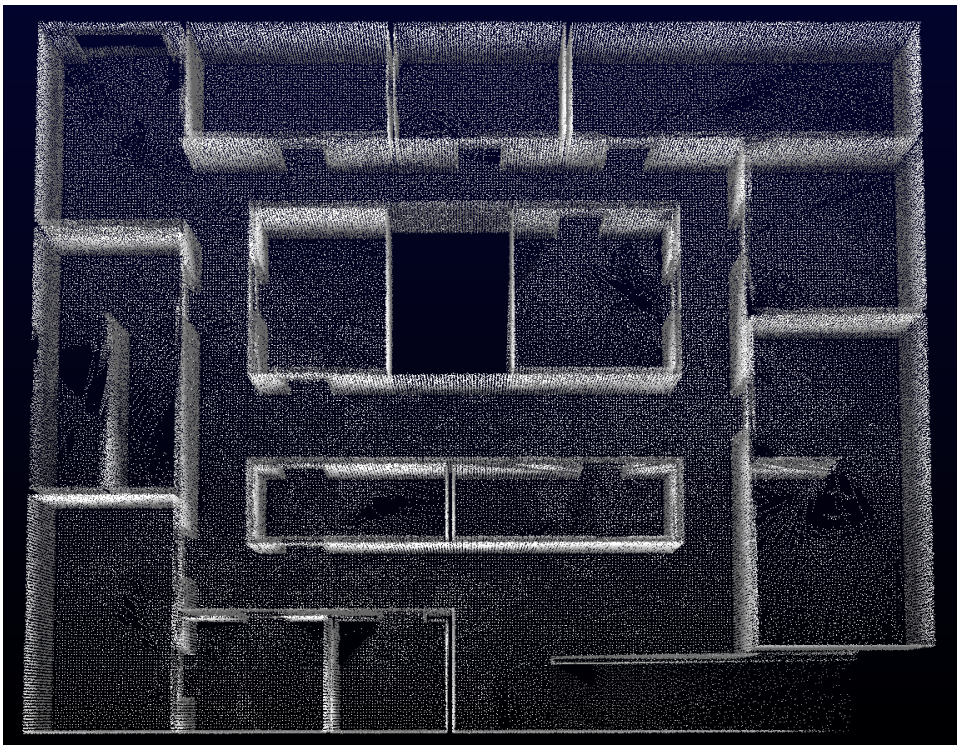
(a) (b)

(c) (d)

Figure 6.16.: (a)-(c): Exploration with Asterix on the small model.
(d): Exploration with Spot on large model.
All images contain the 2D occupancy grid map, the driven trajectory (dark red), the current robot position and the path to the next exploration goal (orange).



(a) Result of execution with the developed planner, shown in Fig. 6.12



(b) Result of exploration, shown in Fig. 6.16c

Figure 6.17.: Accumulated point clouds.

7. Conclusion and Future Work

7.1. Conclusion

In this work, a new 3D coverage path planning approach was proposed for efficient construction progress monitoring. A path is planned based on a selection of viewpoints retrieved from a set of rated candidates. This is done in order to plan a path that combines both, providing a high coverage of a given environment while still being time efficient. In order to achieve this, a highly modular path planner was developed using a plugin system. This provides the flexibility to easily adapt to many environments, robots and sensors while still providing a fully functional base.

It could be shown that the planning part worked well on both small and larger models including multiple levels and complex structures and it has also complied with different target models. In the evaluation, the various implemented selectors and TSP solvers were compared and the best ones could be identified. However, it became apparent that different approaches could better meet diverse requirements, e.g. faster computation of the path or better result.

The quality of the planned path could be shown in the evaluation of the path execution. Here it was found, that the robot is able to follow the planned path and update it when needed. The resulting point clouds illustrated, that the requirement of high coverage could be met with the selected viewpoints. Although executing the path worked for the most part, there are still problems that could be solved in the future like colliding with walls and doors during maneuvering, which will be discussed further in the next section.

7.2. Future work

Although the base idea introduced in the introduction could be solved, several possible extensions and problems became apparent while working on implementation and evaluation. The first part of possible future work is improving the planning.

One possibility to optimize the path in one instead of in two steps, the viewpoint selection and the waypoint order computation, was proposed by Cao et al. [8]. Here, the selection and the waypoint order computation are probabilistic and the combination of both is executed repeatedly.

In the preparations for the TSP, the costs are computed. Here, only the path length is used as costs which can sometimes be inaccurate. For example, a longer path can be faster if it is in a straight line, whereas a path with many curves might require more time even it is shorter. Also the costs on stairs or ramps might be different than on a flat surface. So here another way of cost computation can be useful, depending on the environment.

For the planning, an option to set a minimum distance to the next obstacle would be helpful. In this way, a safety distance could be applied for sensible areas. For example, when the construction site is at an early stage, without such a distance the robot might fall down somewhere due to potentially missing windows, doors or stairs. Even without such risks it is preferred that the robot does not come in contact with walls or objects to reduce the chance of damage to the robot and environment. This also correlates with the path planner used to compute the paths between the waypoints. The current one has the problem, that the paths are often very close to the walls. But the inflation of obstacles cannot be increased further since then doors would be marked as blocked. This stems from the fact that the robot is approximated as a circle and the inflation of obstacles happens independently from the robot's orientation.

In terms of execution, it would be more reasonable to assume the previously planned path between waypoints as an initial path for a planner that can also handle dynamic obstacles, rather than thinking of it as a final path that is only updated when it is already blocked.

Another important improvement for the execution refers to the viewpoints and not to the path between them. On construction sites there are often obstacles, for example machinery or construction materials. These obstacles can block a viewpoint or limit the view from them. In these cases, replacement viewpoints that can cover the missing target points need to be planned and integrated into the path.

A point that is required to perform the execution in a real world scenario, is the transformation between the building and world frame as well as the localization inside of the building. With these and an improved path planner, the execution can be tested on the real robot.

Less of a direct improvement but more of an extension for other applications could be integrating exploration into the planner. This would take effect if an unknown area is detected during the execution of the planned path, e.g. in disaster scenarios, for example when a building partially collapses or is inaccessible due to other reasons but a 3D model is available.

Bibliography

[1] Benjamin Keinert et al. “Spherical Fibonacci Mapping”. In: *ACM Trans. Graph.* 34.6 (Oct. 2015). ISSN: 0730-0301. DOI: 10.1145/2816795.2818131.

[2] Álvaro González. “Measurement of areas on a sphere using Fibonacci and latitude–longitude lattices”. In: *Mathematical Geosciences* 42.1 (2010), pp. 49–64.

[3] Thomas H Cormen et al. *Introduction to algorithms, Third Edition*. MIT press, 2009.

[4] H. Moravec and A. Elfes. “High resolution maps from wide angle sonar”. In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. 1985, pp. 116–121. DOI: 10.1109/ROBOT.1985.1087316.

[5] Helen Oleynikova 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*. 2016.

[6] Sam Cunningham-Nelson et al. “Coverage-based next best view selection”. In: *Proceedings of the Australasian Conference on Robotics and Automation 2015*. Ed. by H Li and J Kim. Australia: Australian Robotics and Automation Association, 2015, pp. 1–9. URL: https://eprints.qut.edu.au/91030/.

[7] Jonathan Daudelin and Mark Campbell. “An Adaptable, Probabilistic, Next-Best View Algorithm for Reconstruction of Unknown 3-D Objects”. In: *IEEE Robotics and Automation Letters* 2.3 (2017), pp. 1540–1547. DOI: 10.1109/LRA.2017.2660769.

[8] Chao Cao et al. “TARE: A Hierarchical Framework for Efficiently Exploring Complex 3D Environments”. In: *Proceedings of Robotics: Science and Systems (RSS ’21)*. July 2021.

[9] Ludvig Ericson, Daniel Duberg, and Patric Jensfelt. “Understanding Greediness in Map-Predictive Exploration Planning”. In: *2021 European Conference on Mobile Robots (ECMR)*. 2021, pp. 1–7. DOI: 10.1109/ECMR50962.2021.9568793.

[10] Marco Steinbrink et al. “Rapidly-Exploring Random Graph Next-Best View Exploration for Ground Vehicles”. In: *2021 European Conference on Mobile Robots (ECMR)*. 2021, pp. 1–7. DOI: 10.1109/ECMR50962.2021.9568785.

[11] Meida Chen et al. “Proactive 2D model-based scan planning for existing buildings”. In: *Automation in Construction* 93 (2018), pp. 165–177. DOI: https://doi.org/10.1016/j.autcon.2018.05.010. URL: https://www.sciencedirect.com/science/article/pii/S0926580517310385.

[12] Stephan Wirth and Johannes Pellenz. “Exploration Transform: A stable exploring algorithm for robots in rescue environments”. In: *2007 IEEE International Workshop on Safety, Security and Rescue Robotics*. 2007, pp. 1–5. DOI: 10.1109/SSRR.2007.4381274.

[13] Markus Sigg. “Combined Observation Planning for mobile platforms and manipulators”. MA thesis. Technische Universität Darmstadt, Department of Computer Science (SIM), 2016.

[14] Aljoscha Schmidt. “Environment-aware Online Inspection Pose Generation for Mobile Robots”. B.S. thesis. Technische Universität Darmstadt, Department of Computer Science (SIM), 2021.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[16] Xiutang Geng et al. “Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search”. In: *Applied Soft Computing* 11.4 (2011), pp. 3680–3689. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2011.01.039. URL: https://www.sciencedirect.com/science/article/pii/S1568494611000573.

[17] Lijin Wang et al. “Enhanced List-Based Simulated Annealing Algorithm for Large-Scale Traveling Salesman Problem”. In: *IEEE Access* 7 (2019), pp. 144366–144380. DOI: 10.1109/ACCESS.2019.2945570.

[18] Péter Fankhauser and Marco Hutter. “A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation”. In: vol. 625. Jan. 2016. ISBN: 978-3-319-26052-5. DOI: 10.1007/978-3-319-26054-9_5.

[19] Armin Hornung 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.

[20] Sebastian Pütz et al. “Continuous Shortest Path Vector Field Navigation on 3D Triangular Meshes for Mobile Robots”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. Software available at https://github.com/uos/mesh_navigation. 2021. DOI: 10.1109/ICRA48506.2021.9560981. URL: https://github.com/uos/mesh_navigation.

[21] Sebastian Pütz, Jorge Santos Simón, and Joachim Hertzberg. “Move Base Flex: A Highly Flexible Navigation Framework for Mobile Robots”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Software available at https://github.com/magazino/move_base_flex. Oct. 2018. URL: https://github.com/magazino/move_base_flex.

[22] Philipp Schillinger, Stefan Kohlbrecher, and Oskar von Stryk. “Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics”. In: *IEEE International Conference on Robotics and Automation*. Stockholm, Sweden, May 2016.

[23] Alessandro Muntoni and Paolo Cignoni. *PyMeshLab*. Jan. 2021. DOI: 10.5281/zenodo.4438750.

[24] *PyMeshLab*. URL: https://pymeshlab.readthedocs.io/en/2021.10/ (visited on 05/10/2022).

[25] *Robot body filter*. URL: http://wiki.ros.org/robot_body_filter (visited on 04/03/2022).

[26] *The Boost Graph Library (BGL)*. URL: https://www.boost.org/doc/libs/1_77_0/libs/graph/doc/index.html (visited on 10/21/2021).

[27] Peter Lepej et al. “Dynamic Arc Fitting Path Follower for Skid-Steered Mobile Robots”. In: *International Journal of Advanced Robotic Systems* 12.10 (2015), p. 139. DOI: 10.5772/61199. URL: https://doi.org/10.5772/61199.

[28] Jonas Springer. *Model-based Path Following*. Technische Universität Darmstadt, Department of Computer Science (SIM), Dec. 2020.

A. Appendix

List of Figures

List of Tables

Acronyms

.obj object file.

.ply Polygon File Format.

BGL Boost Graph Library [26].

BIM building information modeling.

DAF Dynamic Arc Fitting.

ESDF Euclidean signed distance field.

HDF5 Hierarchical Data Format.

MST minimum spanning tree.

NBV next best view.

PCL Point Cloud Library.

POI point of interest.

ROS Robot Operating System.

SA simulated annealing.

SDF signed distance field.

TSDF truncated signed distance field.

TSP Traveling Salesman Problem.

A.1. Evaluation parameters

The parameters used in the evaluation are the following, unless otherwise stated.

General parameters:

Parameter	Value
field of view: horizontal	[-180°, 180°]
field of view: vertical	[-60°,60°]
sensor range	[0.45, 10.0]
cost threshold for candidate generation	0.5
# generated candidate orientations	1
percentage of candidates to keep	0.01 = 1%
viewpoint selector	Greedy
minimum reward	100
cost threshold for path planning	0.5
path planner	Dijkstra mesh planner
waypoint order computer	MST and SA (MST-SA)
Simulated Annealing: use best mutator	true
Simulated Annealing: initial temperature	5
Simulated Annealing: cooling rate	0.99

Model specific parameters:

Parameter	Value (small model)	Value (large model)
Voxel size	0,075 m	0,075 m
TSDF or ESDF	TSDF	TSDF
SDF voxel size	0.0375	0.05
mesh map: height diff layer: threshold	0.2	0.2
mesh map: height diff layer: radius	0.2	0.2
mesh map: inflation layer: inflation radius	0.35	0.35
mesh map: inflation layer: inscribed radius	0.35	0.35
mesh map: inflation layer: inscribed value	1.0	1.0
mesh map: inflation layer: lethal value	2.0	2.0

Robot specific parameters:

Parameter	Value (Asterix)	Value (Spot)
Length	1.0	1.1
Width	0.8	0.5