Linköping University | Department of Computer and Information Science

Master's thesis, 30 ECTS | Elektroteknik

2021 | LIU-IDA/LITH-EX-A--21/082--SE

Coverage Path Planning in Large-scale Multi-floor Urban Environments

– with Applications to Autonomous Road Sweeping

Körvägsplanering i storskaliga och flervåniga stadsmiljöer med tillämpningar mot autonom robotsopning

Daniel Engelsons

Supervisor : Mattias Tiger
Examiner : Fredrik Heintz

External supervisor : Erik Örjehag

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Autonomous lawn mowers and floor cleaning robots are today easily accessible and are utilizing well-studied Coverage Path Planning algorithms. They operate in single-floor environments that are small with simple geometry compared to general urban environments such as city parking garages, highway bridges or city crossings. A next step for autonomous cleaning is road sweeping of these complex urban environments. In this work, a new Coverage Path Planning approach, *Sampled BA* & Inward Spiral* , handling this task was compared with existing well-performing algorithms *BA** and *Inward Spiral*. The proposed approach combines the strengths of existing algorithms and demonstrates state-of-the-art performance on three large-scale 3D environments. It generated paths with less rotation, while keeping the length of the path on the same level. For a given starting point, the new approach had consistently lower cost (length + rotation) for all environments. For random starting points, randomness in the new approach caused less robustness, giving significantly higher cost. To improve the performance of the algorithms and remove bias from manual tuning, the parameters were automatically tuned using Bayesian Optimization. This makes the evaluation more robust and the results stronger.

Acknowledgments

I would like to thank my supervisor Mattias Tiger for his support, engagement and pushing during this project. Big thank you to Erik Örjehag and Fredrik Löfgren at Dyno Robotics for their help and giving me this opportunity. Thank you to my examiner Dr. Fredrik Heintz for his trust and inputs. I would also like to give a thank you to my opponent Jacob Ljungren for a good opposition during the final thesis presentation and to all my friends who were there supporting me.

I will forever be grateful to my family, friends, lecturers and classmates for making my years at Linköping University such an amazing time.

Linköping, December 2021
Daniel Engelsons

Contents

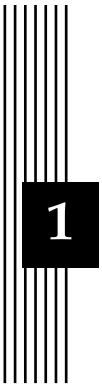
List of Figures

List of Tables

Notations

Notations used in this thesis:

Notation	Description
M	2D-grid with square cells
E	Digital Elevation Model. A 2D-grid with square cells, where every cell has an elevation height.
$\mathcal{W}$	Point cloud
s	3D position in $\mathbb{R}^3$
p	Point in a point cloud $\mathcal{W}$
c	Square cell in a 2D-grid M
N	Number
r	Radius
d	Distance
λ	Step size
h	Height. z-value of a position s
e	Elevation height of a cell $c \in E$
P	Path. A sequential list of points p
C	Coverage rate between 0 and 1
E	Exploration rate between 0 and 1
R	Geometry representing the robot, $R \subset \mathbb{R}^3$



1 Introduction

In this chapter, background and the goal of this project will be presented.

1.1 Motivation

For years, robotics has been an active topic in lab environments and is now entering real human environments by helping humans with dangerous, repetitive and boring labour. A large part of the processes in industries are today executed by robots having higher speed and precision than humans [27]. Autonomous robotics has more recently also entered domestic lives where daily tasks are relieved by easily accessible robots such as with vacuum cleaners and lawn mowers. A natural next step is to evolve the developed robotics technology to tackle more complex tasks in the urban environment such as public streets and parking garages. Efficiently handling large environments with multiple floors remains a challenge for the industry to solve.

A company in Linköping, Dyno Robotics, is currently developing an autonomous sweeping robot. The aim of the robot is to clean streets, parking garages, parks and other outdoor environments. An autonomous robot includes many complex technological solutions, yet in its core lies the path planning. Given data about the environment from the sensors the robot should be able to plan a path that covers all accessible areas in the environment. Making a good plan that covers the whole area is important. With a well performing path planner, the cleaning robot will minimize occurrences where spots are revisited in favor of efficient operation, both in aspect of time and cost. A good plan is short and has few rotations, since rotation leads to degraded cleaning [4].

Today's path planning algorithm of Dyno Robotics sweeping robot is assuming flat terrain. Since this is not the case for many of its intended environments they need an effective path planning solution that can handle complicated environments with complex surface geometry and where it is hard to represent the environment on a 2D plane.

The problem of planning a path that covers a region of interest while avoiding obstacles is called the Coverage Path Planning (CPP) problem [14]. There are two different types of CPP.

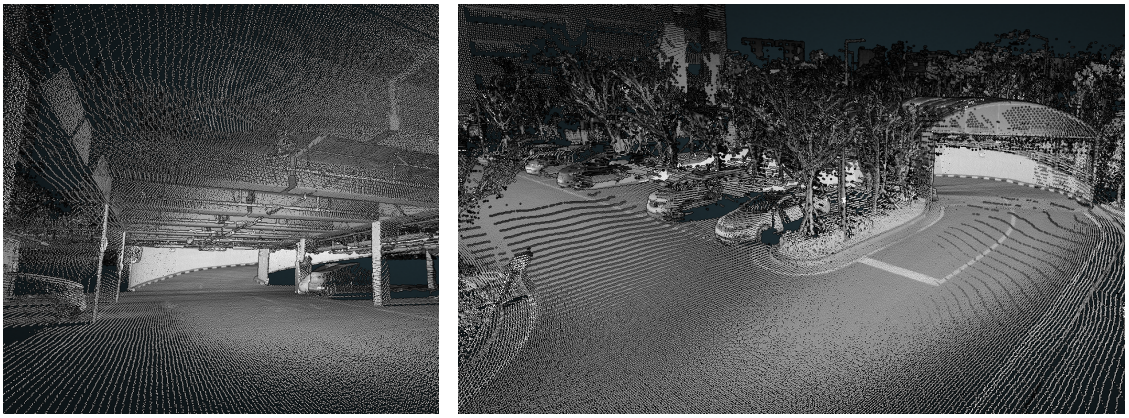
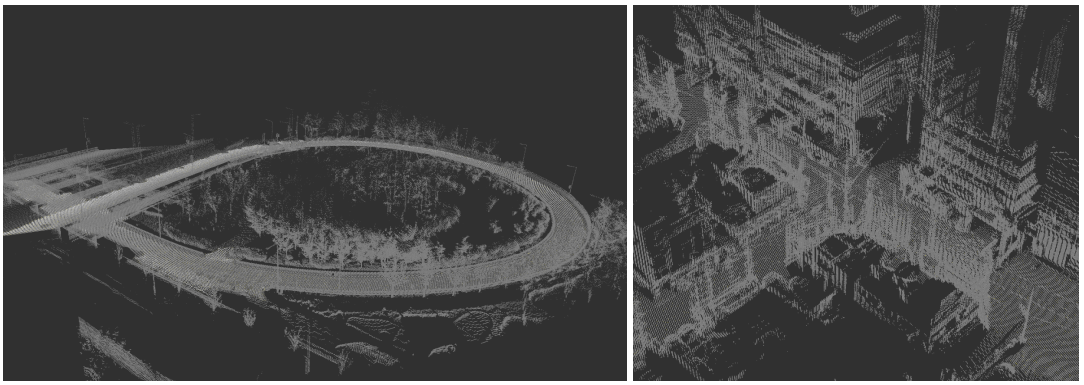


Figure 1.1: A two-floor parking garage. The area to the right is above the area to the left. The areas are interconnected by an arced ramp. The mission of the sweeper robot is to plan a path and clean both floors.



(a) Highway bridge (b) City crossing

Figure 1.2: Two environments that were used for evaluating the performance of the algorithms. To clean these environments the planner needs to handle that the bridge overlaps the road underneath in (a) and the complex geometry because of many obstacles on the sides of the roads in (b).

The first is *Sensor-based Coverage*, where the area gets covered by being in the field of view of a sensor. The goal is to find positions to place the sensor to make every part of the area covered by the sensor. The second type is *Footprint-based Coverage*, where the area gets covered by being visited by the robot. This thesis focuses on the latter, since the cleaning actuator is under the robot. There is also a distinction between online and offline CPP. Offline methods requires knowledge about the entire environment, while online methods plans the coverage path while exploring it. [6]

The CPP problem for covering indoor environments [6], croplands [17] and lawns [10] is a well studied problem with established solutions. These environments have a flat surface and are often easily dividable into rooms. This is however, not the case for many outdoor urban scenarios such as multi-floor parking garages, highway ramps and city streets. Apart from complex surface geometry with height variations they can also have multiple floors, making them impossible to represent as a 2D grid to use the established methods.

According to business leaders, autonomous vehicles are expected to be driving on our streets and do different tasks in complex environments in just a few years [24]. To solve the CPP problem in environments with multiple floors and height variations is increasingly relevant and will enable robots to perform tasks, where simple algorithms are insufficient.

1.2 Aim

This thesis focuses on offline footprint-based CPP that could handle large-scale urban environments with multiple floors and complex geometry. A method is developed to make two well known approaches, Inward Spiral [34] and BA* [31], perform well in a 3D representation of the environment. The aim is to develop a new better performing algorithm by combining existing solutions. To evaluate their performances, all three methods will be implemented in a simulation model of a multi floor parking garage, a highway bridge and a street crossing (see Figures 1.1 and 1.2).

1.3 Research questions

The input data is a 3D point cloud of the environments and the solution is a path with three dimensional waypoints. At first, the environment is analyzed to identify traversable and coverable areas to know what areas are accessible for the robot. Secondly, this information is used by an algorithm, which returns the solution path. This project will mainly focus on the latter part by addressing these questions:

1. How does Inward Spiral [34] and BA* [31] perform based on coverage, length of path and total rotation change over time in realistic outdoor environments?
2. Given a starting point, can both the path length and total rotation be reduced by combining the strengths of Inward Spiral and BA* in a new approach?
3. Using a configuration that was optimized for one starting point, how much does the performance of BA*, Inward Spiral and the new algorithm worsen for other starting points regarding coverage, length of path and total rotation?

1.4 Delimitations

The comparisons between the CPP algorithms is made on point clouds from three different data sets from the same data collection [21]. These point clouds are considered known and all calculations are made offline. In this project it is assumed that the robot follows the planned path exactly and that an area has been cleaned if the footprint of the robot has covered it while following the path. The footprint of the robot is assumed to be a circle with the robot breadth as diameter.

1.5 Thesis Outline

The outline of this thesis is as follows. Chapter 2 summarizes previous work done in the field of Coverage Path Planning. Details of the most relevant approaches together with the problem formulation are presented in Chapter 3. Chapter 4 describes how coverable and traversable areas are identified from a 3D point cloud. Implementation details of the CPP algorithms and the new approach is described in Chapter 5. The experiment setup and methodology are explained in Chapter 6. Results from the experiments are presented in Chapter 7 and discussed in Chapter 8. Chapter 9 concludes the thesis by answering the research questions and suggesting ideas for future work.

2 Related Work

The coverage path planning problem is a well known problem with many different solutions. It has been studied by many researchers throughout the years and in this chapter a summary of the most relevant approaches will be presented.

2.1 Grid Based Approach

A common approach to solve the coverage path planning problem in 2D is to represent the region of interest as a grid and visit all cells that could be covered. There are many grid based algorithms [14].

A simple solution is to follow the walls and make an inward spiral. Appearing in so called dead zones, could be solved with motion planning algorithms, such as A* [34]. Another spiral approach divides the area into big cells and subcells. By following a spanning tree through every free subcell the algorithm provides a close-to-optimal covering path [13].

A benefit of working with grid cells is that they could be assigned property values easily. This can be used by a Wavefront [33] or a Local Energy Minimization [5] algorithm, which creates a path by looking for the next unvisited cell with the best metrics. Metrics could be distance to a specific cell, translational distance, rotational distance or status of neighbour cells. This makes these types of algorithm more customizable for a specific robot and application comparing to the spiral based algorithms.

Other grid based algorithms are based on neural networks [32] and has the advantage of working in environments with dynamic obstacles. In [6], Bormann et al. compared the neural network approach with a Traveling Salesman Problem solution and the previously mentioned Local Energy Minimization. The experiments showed that the Local Energy Minimization was the best approach concerning computational time, total rotation and path length. On the other hand, it gave a little bit less coverage.

Generally, the cons of a grid based approaches is the need of approximation and that memory consumption is often exponential with the size of the area. They also require an accurate localization of the robot, making them more suitable for indoor environments. [14]

2.2 Cellular Decomposition Approach

An other approach is to decompose the area of interest into obstacle free regions and cover them by driving back and forth. The decomposition, the angle of the back and forth lines and the order of regions to cover varies between different algorithms. These algorithms often assumes that all obstacles and cleaning areas are represented as polygons. [14]

Trapezoidal decomposition is the simplest approach. For every vertex in the polygons, representing the obstacles as well as the boundary of the region of interest, a boundary for a trapezoid shaped cell is created by drawing a line. Unfortunately, this decomposition often results in a big number of cells which has to be covered one by one and consequently, generate an ineffective path. [7]

A better performing algorithm is called Boustrophedon decomposition. Instead of making a boundary for every vertex this algorithm uses only critical vertexes. Fewer cells makes the path shorter [7]. When the Boustrophedon decomposition were compared with the grid cell algorithms in [6] it had generally better coverage percentage, but longer computational time and paths in environments with obstacles.

The Boustrophedon decomposition is a specialization of Morse decomposition, which is a general way to decompose areas using slicing and critical points. The general method works for obstacles of any shape and the cells/slices can be shaped as spirals, spikes or any other shape that can be mathematically described. [7]

Since the paths of these back-and-forth methods includes a lot of turns, which are disadvantageous, Bochkarev and Smith [4] proposed a method that first decomposes the area, and then plans the path to minimize the number of turns by looking at the altitude of the polygons and optimize the decomposition.

A different approach is to divide the area by calculating a Voronoi graph which is based on finding central points between obstacles or cell boundaries. The advantage of this approach, in comparison to grid cell based approaches, is that its resolution depends on the complexity of the environment instead of the cell resolution, making it less space consuming. [30]

2.3 Graph Search Approach

An approach proposed in [9] is using graph search. It is based on the A*-algorithm and an occupancy grid of the environment. Since rotation is expensive, the algorithm's cost function is based on turns. The planned path consists of junctions of lines that covers the area and minimizes the number of turns.

A similar approach is called the BA* algorithm, which is a combination of Boustrophedon decomposition, which was described in 2.2, and A*. It is an online method that does not require prior knowledge of the environment. It covers one region at a time by making zigzag line paths and a backtracking list of points that are connected to other regions. As soon as a critical point is reached, it uses A* to find a path to the closest point in the backtracking list. This loop continues until the backtracking list is empty and all points has been covered. Based on simulations, this approach was shown to be better than Boustrophedon decomposition in

terms of the necessity of prior knowledge of the environment, the length of the path and the number of regions to cover. [31]

2.4 Random Sample Approach

The random sample approach algorithms consist of two steps. Firstly, they sample points until the area has been covered. Secondly, they find a continuous path that connects these points in a desirable way [14]. In the literature studies the random sample approach seems to be applied on Sensor-based CPP applications [8, 15, 11, 12]. However, if the sensor view port is set to the sweeping robot's footprint, the same algorithms is possible to apply on cleaning applications.

One approach to solve the first step is to sample a random uncovered point and then add the point nearby, which covers most uncovered area, to a list of goals [8]. Another approach is to sample positions randomly until the area gets covered and then find the best positions by picking the ones that covers most area one by one from the list of all sampled positions. Instead of sampling uniformly at random, a more efficient approach is to first sample points to cover the boundaries of the area, and then sample points inside the area [15].

The second problem is often a variant of the Travelling Salesman Problem (TSP), which is a well studied problem with many different exact and approximate solvers. In [8] the points are connected using a shortest path graph and an approximation to the TSP. A chained Lin-Kernighan TSP algorithm [1] was used in [11] to connect the points and a bi-directional rapidly-exploring random tree (RRT) algorithm [26] was applied between the points to avoid obstacles. Since these generated paths are rarely smooth, an algorithm to smoothen the generated path using a variation of RRT has been proposed [12].

2.5 3D approaches for Coverage Path Planning

Covering an uneven terrain based on a 2D algorithm with a back-and-forth approach leads to skipped spots and overlaps. When the robot is tilting, its range and the localization could be affected. To solve this problem, a Side-to-side 3D CPP approach was proposed in [17]. By using Digital Elevation Model (DEM) of the terrain, cylinders representing the paths can be placed side by side across the terrain to take the height differences in concern when setting the distance between the paths. The cylinder approach can also be used to find the angle that gives the best coverage efficiency.

Another similar 3D approach, which can be applied to any back-and-forth CPP algorithm as well, is to use the DEM to find the angle that minimizes the energy consumption. It can be done by calculating the power needed to execute every path with regard to the height differences for different angles. [18]

Just like [17, 18], the approach in [22] is originally made for agricultural applications as well. It also uses DEMs and creates a plan with side-by-side paths across the area. However, in this approach, the area is first decomposed into subregions based on the slope steepness. In each subregion, an optimal "seed curve" is found by finding a curve that minimizes the cost function. A "seed curve" could be an edge segment or a contour line, which unlike the other approaches does not have to be straight line. The coverage path plan is created by making subsequent paths side-by-side with an offset from the "seed curve" until the area is covered.

The literature of the grid search algorithms in 2.3 did not mention that they could be used in 3D. However, since the paths are dynamically constructed based on neighbour cells and A* can be used in 3D as well, it should be possible to apply them on non-planar surfaces as well.

An advantage of the random sampling approaches mentioned in 2.4 is that they can be used in 2D and 3D. They works good for handling complex structures, but lacks the desirable regularity compared to other methods. One way of solving this issue is to make the algorithm two-phased. In the first phase, waypoints on simple planar surfaces are structured in a grid and covered using a back-an-forth strategy. In the second phase, all points that has not been covered are covered using random sampling. [11]

3 Background

In this chapter, the most relevant approaches mentioned in Chapter 2 are detailed. The problem formulation is presented, as well as relevant theory for the method used in the experiments.

3.1 Problem formulation

We define a point cloud of the environment, $\mathcal{W}$, as a set of points, $p \in \mathcal{W}$, with the coordinates $[x, y, z] \in \mathbb{R}^3$. The geometry of the robot is defined as a 3D region R, transformed by the robots pose (position and rotation). $\mathcal{W}_{cov} \subseteq \mathcal{W}$ is a set of points p that are classified as *coverable* by the robot. A point, p, is coverable if R can be placed in such way that no point in $\mathcal{W} \backslash W_{cov}$ is inside R if p is inside R and is reachable from a given starting position $s_s \in \mathbb{R}^3$ without collisions. Additionally, the height difference Δz between two points inside R, p_1, $p_2 \in \mathcal{W}_{cov}$, can not be bigger than the maximum robot step height, h^R_{max}.

The Coverage Path Planning (CPP) problem is to find a path P starting from s_s with sequential positions $s \in P$ such that every point $p \in \mathcal{W}_{cov}$ must have been inside R at least once after placing R at every position in P.

3.2 Evaluation Measures

There are combinatorically many ways to create a path that solves the CPP problem. The choice of plan has to be based on the most sufficient properties. These properties are described below.

- **Coverage** - Defined as

$$C = \frac{N^p_{cov}}{N^p_{tot}} \tag{3.1}$$

  where N^p_{cov} is the amount of covered points and N^p_{tot} is the total amount of coverable points. In theory, C has to be 1 to solve the CPP problem, but in reality it is often lower and balanced against other properties.

- **Length of Path** - As path length is related to operation cost and time, minimizing the path length is of interest. The length of the path is the total traveled distance when moving through all waypoints in a path.
- **Total Rotation** - The optimal cleaning path for the sweeping robot is a straight path. Turns are not only expensive time-wise and energy-wise, but are also increasing the risk of missing spots. Therefore, a good path should have as few turns as possible. Comparisons are made by calculating the total amount of degrees that the robot rotated while executing the path.
- **Computational time** - The complexity of CPP algorithms can vary a lot. Even though an algorithm with higher complexity gives a slightly better path, a lower complexity algorithm could be more attractive if it saves time and requires cheaper components. An algorithm with low computational time is also more applicable in bigger environments. However, in offline applications this property has lower importance than the others.

3.3 Algorithm: BA*

BA* is a CPP algorithm proposed in [31], that is based on Boustrephedon motions and the A* search algorithm. The algorithm consists of following steps.

1. Cover the local area using the BM algorithm until a critical point is reached.
2. Use a backtracking list to find the next starting point.
3. Use A* to plan a collision free path to the next starting point.
4. Shorten the path using the A*SPT algorithm.
5. Follow the generated path and go to step 1 to cover a new area.

The algorithm will keep repeating these steps until Step 2 can not find a new uncovered starting point.

Step 1: BM algorithm

The goal of the Boustrophedon motion (BM) algorithm is to cover an unknown area by moving one step at a time and create a two dimensional tiling model M of the area. The tiles, c, are squares, with the size of the robot and are allowed overlap with each other. The steps can only be made in the north, south, west or east direction. The algorithm keeps exploring the

area until the robot reaches a position where it can not move in any of these 4 directions, a critical point c_{cp}, see Algorithm 1.

Algorithm 1: BM algorithm

Data: Starting point c_{sp}. Model M of the area.
Result: Updated model M of the area. Critical point c_{cp}. Path P_{local} until critical point.

```
Set CriticalPointFound → false
c_curr = c_sp
P_local = ∅
while CriticalPointFound is false do
    Set CriticalPointFound → true
    for position c_n in direction n ∈ north, south, east, west from c_curr do
        if c_n is available then
            Add tile c_n to M
            Add tile c_n to P_local
            c_curr = c_n
            Set CriticalPointFound → false
            break
        end
    end
end
c_cp = c_curr
return M, c_cp, P_local
```

Step 2: Find next starting point

After a critical point has been reached, the algorithm has to find a new starting point for exploring the next area. This is made by adding specific tiles to a backtracking list while covering areas (Step 1). The backtracking list, L_b, is defined as

$$L_b = \{c | c \in M \text{ and } \mu(c) \geqslant 1\}, \tag{3.2}$$

where $\mu(c)$ is

$$\mu(c) = b(c_1, c_8) + b(c_1, c_2) + b(c_5, c_6) + b(c_5, c_4) + b(c_7, c_6) + b(c_7, c_8), \tag{3.3}$$

and $b(c_i, c_j)$ is

$$b(c_i, c_j) = \begin{cases} 1, & \text{if } c_i \text{ is free and } c_j \text{ is blocked} \\ 0, & \text{otherwise} \end{cases}. \tag{3.4}$$

c_i is a neighbouring cell to c according to figure 3.1. A cell is blocked if it has been visited or if it is inaccessible by the robot.

The next starting point, c_{sp} is chosen by picking the position of the closest cell $c \in L_b$ according to Euclidean distance, Manhattan distance or shortest path using only covered cells.

Step 3: A* algorithm

A* algorithm is a well known search algorithm that can be used to find the optimal path between two points in a grid or graph. The idea is to use a heuristic estimate,

$$f(s) = g(s) + h(s), \tag{3.5}$$

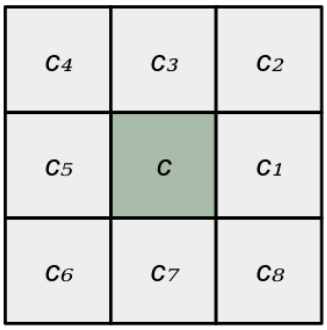


Figure 3.1: Definition of a neighbouring cell c_i of c. Used in BA* to find the next starting point.

where $g(s)$ is the length of the shortest path from the starting position, s_s, to a position s that has been found so far and the heuristic function $h(s)$ is the estimated length between s and the goal s_g. The algorithm is described in Algorithm 2. It uses a priority queue, Q_{open}, which stores points that have been found, but its neighbours have not been evaluated yet. The priority is based on $f(s)$ in equation 3.5. It means that the `pop` function always returns the point with the smallest $f(s)$ and removes it from the priority queue. Every iteration starts with applying `pop` on Q_{open}. The returned point is then moved to a list, Q_{closed}, to keep track of the points that have been visited. Thereafter, the neighbours of this point are evaluated to see if the estimated length of the total path gets shorter if the path goes through them. If that is the case or if a neighbour has not been visited, it gets a new $g(s)$ value and is added to the Q_{open} list. It also gets the original point as *parent*. The *parent* property is used to backtrack the path when the goal point has been reached.

In BA*, this algorithm is used to find the shortest path from the critical point c_{cp} (from Step 1) to the next starting point c_{sp} (from Step 2). The euclidean distance between s and the goal s_g is used as the heuristic function $h(s)$.

Algorithm 2: A* algorithm

Data: Starting point s_s. Goal point s_g. Heuristic function `heuristic(s)`.
Result: Path P from s_s to s_g.

```
Q_open =empty priority queue
Q_closed = ∅
g[s_s] = 0
parent[s_s] = s_s
Q_open.push(s_s, g[s_s] + heuristic(s_s))
while Q_open ≠ ∅ do
    s ← Q_open.pop()
    if s is s_g then
        P = ∅
        while s is not s_s do
            P.push_back(s)
            s ← parent[s]
        end
        return reversed P
    end
    Q_closed.push(s)
    for neighbour s_n of s do
        if s_n not in Q_closed then
            if s_n not in Q_open or g[s] + Distance between s and s_n < g[s_n] then
                g[s_n] = g[s] + Distance between s and s_n
                parent[s_n] = s
                f = g[s_n] + heuristic(s_n)
                if s_n ∈ Q_open then
                    Q_open[s_n] = f
                else
                    Q_open.push(s_n, f)
                end
            end
        end
    end
end
```

Step 4: A*SPT Algorithm

The A* algorithm in Step 3 generates a collision free path, P, from the critical point c_{cp} to the next starting point c_{sp}. Since this path is based on square tiles, it often includes a lot of heading changes. Consequently, it will not be the shortest path. A*SPT is an algorithm that takes a path as input and smooths it out to make it shorter and unbounded to the tile model,

see algorithm 3. The idea is to find the farthest point from the start with a line-of-sight and add it to the smooth path $\hat{P}$. This is repeated until the found point is the goal point.

Algorithm 3: A*SPT algorithm

Data: Path $P = \{s_1, s_2, ..., s_n\}$.
Result: Smooth path $\hat{P} = \{\hat{s}_1, \hat{s}_2, ..., \hat{s}_k\}$.
$k = 1$
$\hat{P} = \{s_1\}$
while s_k is not s_n **do**
  for $i \in n, n\text{-}1, ..., k\text{+}1$ **do**
    P_{s_k,s_i} = Straight line path between s_k and s_i
    if P_{s_k,s_i} is collision free **then**
      $\hat{P}$.`push_back`(s_i)
      Set $k \rightarrow k + 1$
    end
  end
end
return $\hat{P}$

Step 5: Follow path and go back to Step 1

When the smooth path $\hat{P}$ between the critical point and the next starting point has been generated, the next step is to make the robot follow this path. When the next starting point is reached, the algorithms goes back to Step 1 and repeats the process.

3.4 Algorithm: Inward Spiral

The Inward Spiral is a grid-based solution to the coverage path planning problem proposed in [34]. The idea is to clean the area counter-clockwise (or opposite) by keeping the robot as much to the right as possible and only turn left if the grid cell in front of the robot has been cleaned already or is inaccessible. This continues until the robot has no valid options, reaches a dead zone. By using a breadth first search (BFS), it then finds the closest grid cell that has not been cleaned and finds the shortest path to that point using A*. This cycle repeats until the area has been covered. In summary:

1. Clean area in an inward spiral motion until a dead zone is reached
2. Find closest uncovered accessible cell using BFS.
3. Find shortest path using A* to the closest uncovered cell and go back to Step 1.

Step 1 - Clean area in an Inward spiral motion

The inward spiral motion starts in a corner of the area. If the counter-clockwise the direction is chosen it follows the right wall until it reaches an obstacle or a covered cell. The algorithm to create this movement is described in Algorithm 4.

Algorithm 4: Generate path that cover the area in an inward spiral motion until a dead zone has been reached

Data: Starting point c_{sp}.
Result: Path P_{local} until dead zone.
Set $DeadZoneReached \rightarrow false$
$P = \{c_{sp}\}$
$c_{curr} = c_{sp}$
while $DeadZoneReached$ is $false$ **do**
  Set $DeadZoneReached \rightarrow true$
  for neighbour c_n of c_{curr} for $n \in$ right, forward, left **do**
    if c_n is covered or inaccessible **then**
      continue
    end
    P_{local}.`push_back`(c_n)
    Set $c_{curr} \rightarrow c_n$
    Set $DeadZoneReached \rightarrow false$
    break
  end
end
return P_{local}

Step 2 - Find closest uncovered cell using BFS

After reaching a dead zone the robot needs a new starting point for the next inward spiral motion. Since this method is grid based a BFS algorithm can be used to find the closest uncovered cell that is free from obstacles. The idea is to walk layer by layer from the original robot position c_s to find the closest uncovered obstacle free cell, see details in Algorithm 5 and figure 3.2.

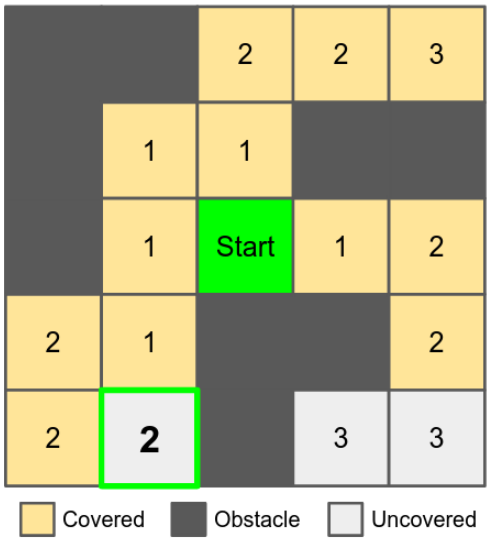


Figure 3.2: An example of the BFS algorithm. The number in the cells represents the number of steps from START. The goal of the algorithm is to find the closest uncoverable cell.

Algorithm 5: Breadth First Search (BFS)

Data: Start cell c_s.
Result: Closest uncovered obstacle free cell c_f.
$Q = \{c_s\}$
$V = \{c_s\}$
while $Q \neq \emptyset$ **do**
  $c = Q$.`pop_front()`
  for neighbour c_n of c **do**
    if c_n is inaccessible or is in V **then**
      continue
    end
    if c_n is uncovered **then**
      return c_n
    end
    $V = V \cup c_n$
    $Q = Q$.`push_back`(c_n)
  end
end
return *Failure*

If the algorithm does not find a new starting point, the area has been fully covered and the path planning is finished.

Step 3 - Find shortest path using A* to the closest uncovered cell

When a new starting point has been discovered, the shortest path from the current robot position to the new starting point is generated using A*, which is described in detail in Algorithm 2. After reaching the point, the algorithm goes back to step 1 and covers the area in an inward spiral motion until a new dead zone is reached.

3.5 Algorithm: Sampling-Based Coverage Path Planning

An approach that is used to cover complex 3D structures was proposed by B. Englot and F. Hover in [11]. Unlike BA* and Inward Spiral it is a sensor-based CPP algorithm. The approach can be divided into following steps:

1. Locate planar areas and waypoints for simple back-and-forth covering.
2. Find waypoints in individual complex areas with randomized sampling.
3. Remove redundant waypoints using greedy and pruning algorithms.
4. Create a sequence of points that visits all waypoints using a Traveling Salesman Problem algorithm.
5. Create collision free paths between all waypoints using RRT.

Step 1 - Planar areas

The first step is to segmentize the data representing the environment into planar areas. In [11], this was made with a method proposed in [2] where the data was a triangle mesh.

The next step is to generate waypoints for every plane. This is done by choosing a random point on the plane and expand a grid of waypoints in the four directions north, south, east and west. It generates new waypoints in every direction along the plane until collision or if the waypoint is out of range to cover the plane. This generates a set of paths that covers all planar areas in the environment.

Step 2 - Complex areas

Waypoints on complex areas that could not be expressed as planes are generated using random sampling. The algorithm samples positions until every coverable point could be covered from k different sampled positions.

Step 3 - Remove redundant waypoints

After Step 1 and 2 each point in the area should be covered at least once if the robot visits all the waypoints. Since some points are covered more than once, there is a possibility that some of them could be removed, which would reduce the computational time and memory usage of the algorithm in the following steps. The following algorithms were proposed in [23], and applied in [11].

At first, a greedy algorithm will be used. It iteratively chooses the set, the path for planar areas and the waypoint for complex areas, that covers most uncovered points until all points have been covered at least once.

Then, a pruning algorithm removes every set that does not cover a point uniquely. It iteratively chooses the set that minimizes the overlapping with covered points. This algorithm is also applied to individual rows and columns of the path grids that covers the planar areas.

Step 4 - Traveling Salesman

When the minimum amount of waypoints to cover the area has been generated, the next step is to generate the path, that visits all waypoints. Before solving this problem, which is called the Traveling Salesman Problem (TSP), all waypoints has to be represented as a graph.

The path through the grids on planar areas is trivial since it is a basic back-and-forth motion. Therefore, each planar area path can be reduced to a pair of points to represent the entry and exit. To ensure that this pair appear adjacent in the TSP solution, the cost of the edge between them, is set to zero. All other edges between nodes are given a cost of the Euclidean distance plus a large number big enough to ensure that the entry-exit pair will be adjacent in the TSP solution.

TSP is NP-hard, so approximate methods are used in practice. One efficient approximate mathod, which was used in [11], is to use Lin-Kernighan algorithm [19]. It is an approximate algorithm giving a non-optimal solution in a relatively short time. The idea is to start with a tour that is generated in a randomized way and improve it until no more improvements are possible.

The entry and exit of the sweep paths were defined before starting the TSP algorithm. Since it is possible that another order or set of entries and exits gives a better solution, different combinations are tested to see if any changes results in a shorter path.

Step 5 - RRT

After generating a sequence of waypoints, a bi-directional rapidly-exploring random tree (RRT) algorithm is used to find collision free paths between the waypoints. RRT, described in algorithm 7, builds two search trees, that are biased to grow towards each other. When the two trees meet, a path that goes from starting point p_s to goal point p_g is generated. The trees grow using the function `extend`, see algorithm 6. At first, one of the trees is extended towards a random sampled point p_{rand}. If the extension was successful and generated a new point p_{new}, the other tree is extended towards p_{new}. Before the next iteration, the trees get swapped, meaning that the second tree will now be extended towards a new random sampled point. [26]

The first step of the `extend` function is to find the point, p_{near}, in the search tree, T, that is the closest to the given point, p_{ext}. The second step is to generate a point, p_{new}, at a user defined distance from p_{near} towards p_{ext}. If p_{new} is a traversable point, the point and the edge from p_{near} to p_{new} is added to the search tree. This is illustrated in figure 3.3. The status of the point is set to *Reached* if the point reached p_{ext}, $Trapped$ if it was untraversable and *Advanced* if it was traversable, but did not reach p_{ext}. [26]

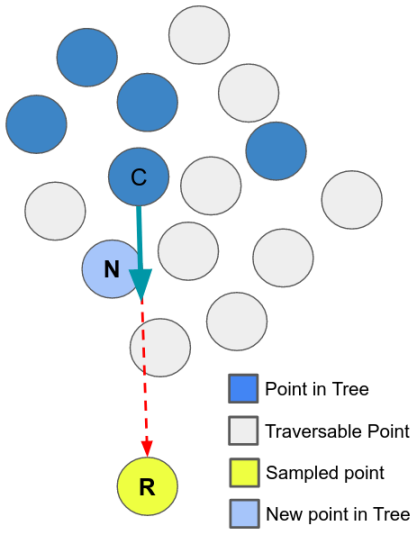


Figure 3.3: Illustration of the `extend` algorithm. R represents the sampled p_{ext} point, C is the closest point in the tree p_{near} and N is the new node in the tree p_{new}.

Algorithm 6: Extend algorithm

Data: Search tree T. Point p_{ext} defining the direction of extension. Step size λ_{RRT}
Result: New point p_{new}.
p_{near} = closest point $\in T$ from p_{ext}
p_{new} = point $\notin T$ closest to the position at a distance λ_{RRT} towards p_{ext} from p_{near}
if p_{new} is traversable **then**
  T = T $\cup \{p_{new}\}$
  if p_{new} is p_{ext} **then**
    $status[p_{new}] = Reached$
  else
    $status[p_{new}] = Advanced$
  end
else
  $status[p_{new}] = Trapped$
end
return p_{new}

Algorithm 7: RRT Path Planning Algorithm to find a path between to points.

Data: Point cloud of environment $\mathcal{W}$. Start point p_s. Goal point p_g. Max iterations N_{max}^{RRT}.
Result: Path P from p_s to p_g
$T_a = \{p_s\}$
$T_b = \{p_g\}$
for $k \in 1, 2, ..., N_{max}^{RRT}$ **do**
  p_{rand} = random sampled point in $\mathcal{W}$
  p_{new} = `extend`(T_a, p_{rand})
  if $status(p_{new}) \neq Trapped$ **then**
    if $status($`extend`$(T_b, p_{new}))$ is *Reached* **then**
      return Shortest path from p_s to p_g through T_a and T_b
    end
  end
  `swap`(T_a, T_b)
end
return *Failure*

When collision free paths between the points have been generated, the cost of the edges between the waypoints are set to the distance of the generated paths. Finally, the TSP algorithm is repeated with the updated costs until a stable solution is found.

3.6 Bayesian Optimization

Many systems have multiple parameters that needs to be tuned. To avoid hand tuned optimization of parameters, automatic approaches have been developed. One of these methods is HyperOpt [3], which is a library implementing Bayesian Optimization. Bayesian Optimization is model-based approach that minimizes a given loss function by tuning parameters. Starting from a prior probabilistic model it sequentially predicts the input parameters that would generate the lowest loss and updates the model according to the output. After N iterations it makes a final input recommendation that is the set of parameter values that gave the lowest loss during the optimization. This method is very sample efficient and useful in situations where it is costly to try new input values. [29]

4 Terrain Assessment

To make a successful coverage path planning, good information about what regions of the environment that has to be covered is a necessity. Finding traversable and coverable regions in a point cloud is not the main focus of this thesis, but an important part of the method to plan a path on a given point cloud. In this chapter, the problem, method and results will be presented with some discussion at the end.

4.1 Problem

The problem is to distinguish which points in a given point cloud are reachable for the sweeping robot. In many applications all reachable points should be covered. We make following definitions for individual points (see figure 4.1):

- **Obstacle** - Point that can not be inside nor in contact with the robot body
- **Traversable** - Point that can be visited by the robots geometrical centre.
- **Coverable** - A point that could be covered by the range of the robot. After visiting every traversable point, all points that has been within the range of the robot are coverable.
- **Inaccessible** - Traversable and coverable points that are not accessible since there are no feasible path to reach them.

For comuptational reasons, the point cloud of the environment can be arbitary discretized into floors and cells to avoid calculating traversability for every point. Every floor consists of a two dimensional cell grid where every cell has an elevation height, a Digital Elevation Model (DEM) of the floor. The maximum size of these cells is the robot footprint area and could be smaller to get a result with higher resolution. A cell can be:

- **Invalid** - If the number of points in the cell is too low.

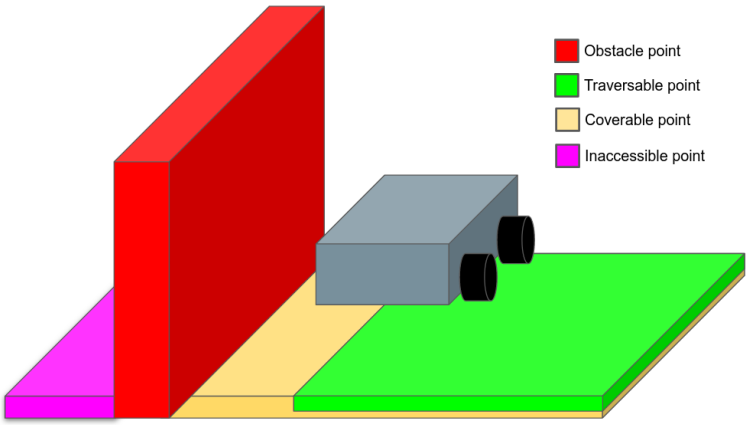


Figure 4.1: Point definitions.

- **Ground** - If the elevation height of the cell is close to the lowest point in the point cloud of the floor.
- **Obstacle** - A cell with a wall or an obstacle that makes it inaccessible by the robot.
- **Inaccessible** - Cell that are not accessible since there are no feasible path to reach it from the main coverable area.
- **Coverable** - A cell that could be covered by the robot. The criterias are described below.
- **Border** - An obstacle or invalid cell that has a coverable cell as neighbour.

Coverable points can only exist in coverable cells. Points in all other cells are classified as obstacle points. Only the points in a cell that are close to the elevation height of the cell are classified as coverable, the rest are obstacle points as well.

The criterias of a coverable cell is based on three parameters:

- **Minimum ceiling height**, h^C_{min} - the minimum height of free space in a cell. Free space is the space between the elevation height of the cell and the lowest point above it, see figure 4.2
- **Maximum step height**, h^R_{max} - the height difference between two points that the robot is able to get over without getting stuck, see figure 4.3
- **Minimum number of coverable points in cell**, N^{cov}_{min} - tuned constant to exclude cells with insufficient information. The constant is representing a minimum amount of coverable points in the cell to make it count as valid.

The first two are robot specific, while the third is a hand tuned constant that depends on the density of the point cloud.

The requirements for a coverable cell is following,

- The absolute height difference between the cell and at least one of it's neighbors has to be lower than the maximum robot step h^R_{max}.

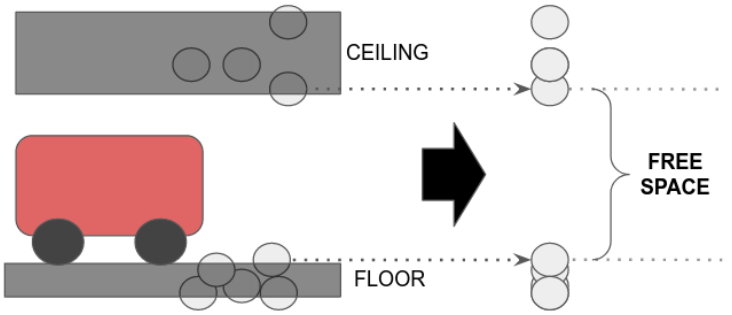


Figure 4.2: Definition of free space. It is detected by sorting the z-values of all points in the cell and find two following points with a height difference bigger than the robot height.

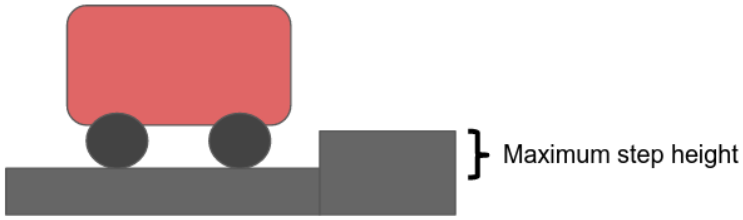


Figure 4.3: Definition of the maximum step height. The biggest height that the robot can go over without getting stuck

- The minimum ceiling height h^C_{min} has to be bigger than the height of the robot h_R.
- The amount of coverable points in the cell have to be more than the minimum number of coverable points in cell constant N^{cov}_{min}
- The cell should be accessible from other coverable cells.

4.2 Method

The method that was used to find all traversable and coverable points in a point cloud can be divided into four parts:

1. **Floor segmentation** - Divides the point cloud into different floors.
2. **Cell segmentation** - For each floor separately creates a DEM of the floor by splitting the point cloud into cells and finding their elevations. Finds invalid cells.
3. **Cell classification** - Finds the biggest connected area of coverable cells - the *main coverable area*. Finds border, inaccessible and obstacle cells as well.
4. **Point classification** - Classifies points in the point cloud using following steps:
   a) Sets all points that are not in the main coverable area as obstacle.
   b) Sets all points in the main coverable area that are more than a robot radius away from an obstacle point as traversable.
   c) Sets all points that are within a robot radius from a traversable point as coverable. Set unclassified points to inaccessible.

These parts are described in detail below.

Floor Segmentation

The first step was to divide the point cloud into different floors. Following steps, proposed in [28], were made:

1. The point cloud was divided into thin boxes with thickness Δh_L laying upon each other.
2. The amount of points in every box were counted and put into a histogram. The local maximas were identified.
3. A threshold parameter N^F_{min} was hand tuned based on the histogram to exclude local maximas with low amount of points. After the tuning, all local maximas that had a value bigger than N^F_{min} were seen as potential ground heights of a floor.
4. The potential ground heights were examined to detect ceilings of rooms by requiring height difference of at least h^C_{min}.
5. When the ground height of every floor was known, the original point cloud was cut at these heights into segments. To include all points of the ground floor in case of small inclinations an offset parameter h^F_{offset} was hand tuned and added to the ground floor heights. The result is one point cloud for each floor.

Cell Segmentation

Next step was to make a DEM of each floor, E_i. The point cloud of the floors was transformed into a 2D grid with cells, where every cell had an elevation height. It was assumed that every valid cell had only one elevation, e_c. The method to find the elevation height was proposed in [28].

Firstly, the point cloud was simply projected on a 2D plane and divided into square cells to create a 2D-grid M. Secondly, the elevation for every cell was found and invalid cells were detected. It was done using Algorithm 8. The idea of the algorithm was to sort the z-values for all points in a cell and then find two following points with a height difference that was

bigger than the robot height h_R. The elevation height of the cell in the DEM was set to the height of the lowest point.

Algorithm 8: Find elevation for each cell and detect invalid cells

Data: 2D-grid M of environment. Point cloud $\mathcal{W}$. Minimum number of coverable points in cell N_{min}^{cov}. Robot height h_R. Maximum step height h_{max}^R.
Result: DEM E of the environment

```
for cell c ∈ M do
    W_c = points in W inside cell c
    Z_c = z-values of W_c
    for z_i in sorted Z_c do
        if | z_{i-1} - z_i | > h_R then
            break
        end
    end
    e_c = z_{i-1}
    W_{c,cov} = Points in W_c with a z closer than h^R_max from e_c
    if Number of points in W_{c,cov} > N^{cov}_min then
        Add a cell c with e_c as elevation height and W_{c,cov} as coverable points to E
    else
        Add cell c to E as Invalid
    end
end
```

Cell Classification

After finding the elevation height of the cells, the heights were used to classify the accessibility of the cells for the robot. Some of the cells were already classified as invalid in the previous step. The classification algorithm is described in Algorithm 9. It is based on the idea of doing a breadth first search starting from a ground cell which was presented in [28].

First, all ground cells, cells with the same height as the lowest z of the floor, z^F were inserted into a set of ground cells E_{ground}. All these cells are potential starting points for a cluster of connected coverable cells, an *island*. These islands were built using a breadth first search on the DEM of the environment. When all islands had been detected, the biggest island, consisting the biggest amount of cells was chosen as the main coverable area, E_{cov}. All cells in E_{cov} were classified as coverable. Cells in other islands were classified as inaccessible. Cells that were neither invalid, coverable or inaccessible were classified as obstacle cells.

For later calculations of traversable points a set of border cells, E_{border} had to be generated. This was done by looking for uncoverable neighbours of every cell of the main covarble area, see details in Algorithm 10.

Algorithm 9: Detect the main coverable area

Data: Height z^F of the floor ground. DEM E of the floor
Result: Coverable cells E_{cov}
E_{ground} = cells in E with elevation close to z^F
$I = \varnothing$
while $E_{ground} \neq \varnothing$ **do**
  $c_{start} \leftarrow$ arbitrary cell in E_{ground}
  $E_{visited} = \{c_{start}\}$
  $Q_{cov} = \{c_{start}\}$
  while $Q_{cov} \neq \varnothing$ **do**
    $c_{curr} \leftarrow Q_{cov}$.pop()
    for neighbour c_n of c_{curr} **do**
      if c_n is not accessible from c_{curr} **then**
        continue
      end
      if c_n is in $E_{visited}$ **then**
        continue
      end
      if c_n is in E_{ground} **then**
        $E_{ground} = E_{ground} \setminus c_n$
      end
      $E_{visited} = E_{visited} \cup c_n$
      Q_{cov}.push(c_n)
    end
  end
  $I = I \cup E_{visited}$
end
E_{cov} = the set with biggest amount of cells in I
return E_{cov}

Algorithm 10: Detect all border cells of the main coverable area

Data: Coverable cells E_{cov}. DEM E of the environment.
Result: Border cells E_{border}.
$E_{border} = \varnothing$
for cell c in E_{cov} **do**
  for neighbour c_n of c in E **do**
    if c_n is not in E_{cov} **then**
      $E_{border} = E_{border} \cup c$
    end
  end
end
return E_{border}

Point Classification

The last step in the terrain assessment is to classify the points into traversable, coverable and obstacle/inaccessible. The purpose is to stop the path planning from making waypoints that are close to obstacles, where there is a risk of collision.

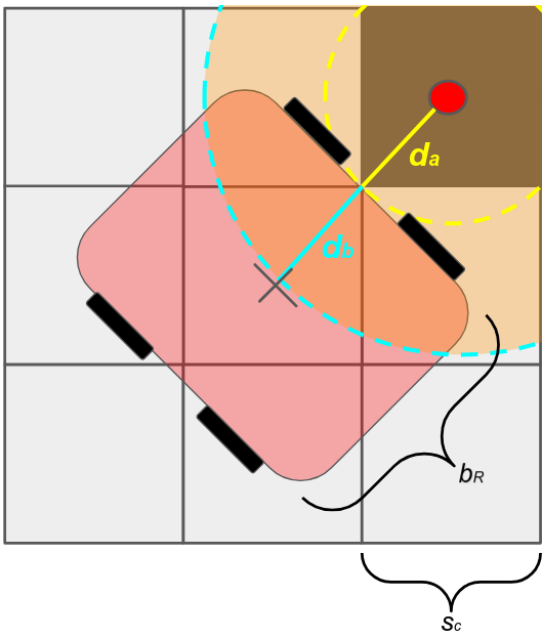


Figure 4.4: Illustration of the distance $d_{collision} = d_a + d_b$, where $d_a = \frac{1}{\sqrt{2}} s_c$ and $d_b = 0.5 b_R$. The red point is a border point. Yellow area is untraversable due to risk of collision. s_c is the side length of the cells and b_R is the breadth of the robot

First, all coverable points for every cell in the main coverable area E_{cov} were extracted as a potentially coverable point cloud $\mathcal{W}_{p,cov}$. Then, a point was created for every border cell in E_{border} and put into a border point point cloud $\mathcal{W}_{border}$. The (x, y)-position of the created points were set to the center position of the border cell. The z value was set to the elevation height of the neighbouring coverable cell in E_{cov}.

A point was classified as traversable if the distance to the closest border point was bigger than

$$d_{collision} = \frac{1}{\sqrt{2}} s_c + 0.5 b_R$$

where s_c is the side length of the square cells in the cell grid and b_R is the breadth of the robot. See motivation and explanation in Figure 4.4.

The idea was to first assume that every coverable point was traversable. Then, go through every border point and set points nearby as untraversable. Thereafter, the algorithm detects all inaccessible points by finding coverable points that are not within the robot range from a traversable point. It is assumed that the robot covers all points within a radius r_R from the position of the robot. The algorithm is described in detail in algorithm 11 and illustrated in figure 4.5.

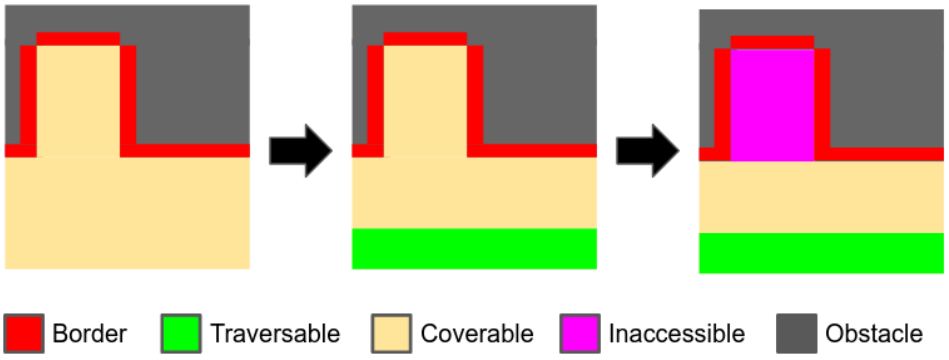


Figure 4.5: Illustration of the point classification algorithm. First, border points and potential coverable points are identified. Secondly, coverable points that are not close to border points are classified as traversable. Thirdly, only points that are reachable from the traversable points are classified as coverable, the rest are inaccessible.

Algorithm 11: Find traversable positions for the robot and all coverable points.

Data: Potentially coverable point cloud $\mathcal{W}_{p,cov}$. Point cloud $\mathcal{W}_{border}$ with border points.
Distance $d_{collision}$ from a border point to robot position with no risk of collision.
Radius range of robot r_R
Result: Coverable point cloud $\mathcal{W}_{cov}$. Traversable point cloud $\mathcal{W}_{trav}$.
$\mathcal{W}_{trav} = \mathcal{W}_{p,cov}$
for point p in $\mathcal{W}_{border}$ **do**
  $\mathcal{W}_{collision}$ = points in $\mathcal{W}_{trav}$ closer than $d_{collision}$ from p
  $\mathcal{W}_{trav} = \mathcal{W}_{trav} \setminus \mathcal{W}_{collision}$
end
$Q_{cov} = \mathcal{W}_{p,cov} \setminus \mathcal{W}_{trav}$
$\mathcal{W}_{cov} = \mathcal{W}_{p,cov}$
while $Q_{cov} \neq \emptyset$ **do**
  $p_{cov} \leftarrow Q_{cov}$.pop()
  if distance from p_{cov} to the closest point in $\mathcal{W}_{trav}$ is bigger than r_R **then**
    $\mathcal{W}_{cov} = \mathcal{W}_{cov} \setminus p_{cov}$
  end
end
return $\mathcal{W}_{cov}$, $\mathcal{W}_{trav}$

4.3 Results

Three different point clouds were classified; a two-storey parking garage (Figure 1.1), a highway bridge ramp (Figure 1.2a) and a city street with a crossing (Figure 1.2b).

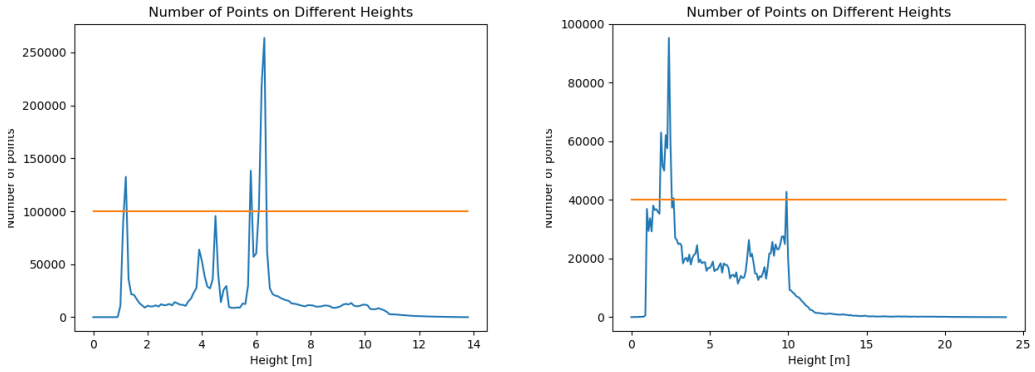
Two floors were detected in the parking garage and the highway bridge point clouds. The histogram in figure 4.6 shows the number of points for different heights. By looking at the histogram a tuned threshold value N^F_{min}, represented as the orange line in the figure, was chosen to leave the three highest peaks as potential ground heights. Since the peak in the middle in Figure 4.6a and the first peak in Figure 4.6b does not fulfill the requirement of having h^F_{min} meters free until next peak, the floor segmentation algorithm classified it as a false ground height. Finally, the other two peaks defined the heights of the floor ground and the point cloud could be divided.

Table 4.1: Hand tuned parameters used in Terrain Assessment. N_{min}^{cov} was tuned based on the density of the point cloud. N_{min}^{F} was tuned to make a reasonable floor segmentation using the histograms in Figure 4.6. h_{offset}^{F} was tuned to include all points of the ground of the floor.

Parameter	Explanation	Parking garage	Highway bridge	City crossing
N_{min}^{cov}	Min. coverable points in cell	12.5	12.5	25
N_{min}^{F}	Min. points in ground layer	10^5	$4 \cdot 10^4$	$1.5 \cdot 10^4$
h_{offset}^{F}	Ground floor height offset	0.7m	2m	0.1m

Table 4.2: Parameters used in Terrain Assessment. Values of *Real data* parameters are based on data from the real world provided by the developers of the robot. *Prediction* parameters are based on predictions provided by the developers of the robot. Values of *Resolution* parameters are minimised to give a good resolution while keeping the computational time on a reasonable level.

Parameter	Value	Explanation	Based on
b_R	0.75 m	Breadth of robot	Real data
h_R	1 m	Height of robot	Real data
h_{max}^{R}	0.2 m	Max. step height of robot	Prediction
s_c	0.5 m	Cell size. Length of side.	Resolution
Δh_L	0.1 m	Thickness of each layer	Resolution
h_{min}^{F}	2 m	Min. height of a floor	Real data

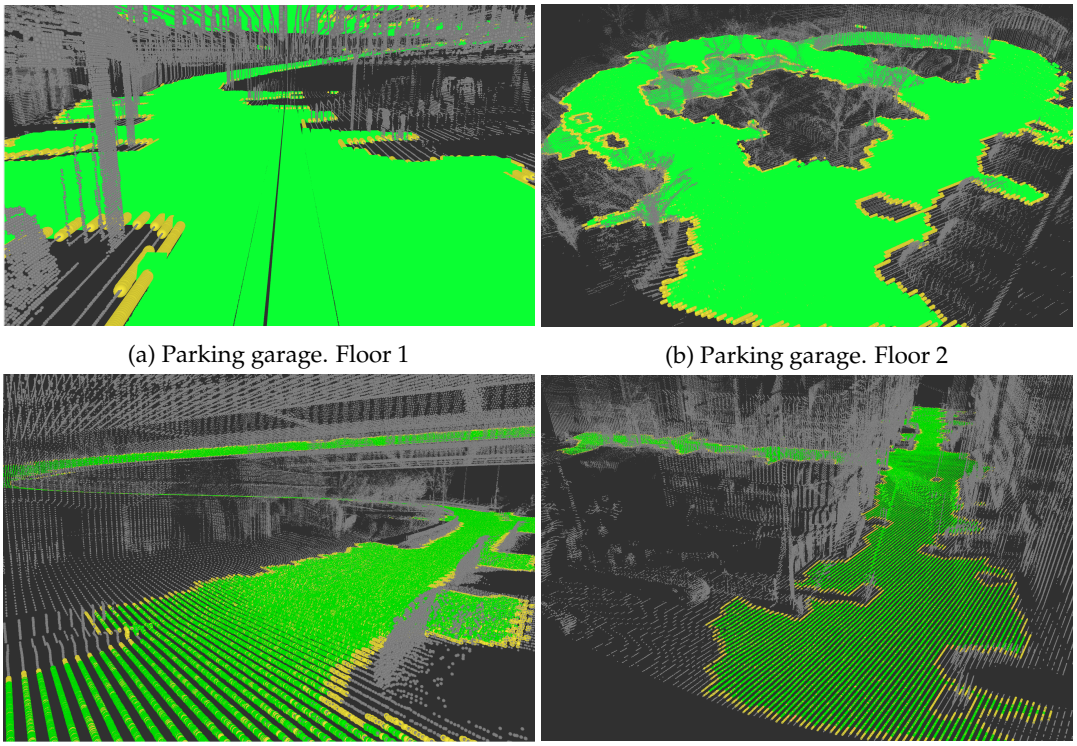


(a) Parking garage

(b) Highway bridge

Figure 4.6: The histograms used for floor segmentation. Two major peaks at the height can be identified for both environments. They represent the height of the ground of the two floors. The other peaks are obstacles and ceiling. The orange line represents the hand tuned N_{min}^{F} parameter, representing the minimum amount of points in a layer to be classified as a potential ground floor height. By neglecting all peaks under this line it leaves only three peaks to possibly represent the height of a ground floor. One peak is then excluded, since it has less than 2 meter distance until next peak.

The values of hand tuned parameters for each environment is presented in Table 4.1. Values and explanation of all common parameters are presented in Table 4.2. The result of the terrain assessment is shown in figure 4.7 and tables 4.3 and 4.4.



(a) Parking garage. Floor 1

(b) Parking garage. Floor 2

(c) Highway bridge

(d) City crossing

Figure 4.7: Result of Terrain Assessment. Green points represents traversable areas, yellow are only coverable and grey are points that are obstacle and inaccessible points

Table 4.3: Result Cell classification of the Terrain Assessment. COVERABLE area can possibly be covered by the robot. OBSTACLE area that has an elevation height that makes it inacessible. INACCESSIBLE area is the area that could be covered, but is not connected to the main coverable area. INVALID area consists of cells that does not have enough information to be evaluated. TOTAL is the total area of the floor, including areas with no points.

Environment	**Floor #**	**COVERABLE** $[m^2]$	**OBSTACLE** $[m^2]$	**INACCESSIBLE** $[m^2]$	**INVALID** $[m^2]$	**TOTAL** $[m^2]$
Garage	1	694	235	2	2290	3220
Garage	2	1208	378	44	1941	3570
Bridge	1	2116	693	433	16368	19610
Bridge	2	521	217	849	18024	19610
Crossing	1	1296	436	0	8119	9850

Table 4.4: Result of Point Classification of Terrain Assessment. TRAVERSABLE points are drivable positions and can be visited by the robot. If the robot would visit all these points, all COVERABLE points would be covered. INACCESSIBLE are the points that could be covered, but is too far away from a TRAVERSABLE point. OBSTACLE points are not allowed to be within the range of the robot. TOTAL is the total amount of points in the point cloud

Environment	**COVERABLE #**	**TRAVERSABLE #**	**INACCESSIBLE #**	**OBSTACLE #**	**TOTAL #**
Garage	1 171 664 (45%)	1 089 865 (42%)	43 027 (2%)	1 411 447 (54%)	2 626 138
Bridge	900 137 (38%)	837 584 (35%)	31 632 (1%)	1 433 675 (61%)	2 365 444
Crossing	1 133 544 (35%)	1 030 455 (32%)	40 813 (1%)	2 054 517 (64%)	3 228 874

4.4 Discussion

As shown in figure 4.7 the results of the terrain assessment are in line with expectations. Points that were classified as traversable were on a distance from obstacle points with coverable points in between, similar to the illustration in figure 4.1.

A fault of this process is that it only makes sure that all coverable cells are connected in a main coverable area, but does not ensure that all traversable points are connected. Therefore, in scenarios such as in Figure 4.8 a few small clusters of traversable points with connected coverable points are in fact inaccessible.

The algorithm did also struggled with classifying some parts. An example is the wall of the ramp in the parking garage environment. The floor segmentation divided the environment into two floors and classified the cells separately without taking the classification of the other floor into account. For a wall of a ramp that ranges over multiple floors this became a problem. The classification of the first floor classified wall points at the floor splitting height as coverable. Most of these points were later classified as inaccessible by the point classification, but it still caused some unexpected classifications of points on the ramp close to the wall, see figure 4.9.

The used method gave results that were good enough to be used for the main focus of this thesis. Because of faults in the terrain assessment, some clusters of points that were classified as coverable will be unreachable for the robot. This means that the CPP algorithms will not be able to reach 100% coverage, which is something that needs to be taken into consideration when evaluating these algorithms.

The method included some unwanted hand tuning and a few false classifications. For future work, it would be recommended to find a more reliable method that does not include any biased hand tuning and can handle the scenarios where this method gave incorrect classifications.

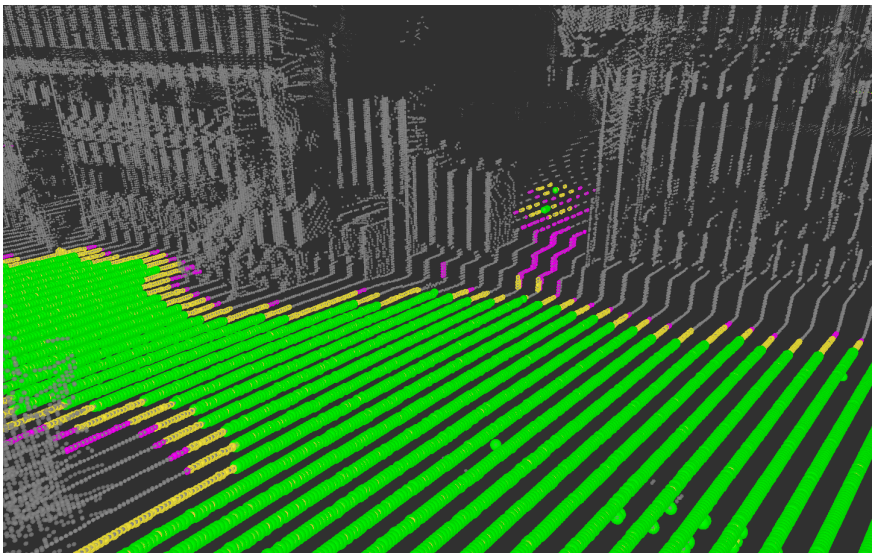


Figure 4.8: Part of the crossing environment showing that the used method does not make sure that all traversable areas are connected. In narrow passages like in this scenario, this creates small clusters of traversable point that are unreachable. Green points are traversable, yellow points are coverable and purple points are inaccessible.

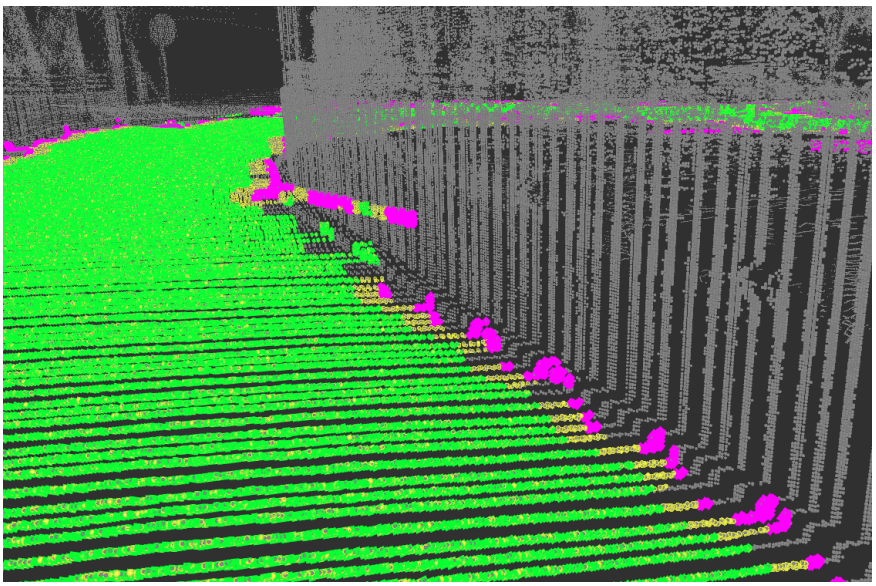


Figure 4.9: Part of the wall of the ramp in the parking garage environment where the terrain assessment did some unexpected classification. A result of classifying the two floors separately without taking the other floor into account. Green points are traversable, yellow points are coverable and purple points are inaccessible.

5 Coverage Path Planning

The main focus of this thesis is coverage path planning (CPP) of large-scale urban environments with multiple floors. This chapter describes implementation details of existing CPP algorithms and theory for a new approach incorporating strengths of existing solutions.

5.1 Environment representation

The main problem with CPP over multiple floors is the representation of the environment.

One approach is to make a 2D plane representation of each floor and to cover them using well known approaches seperatly. This would require floor segmentation, which in scenarios such as a spiral ramp in a parking garage, is not trivial even with hand-tuning. Another disadvantage is that it would make a plan for each floor separately and not take the full path over multiple floors into consideration when optimising the path.

Another approach could be to transform the point cloud into a voxel grid with algorithms such as OctoMap [20]. This method however, transforms multiple points into boxes, which means that information gets lost. Good coverage can not be achieved without having high voxel resolution and paths that seems shortest in the voxel grid can in reality be unfeasible or longer [25].

Taking these arguments into consideration, the representation of the environment in this project was chosen to be a point cloud with classified points. The points were classified as coverable, traversable and inaccessible/Obstacle, see details and definitions in Chapter 4. No discretizations were made and all calculations were directly made on the points in the given point clouds.

5.2 Motion Planner

The motion planning is an important base of many coverage path planning algorithms. It solves the problem of planning the shortest obstacle free path between two given points. The same motion planner was used for all implemented algorithms. It was based on A* described

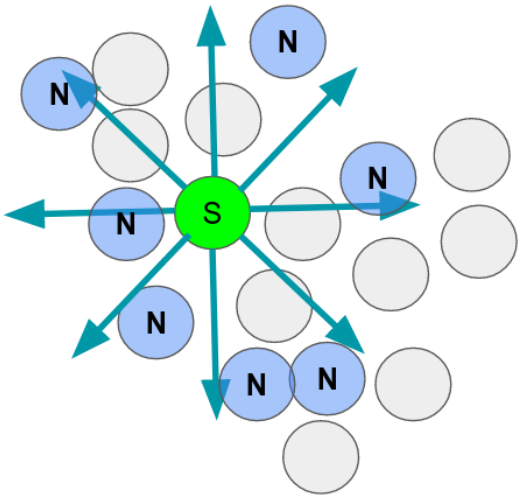


Figure 5.1: Neighbour positions, *N*, of a point *S*, were generated by taking a step in 8 directions from *S* and choosing the closest traversable point. All points in the figure are traversable.

in algorithm 2 to find the shortest path and A*SPT, see algorithm 3 to make the path smoother. If no path was found by the A*, the RRT algorithm, see Algorithm 7 was used. It is more reliable but does not generate the optimal path.

To find neighbour positions of a point is a task that is frequently required in both motion planning and in CPP. The neighbours of a position were given by taking a step in 8 directions from the position and taking the closest traversable point from that spot, see Figure 5.1.

Another task that is common and handled by the Motion Planner is to check the validity of a path between two points. This function assumes that the two points are traversable. The validation is described in detail in algorithm 12. If the closest traversable point is the same as the original point it means that this point is on the border between traversable and untraversable space, and that the path could not be valid. However, if the distance is smaller than the resolution, the step size, the path is traversable. If the distance between two points is bigger, the line-of-sight path between these points is divided into small steps and is only valid if every step ends on a position close to a traversable point, see figure 5.2.

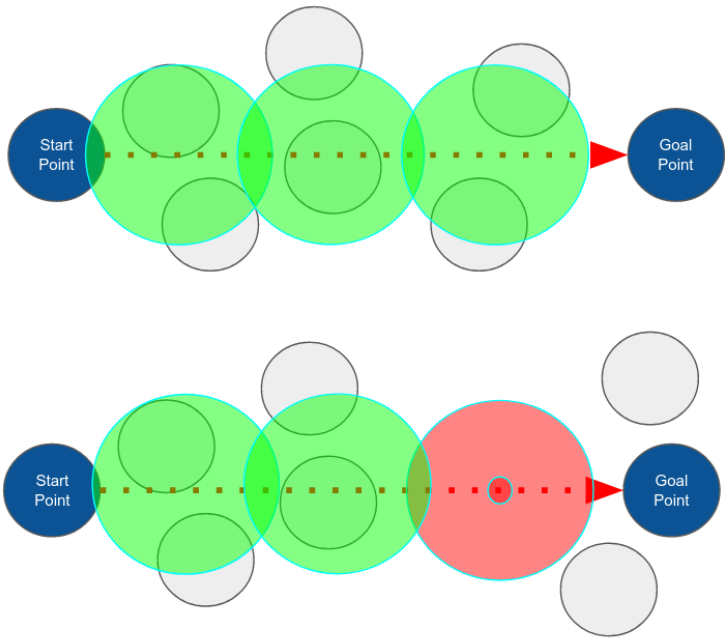


Figure 5.2: An example of a valid (upper illustration) and an invalid (lower illustration) step between two points. The line-of-sight path between the points are divided into three steps. At each step the algorithm calculates the distance to the closest traversable point. If the distance is bigger than the untraversability threshold (see red circle in lower illustration) the path will be classified as invalid.

Algorithm 12: Algorithm that returns if a step between two points is valid or not.

Data: Start point p_s. Goal point p_g. Step size λ. Threshold radius for untraversability r_{trav}.
Result: A boolean, representing whether the path is valid or not.
if p_g is p_s **then**
  return *False*
end
$d_{tot} = |p_g - p_s|$
if $d_{tot} < \lambda$ **then**
  return *True*
end
$N_{steps} = d_{tot}/\lambda$
$\vec{D}$ = Direction vector from p_s to p_g of length λ
for step $i \in 1, 2, ..., N_{steps}$ **do**
  Set $s_i \rightarrow \mathrm{p}_s + i \cdot \vec{D}$
  if Distance to nearest traversable point to $s_i > r_{trav}$ **then**
    return *False*
  end
end
return *True*

5.3 Coverage Path Planner

The focus of this project was to evaluate different CPP algorithms. These algorithms had many common functions that are used to calculate the path that covers the area. The algo-

rithms BA* and Inward Spiral that were described in section 3.3 and 3.4 are based on a two dimensional world representation with tiles. To make them applicable directly on a point cloud and efficient in a multi floor environment following modifications were made.

- Instead of moving between tiles c, the path P consisted of traversable points p.
- To keep track of covered areas, BA* adds tiles to a model and Spiral marks cells as visited. Instead, this was handled by marking all coverable points as *covered* within a radius, r_R from the visited position. When covering a path between two points, positions along the path were visited as well with such frequency that it made sure that all points along the path within the radius were covered.
- A point was classified as *visited* if the distance to the closest point in the visited path P was smaller than a threshold $r_{visited}$.
- Getting neighbour positions is a task required in both BA* and Inward Spiral. Instead of receiving tiles nearby in each of the 4 directions this function returned traversable points in 8 directions, as previously described in Section 5.2.
- A neighbour point was *blocked* if the point was classified as *visited* or if the path to the point was classified as *invalid* by Algorithm 12.

The two algorithms BA* and Inward Spiral has the same structure, but different ways of solving following steps:

1. **CPP Step 1** - Find a path that covers the area until it reaches a dead end. Follow the path.
2. **CPP Step 2** - Find a new starting point.
3. **CPP Step 3** - Find the shortest path from the dead end to the next starting point. Follow the path and go back to CPP Step 1.

5.4 BA* implementation

This section describes details about the implementation of BA* algorithm. The implementation of CPP Step 1 was mostly based on Algorithm 1. Since the number of neighbours was eight instead of four the priority order was set to north, south, northeast, northwest, southeast, southwest, east, west. The used BM algorithm is described in Algorithm 13. CPP Step 2 and 3 were implemented according to Section 3.3, with the modifications described in Section 5.3. An illustration that shows a simplified plan of the implemented BA* is shown in Figure 5.3.

5.5 Inward Spiral implementation

The implementation of the Inward spiral algorithm was based on Section 3.4. Just like in the BA* implementation the number of neighbours were increased and the priority order had to be changed. The priority for the neighbours were set to backwards-right, right, forward-right, forward, forward-left, left, backwards-left. CPP Step 1 was implemented by applying this change of priority and the modifications in Section 5.3 to Algorithm 4. BFS, Algorithm 5, was implemented with the same modifications for the CPP Step 2. For CPP Step 3, the same solution was implemented as for the BA*. Unlike the original description of the Inward Spiral algorithm in Section 3.4, the smoothing of the path, see Algorithm 3, was applied on the A*

Algorithm 13: CPP Step 1 for BA*. A modified BM algorithm (Algorithm 1)

Data: Starting point p_{sp}. Traversable point cloud $\mathcal{W}_{trav}$
Result: Covering path $\mathcal{W}_{local}$
Set $CriticalPointFound \rightarrow false$
$p_{curr} = p_{sp}$
$P_{local} = \varnothing$
while $CriticalPointFound$ is $false$ **do**
  Set $CriticalPointFound \rightarrow true$
  for point $p_n \in \mathcal{W}_{trav}$ in direction $n \in$ north, south, northeast, northwest, southeast, southwest, east, west from p_{curr} **do**
    if p_n is not *blocked* **then**
      P_{local}.push_back(p_n)
      $p_{curr} = p_n$
      Set $CriticalPointFound \rightarrow false$
      break
    end
  end
end
return P_{local}

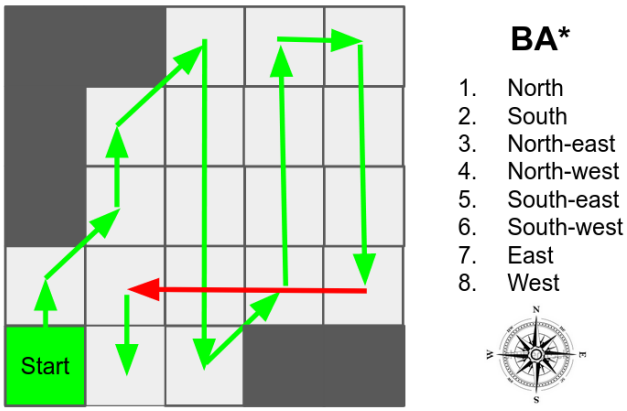


Figure 5.3: An illustration how the BA* algorithm would solve the CPP. Green arrows are the main path that covers new points. When reaching a dead zone, the robot needs to travel to a new uncovered spot. Red arrows are symbolising these movements when no new points are covered.

path in the implementation. An illustration that shows a simplified plan of the implemented Inward Spiral is shown in Figure 5.4.

5.6 Sampled BA* and Inward Spiral

The BA* algorithm generates straight back-and-forth paths across the area, while the Inward Spiral follows the walls of the environment and goes in a spiral motion towards the center. The advantage of BA* path is the straight paths that are optimal for cleaning and works well on big open areas. However, a real world outdoor environment has many obstacles and uneven walls that requires the path to adjust to the environment by following the geometry of the borders. This is better done by the Inward Spiral.

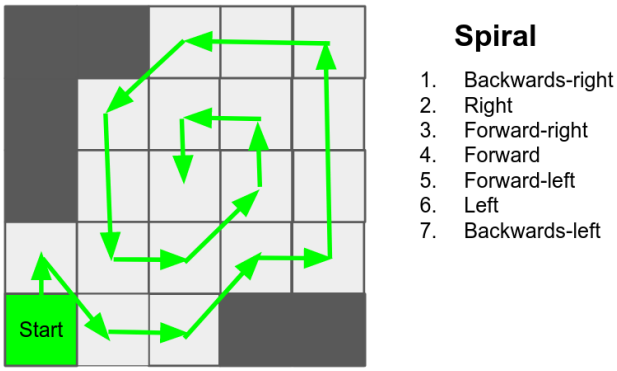


Figure 5.4: An illustration how the Spiral algorithm would solve the CPP. Green arrows are the main path that covers new points.

We propose the novel approach *Sampled BA* and Inward Spiral* to handle covering of environments that consists of both open areas and many obstacles. It is based on the idea of the Sampling-Based Coverage Path Planning algorithm described in Section 3.5. The general idea is to minimize the cost $f(P)$ of path P, defined as

$$f(P) = \text{Length of } P + \text{Total rotation of } P \tag{5.1}$$

by covering open areas with BA* and more complex areas with Inward Spiral. This is done by covering segments of the area with BA* and Inward Spiral starting from random sampled edge points and then connect these segments with a Traveling Salesman algorithm [16]. A random edge point is found by sampling a random uncovered point, check if it is accessible from the starting point using RRT (see Algorithm 7) and then find the closest inaccessible point using BFS (modified Algorithm 5).

The new algorithm Sampled BA* & Inward Spiral , see Algorithm 14, follows these steps:

1. Initialise a list of detached path segments S.
2. **PART I**: For $N_\phi = 4$ different angles ϕ_i, define *north* as the angle ϕ_i and start covering using the BA* algorithm (Section 5.4) starting from a traversable edge point of a randomly sampled uncovered area. This is illustrated in Figure 5.5a. Stop covering when the distance to the next starting point, given by CPP Step 2, exceeds d^1_{max}.
3. If none of these N_ϕ paths had a cost $f(P)$ per coverage C (Equation 5.1 and 3.1) that were lower than C^1_{min}, set the points covered by the path with the biggest coverage as *explored*. Otherwise, choose the path with the lowest cost per coverage. Add it to S and set its covered points as *explored* and *covered*.
4. Repeat Step 2-3 until E^1 of the points are *explored*.
5. **PART II**: Start covering using the Inward Spiral algorithm (Section 5.5) starting from a traversable edge point of a randomly sampled uncovered area, see Figure 5.5b. Stop covering when the distance to the next starting point, given by CPP Step 2, exceeds d^2_{max}. Set the points that were covered by the path as *explored*. If the cost per coverage is lower than C^2_{min}, add the path to S and set its covered points as *covered*.
6. Repeat Step 5-6 until C of the points are *covered* or all points have been *explored*.

7. **PART III**: Add the start and end positions of the paths in S as nodes to a graph tree T. All nodes are connected with edges with weights

   - $w = 0$, if the edge is between the start and end node of the same path in S.
   - Otherwise, distance based according to equation

   $$w = w_{\text{offset}} + |s_A(x,y) - s_B(x,y)| + K|s_A(z) - s_B(z)|$$

   where s_A and s_B are the position of the two nodes, w_{offset} is a large number and $K \geqslant 1$.

   The purpose of w_{offset} is to make sure that the traveling salesman algorithm in the following step always chooses to connect corresponding start and end nodes. Since the environment could have multiple floors, extra weight K is added to difference in height, to avoid potential movement between floors.

8. Solve the TSP problem [16] of the cheapest visitation order, P_{TSP}, to visit all nodes in T (see Figure 5.5c).

9. Walk through every node in P_{TSP} and create an ordered list of paths S_{TSP}. For every node s_i,

   - If s_i is start node, add corresponding path to S_{TSP}.
   - If s_i is an end node, add the corresponding path, but reversed, in S to S_{TSP}.

10. Create a continuous path P by following the paths in S_{TSP}. Connect them with obstacle free smooth paths as in CPP Step 3 (same in Sections 5.4 and 5.5). This is illustrated in Figure 5.5d.

5.7 Parameters

All algorithms mentioned in this chapter has multiple parameters that had to be tuned. They can be hand tuned or systematically tuned using Bayesian optimization, see Section 3.6.

Visited threshold, $r_{visited}$ and Step size, λ_{CPP} are common parameters for all algorithms and to set their value is a general problem for CPP. Description of these parameters and theoretical motivation behind their boundary values in the tuning are described below.

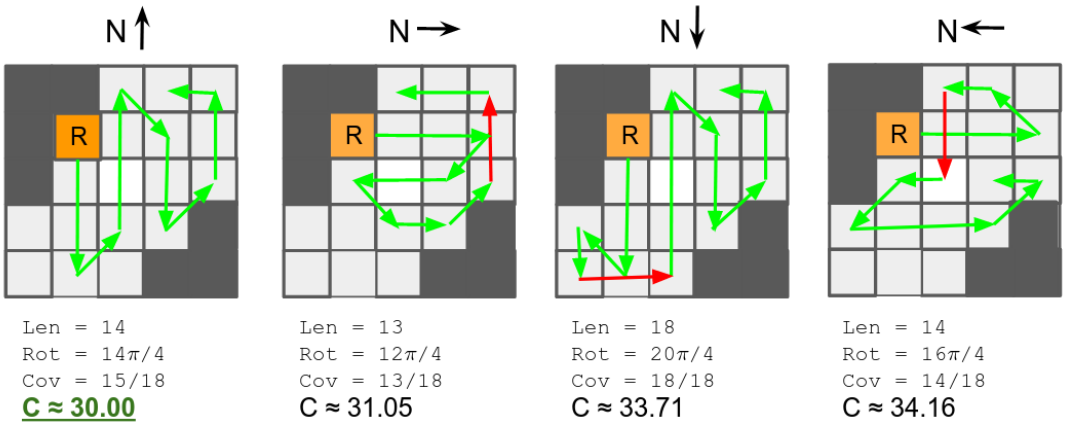
In addition to λ_{CPP} and $r_{visited}$, BA* algorithm has a parameter representing the direction angle defining *north*. The parameters of Sampled BA* & Inward Spiral are presented in Table 5.1.

CPP Step size, λ_{CPP}

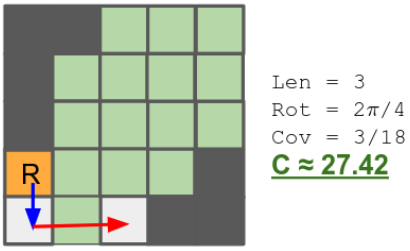
The parameter CPP Step size λ_{CPP} defines the length of each step to a neighbour, see blue arrows in Figure 5.1. A too small λ_{CPP} leads to multiple coverage of the same area. A too big λ_{CPP} results in gaps between paths, which needs to be visited later and makes the covering path less effective.

The value of the parameter was tuned with boundaries based on the covering radius range of the robot, r_R. It is based on the two scenarios in Figure 5.6. The λ_{CPP} in the scenario in Figure 5.6a, shows the step size that would guarantee full coverage for every step without gaps.

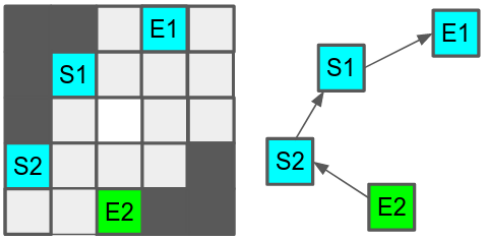
$$\lambda_{CPP}^{min} = \sqrt{2} \cdot r_R \approx 1.41 r_R \tag{5.2}$$



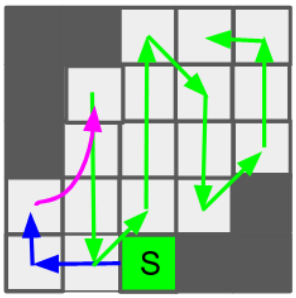
(a) For different angles ϕ_i, define north N as the angle ϕ_i and start covering using the BA* starting from random sampled edge point R. Keep covering until the distance to the next starting point is too big. Choose the path with the lowest cost per coverage $C = (Len + Rot)/Cov$.



(b) Start covering using the Inward Spiral algorithm starting from random sampled uncovered edge point R. Green boxes in the figure are covered by the best path from BA*, see the first path from left in Figure 5.5a.



(c) Apply a traveling salesman algorithm to calculate the cheapest route to visit all start and end points of path. Since the weight between start and end node pairs is set to 0 they will always be connected. $S1$ and $S2$ are start nodes, $E1$ and $E2$ are end nodes of the paths in the segment list S. $E2$ is also the start position of the total path.



(d) The final path. The robot follows all paths according to the cheapest route and connects them using smooth A*, see purple arrow. Since $E2 \rightarrow S2$ starts from and end node, the path is reversed to the path in 5.5b

Figure 5.5: Simplified illustration of the Sampled BA* & Inward Spiral algorithm. Dark boxes are untraversable. Green arrows shows paths generated by BA*, blue arrows by Inward Spiral and purple arrows by the Motion Planner. Red arrows is parts of the paths as well, but indicates that robot moves over already covered area.

Algorithm 14: Sample Based BA* & Inward Spiral.

Data: Starting position p_s. $P = \varnothing$. $S_{\text{TSP}} = \varnothing$.
Result: Path P to cover the area.
$S_{\text{BA*}} \leftarrow$ Sample BA* path segments (Algorithm 15)
$S_{\text{IS}} \leftarrow$ Sample Inward Spiral path segments (Algorithm 16 with segments in $S_{\text{BA*}}$ set as covered)
$S \leftarrow S_{\text{BA*}} \cup S_{\text{IS}}$
$T \leftarrow$ Empty graph tree
for path P_i in S **do**
  Add start and end point of P_i as nodes in T.
end
$P_{\text{TSP}} \leftarrow$ Cheapest route to visit all nodes in T
$p_{curr} \leftarrow p_s$
for node p_i in P_{TSP} **do**
  if $p_i = p_{curr}$ **then**
    continue
  end
  if p_i is a start node **then**
    $P_i \leftarrow$ Path in S with p_i as start node
  else
    $P_i \leftarrow$ Reversed path in S with p_i as end node
  end
  $S_{\text{TSP}} \leftarrow S_{\text{TSP}} \cup P_i$
  $p_{curr} \leftarrow$ Last point in P_i
end
$p_{curr} \leftarrow p_s$
for path P_i in S_{TSP} **do**
  $P_{A*} \leftarrow$ Shortest path from p_{curr} to the start of P_i.
  $P \leftarrow P \cup P_{A*} \cup P_i$
  $p_{curr} \leftarrow$ Last point in P_i
end
return P

Algorithm 15: PART I: Sample BA*

Data: $S = \varnothing$.
Result: Set of paths S covering local regions.
while exploration goal E^1 has not been reached **do**
  s_r = random position in uncovered area
  p_r = closest edge point to s_r given by BFS
  $S_{BA*} = \varnothing$
  for angle $\phi \in [0, 1, ..., N_\phi] \cdot 2\pi / N_\phi$ **do**
    while distance to new starting point $< d^1_{max}$ **do**
      $P_{BA*} \leftarrow$ BA* from p_r with north $= \phi$
    end
    $S_{BA*} = S_{BA*} \cup P_{BA*}$
  end
  $S_{BA*_{accepted}} \leftarrow P_{BA*} \in S_{BA*}$ with $\frac{\text{cost}}{\text{coverage}} > C^1_{min}$
  if $S_{BA*_{accepted}} = \varnothing$ **then**
    Explore $P_{BA*} \in S_{BA*}$ with max coverage
  else
    $P_{best} \leftarrow P_{BA*} \in S_{BA*}$ with lowest $\frac{cost}{coverage}$
    Explore and cover P_{best}
    $S = S \cup P_{best}$
  end
end
return S

Algorithm 16: PART II: Sample Inward Spiral

Data: $S = \varnothing$.
Result: Set of paths S covering local regions.
while goal coverage C or exploration = 1 has not been reached **do**
  s_r = random position in uncovered area
  p_r = closest edge point to s_r given by BFS
  while distance to new starting point $< d^2_{max}$ **do**
    $P_{Spiral} \leftarrow$ Inward Spiral from p_r
  end
  Explore P_{Spiral}
  if $\frac{cost}{coverage}$ of $P_{Spiral} > C^2_{min}$ **then**
    $S \leftarrow S \cup P_{Spiral}$
    Cover P_{Spiral}
  end
end
return S

Table 5.1: Sampled BA* & Inward Spiral parameters.

Parameter	Description
E^1	Exploration goal for PART I
C	Total coverage goal
d^1_{max}	Max distance to next starting point in PART I
d^2_{max}	Max distance to next starting point in PART II
C^1_{min}	Min cost (length + rotation) per coverage in PART I
C^2_{min}	Min cost (length + rotation) per coverage in PART II
λ_{CPP}	Step size
$r_{visited}$	Visited threshold

is the lower boundary of the step size, since lower step size would result unnecessary coverage of same points. The upper boundary

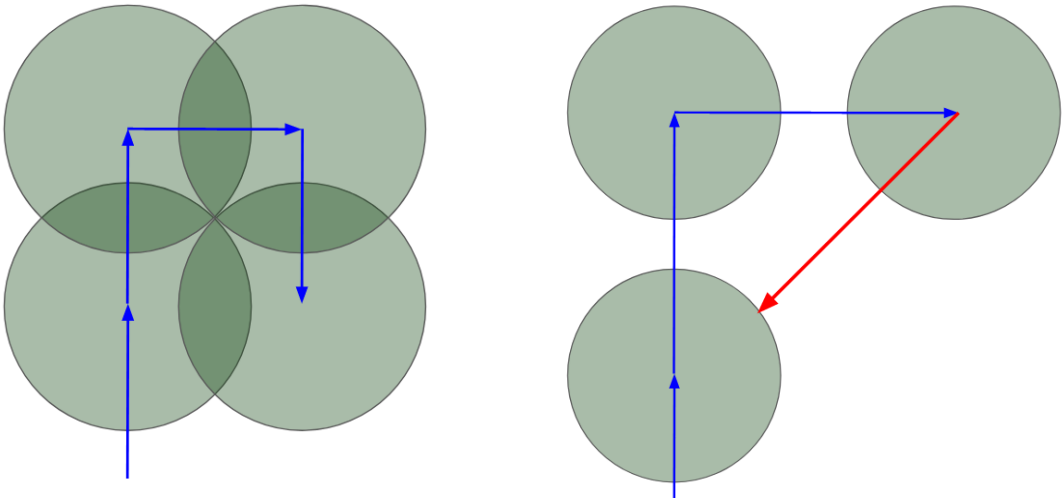
$$\lambda^{max}_{CPP} = (\sqrt{2}+1) \cdot r_R \approx 2.41 r_R \tag{5.3}$$

is based on the scenario in Figure 5.6b. Bigger step size would make the last step (red arrow in Figure 5.6b) to end up in an already covered area, which would result in ineffective coverage.

CPP Visited threshold, $r_{visited}$

CPP Visited threshold $r_{visited}$, is a radius that defines if a position has been visited by the robot. If this radius is too small, a lot of points will be covered multiple times, which makes the path longer and the algorithm less effective. If it is too big, it will be harder for the algorithms to find valid points to go to and consequently miss to cover spots because of inaccessibility.

The radius $r_{visited}$ was set based on the step size and three different scenarios, see Figure 5.7. It is desired to allow the algorithm to make paths with straight lines side by side. Therefor, $r_{visited}$ should be small enough to allow the scenarios in Figure 5.7a and 5.7c. However, $r_{visited}$ needs to be big enough to not accept the scenario in Figure 5.7b. The distance between two side-by-side paths depends on the third step in the figures. If the distance is big, gaps between the circles allows the algorithm to take steps between the paths, which is unwanted. Therefor, $r_{visited}$ was calculated to make the scenario in Figure 5.7a, not allowing any gaps between the side-by-side paths. This value is big enough to not allow the scenario in Figure 5.7b, but small



(a) North → North → East → South. Step size $\lambda_{CPP}^{min} = \sqrt{2} \cdot r_R \approx 1.41 r_R$.

(b) North → North → East ↛ South-west. Step size $\lambda_{CPP}^{max} = (\sqrt{2}+1) \cdot r_R \approx 2.41 r_R$.

Figure 5.6: The two different scenarios that was used to define the boundaries of the CPP Step size, λ_{CPP}. Green circles shows the covering range of the robot at each step.

enough to allow the scenario in Figure 5.7c. Given a step size λ, $r_{visited}$ is given by

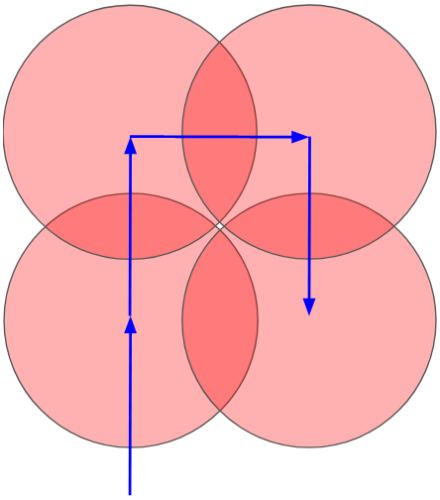
$$r_{visited} = \frac{\lambda}{\sqrt{2}} \approx 0.71\lambda \tag{5.4}$$

The visited threshold needs to be tuned as well, meaning a lower and an upper boundary is needed. The visited threshold has to be lower than the step size to be able to make a step in any direction. Consequently, the upper boundary $r_{visited}^{max}$ was set to the lower boundary of the CPP step size (Equation 5.2),

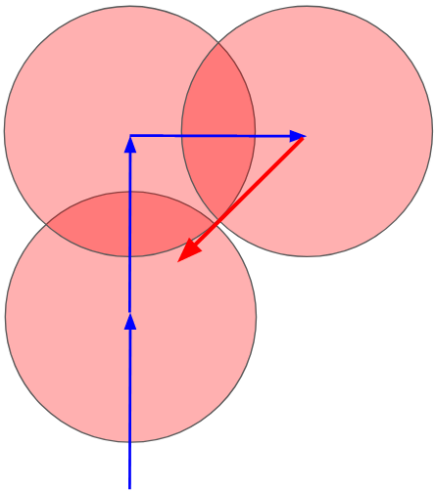
$$r_{visited}^{max} = \sqrt{2} r_R. \tag{5.5}$$

The lower boundary was set based on Equation 5.4 for the lower boundary of the step size. The lowest boundary was set to one standard deviation away from this value in a normal distribution resulting in

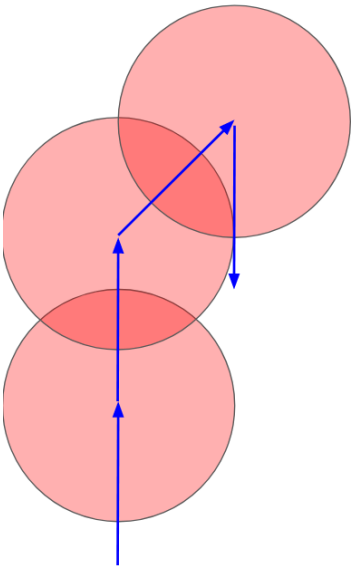
$$r_{visited}^{min} = \frac{\lambda}{\sqrt{2}}(1-\sigma) = \frac{\sqrt{2} r_R}{\sqrt{2}}(1-0.341) = 0.659 r_R \tag{5.6}$$



(a) North → North → East → South. Should be accepted and not leave any gaps.



(b) North → North → East ↛ South-west. Should not be accepted.



(c) North → North → North-east → South. Should be accepted.

Figure 5.7: The three different scenarios that was used to define the CPP Visited threshold $r_{visited}$.

6 Method

This chapter contains details about the method that were used to evaluate the CPP algorithms that were described in chapter 5.

6.1 Terrain Assessment

Before evaluating the CPP algorithms, the point clouds on which they would operate on, had to be prepared. Given a point cloud, all traversable positions for the robot and all coverable points had to be found. The used method and the results of the Terrain Assessment are described in Chapter 4.

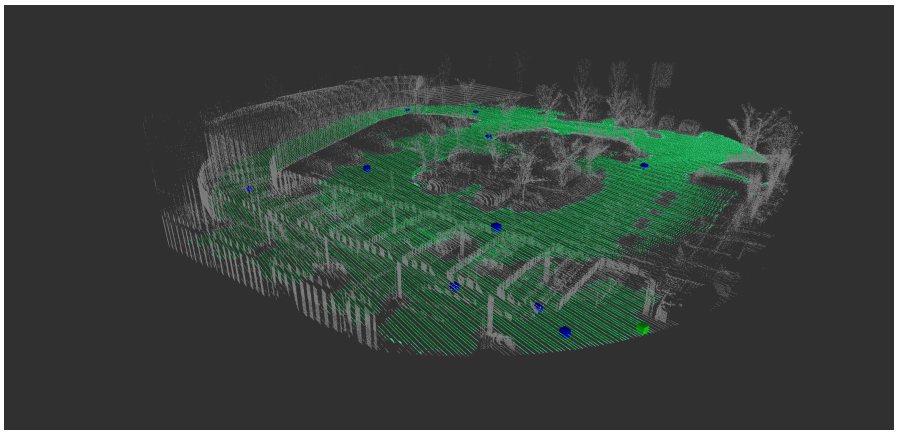
6.2 Experiment 1 - Optimize path for one starting point

The goal of the first experiment was to find the best path among the three algorithms BA*, Inward Spiral and Sampled BA* & Inward Spiral (Sections 5.4-5.6). Coverage path planning was solved for three different environments, see Figure 6.1 from a given starting point.

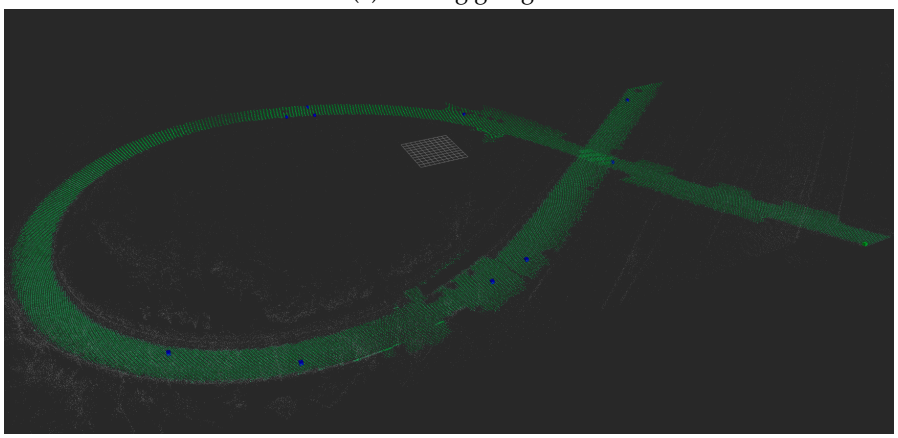
For each algorithm, optimization of the parameters was done using HyperOpt [3], see Section 3.6. The loss function $f(P)$ to minimize was the sum of the total path length and total accumulated path rotation (Equation 5.1) and the number of evaluations N was set to 100. The algorithms planned paths until they reached 95% coverage or the computational time limit of 250 seconds, whichever was reached first.

6.3 Experiment 2 - Best configuration for other starting points

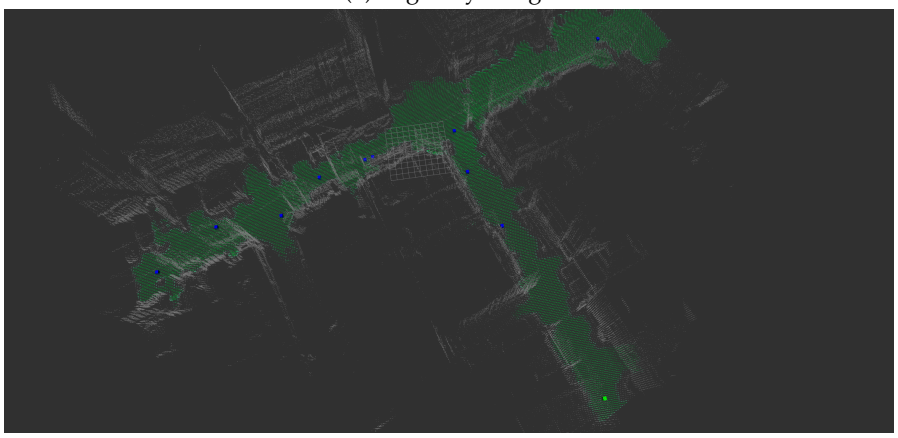
The goal of the second experiment was to compare the three algorithms with their optimal configurations given from Experiment 1 for other starting points. For each environment, ten random points were sampled, see Figure 6.1. For each point, plans were made using the three algorithms. Plans were made until maximum coverage or the computational time limit of 400 seconds were reached. Coverage, total rotation and length of path was calculated at different times during the planning. For BA* and Inward Spiral, calculations were done every



(a) Parking garage



(b) Highway bridge



(c) City crossing

Figure 6.1: Environments that were used. Green area is coverable, grey is inaccessible. Light green marker is the given starting point used in Experiment 1. Blue markers are the 10 randomly sampled starting points that were used in Experiment 2.

time they reached a dead end. For Sampled BA* & Inward Spiral , every tenth percentage coverage. At each calculation, Part III of the algorithm (see Section 5.6) was applied on the segments in S from PART I and PART II. Mean and 95% confidence interval over the starting points was computed for every time and coverage sample.

6.4 Properties

All values presented in the results are based on a path P and a coverable point cloud $\mathcal{W}_{cov}$. Both P and $\mathcal{W}_{cov}$ consist of three dimensional points p.

Length of path

Length of path, l_P was calculated by taking the sum of the euclidean distances between the N points in the path, P, when visiting them in order, i.e.

$$l_P = \sum_{i=1}^{N} |p_i - p_{i-1}| \tag{6.1}$$

Total rotation

Total rotation is the sum of the differences in the yaw angle between the vector into each point and out of each point. After projecting every point in path P on a 2D plane, the total rotation, ϕ_P, was calculated with

$$\phi_P = \sum_{i=2}^{N} |\arccos\left(\frac{p_i - p_{i-1}}{|p_i - p_{i-1}|} \cdot \frac{p_{i-1} - p_{i-2}}{|p_{i-1} - p_{i-2}|}\right)| \tag{6.2}$$

Coverage

Coverage is the percentage of coverable points that has been covered. A point has been covered if it is within the range radius of the robot r_R from any point in path P. If the distance between two points was bigger than step size λ_{cov}^R, steps were taken along the path while covering points within the range at each step, see algorithm 17.

Algorithm 17: Calculating Coverage

Data: Path P, Coverable point cloud $\mathcal{W}_{cov}$
Result: Coverage C
$\mathcal{W}_{covered} = \varnothing$
for point $p_i \in \mathcal{W}_{cov}$ **do**
  if $|p_i - p_{i-1}| > \lambda_{cov}^R$ **then**
    P_{sub} = Points along the line from p_i to p_{i-1} with spacing λ_{cov}^R
  else
    $P_{sub} = p_i$
  end
  for point $p \in P_{sub}$ **do**
    $\mathcal{W}_{covered} \leftarrow$ all points in $\mathcal{W}_{cov}$ within radius r_R from p
  end
end
return $|\mathcal{W}_{covered}|/|\mathcal{W}_{cov}|$

7 Results

Results from the experiments and used parameter values are presented in this chapter. Figure 7.1 shows paths in the parking garage environment from simulations of the implemented CPP algorithms.

7.1 Common Parameters

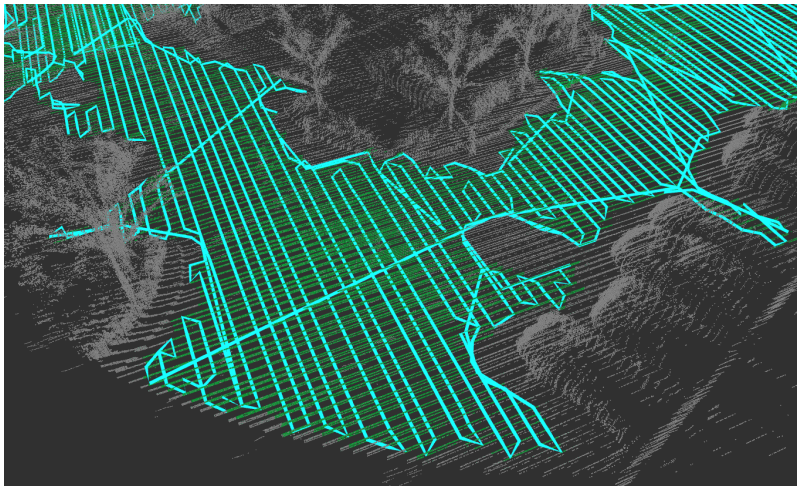
This section presents the values of the common parameters that were used. Same robot geometry and motion planner was used in all three algorithms. Parameters related to the robot are presented in Table 7.1. Parameters for the Motion Planner are presented in Table 7.2.

Table 7.1: Robot parameters. Values of *Real data* parameters are based on data from the real world provided by the developers of the robot. Values of *Resolution* parameters are minimised to give a good resolution while keeping the computational time on a reasonable level.

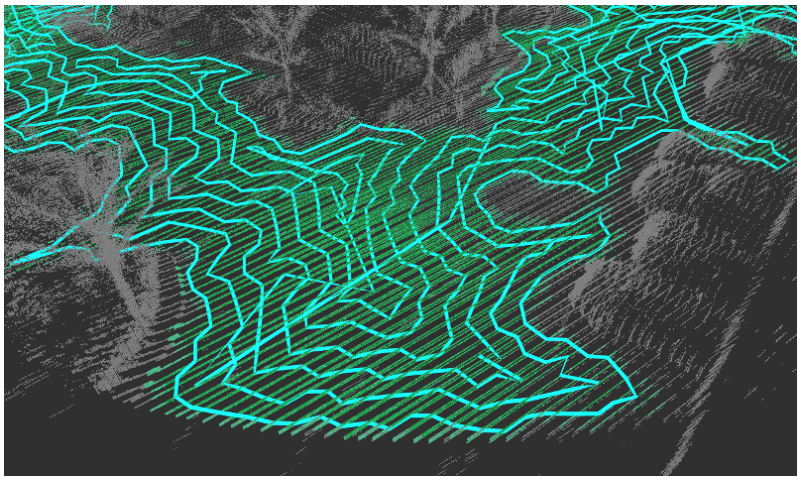
Parameter	Value	Explanation	Based on
r_R	0.375 m	Coverage radius	Real data
λ_{cov}^R	0.2 m	Coverage step size	Resolution

Table 7.2: Motion Planner parameter. Values of *Resolution* parameters are set to give a good resolution while keeping the computational time on a reasonable level. *Hand tuned based on tests* parameters are hand tuned by making multiple tests and adjust the values to give plausible reliance, computational time and paths.

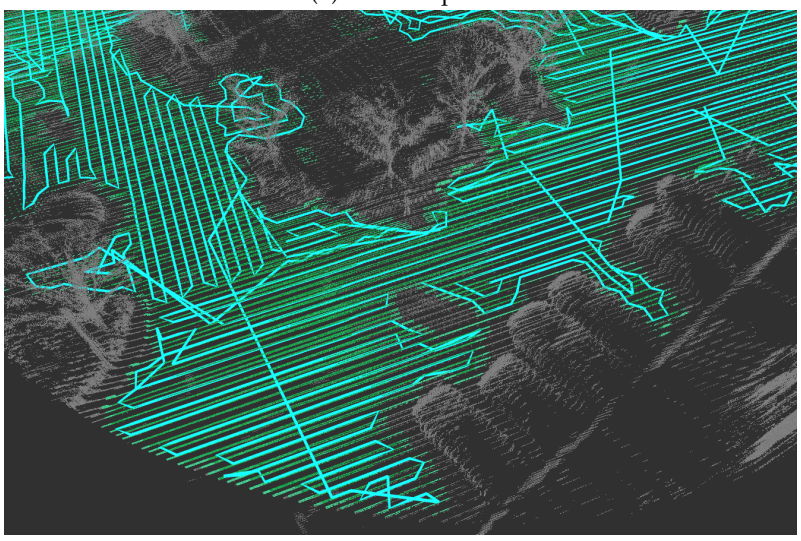
Parameter	Value	Explanation	Based on
λ	0.1 m	Step size	Resolution
λ_{A*}	0.5 m	Step size used in A*	Hand tuned b.o. tests
λ_{RRT}	0.3 m	Step size used in RRT	Hand tuned b.o. tests
r_{trav}	0.2 m	Threshold for untraversability	Hand tuned b.o. tests
N_{max}^{RRT}	10 000	Max iterations in RRT	Resolution



(a) BA*



(b) Inward Spiral



(c) Sampled BA* & Inward Spiral

Figure 7.1: Coverage paths on the parking garage environment.

Table 7.3: CPP parameters optimized using HyperOpt on given start position.

BA*				
Parameter	**Parking garage**	**Highway bridge**	**City crossing**	**Prior**
λ_{CPP}	0.634m	0.557m	0.524m	$\mathcal{U}(0.5, 1)$
$r_{visited}$	0.473m	0.453m	0.404m	$\mathcal{U}(0.25, 0.5)$
ϕ	4.65 rad	5.34rad	4.70 rad	$\mathcal{U}(0, 2\pi)$
Inwards Spiral				
Parameter	**Parking garage**	**Highway bridge**	**City crossing**	**Prior**
λ_{CPP}	0.628m	0.737m	0.665m	$\mathcal{U}(0.5, 1)$
$r_{visited}$	0.499m	0.483m	0.500m	$\mathcal{U}(0.25, 0.5)$
Sampled BA* & Inwards Spiral				
Parameter	**Parking garage**	**Highway bridge**	**City crossing**	**Prior**
E^1	0.865	0.940	0.863	$\mathcal{U}(075, 0.95)$
d^1_{max}	4.13m	4.49m	1.69m	$\mathcal{U}(1, 5)$
d^2_{max}	6.94m	7.06m	4.48m	$\mathcal{U}(4, 10)$
C^1_{min}	8238	12773	6488	*
C^2_{min}	13645	25989	8141	**
λ_{CPP}	0.548m	0.619m	0.554m	$\mathcal{U}(0.5, 1)$
$r_{visited}$	0.373	0.389m	0.258m	$\mathcal{U}(0.25, 0.5)$

* Based on coverable area A: $\mathcal{U}(0.165A, 0.4A)$
** Based on coverable area A: $\mathcal{U}(0.4A, 0.66A)$

Table 7.4: Result of the best found solution for each environment and algorithm

Environment	**Algorithm**	**Cost**	**Length [m]**	**Rotation [rad]**
Parking garage	Sampled BA* & Inward Spiral	**4077**	2999	**1078**
	BA*	4238	**2896**	1342
	Inwards Spiral	5787	3101	2686
Highway bridge	Sampled BA* & Inward Spiral	**5797**	4643	**1154**
	BA*	6385	4562	1823
	Inwards Spiral	7213	**4440**	2773
City crossing	Sampled BA* & Inward Spiral	**3001**	**2186**	**815**
	BA*	3262	2249	1013
	Inwards Spiral	4054	2239	1815

7.2 Experiment 1 - Optimize path for one starting point

Results of Experiment 1 are presented in the Tables 7.3 and 7.4. Sampled BA* & Inward Spiral consistently performs significantly better on cost than the others. Primarily due to lower rotation in the Parking garage and Highway bridge environments, but on the single-level City crossing environment it provided both the shortest length and lowest total rotation.

7.3 Experiment 2 - Best configuration for other starting points

Figures 7.2-7.4 shows sample values from Experiment 2 for Coverage, Length of path and Total rotation over computational time. Performance of the algorithms from Experiment 2 is presented in the Figure 7.5.

Computational time

The computational time efficiency were evaluated by looking at the gradient of the graphs in Figures 7.2a, 7.3a and 7.4a. For all three environments Inward Spiral graph is above the

Table 7.5: Values at 95% coverage in Figure 7.5 from Experiment 2 compared with values from Experiment 1.

Environment	Algorithm	Exp.1 Cost	Exp.2 Cost	Increase [%]
Parking garage	Sampled BA* & Inward Spiral	**4077**	4610	13
	BA*	4238	**4361**	3
	Inwards Spiral	5787	5874	2
Highway bridge	Sampled BA* & Inward Spiral	**5797**	**6375**	10
	BA*	6385	6480	1
	Inwards Spiral	7213	8745	21
City crossing	Sampled BA* & Inward Spiral	**3001**	**3121**	4
	BA*	3262	3407	4
	Inwards Spiral	4054	4225	4

others, making it clear that Inward Spiral is the fastest, followed by BA* and that Sampled BA* & Inward Spiral is the slowest.

Maximum coverage

The maximum reached coverage for each algorithm varied for different environments. For the Parking garage environment, Sampled BA* & Inward Spiral reached the highest coverage, slightly higher than Inward Spiral (See Figure 7.2b). For the Highway bridge, it was the other way around (See Figure 7.3b) and for the City crossing, BA* reached higher coverage than the others (See Figure 7.4b). Inward Spiral reached a coverage that was 1% higher than BA* in the Parking garage and 3% higher in the Highway Bridge.

Length of path

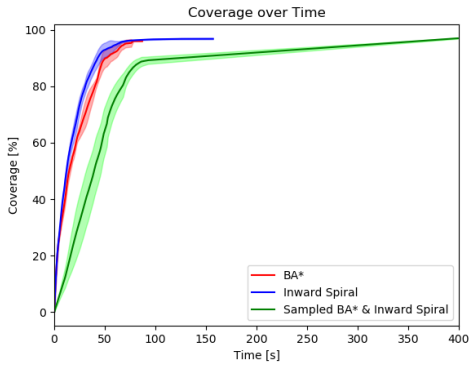
The effectiveness regarding length of the path was fairly the same for Inward Spiral and BA* for low coverage rates. However, when reaching higher coverage percentages the Inward Spiral algorithm becomes ineffective. This can be observed in the Figures 7.2-7.4. The length for Inward Spiral keeps increasing much more than the coverage for higher coverage values. This can also be seen in Figure 7.5 where the cost increases a lot for the last coverage percentages compared to the other algorithms.

Total Rotation

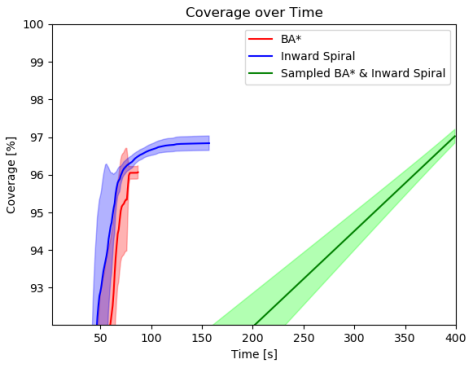
The total rotation was significantly higher for Inward Spiral compared to other algorithms, see Figures 7.2d, 7.3d and 7.4d. The results did also show that the biggest advantage of the Sampled BA* & Inward Spiral algorithm over BA* is that it's total path requires less rotation. It was consistently lower in all environments.

Robustness

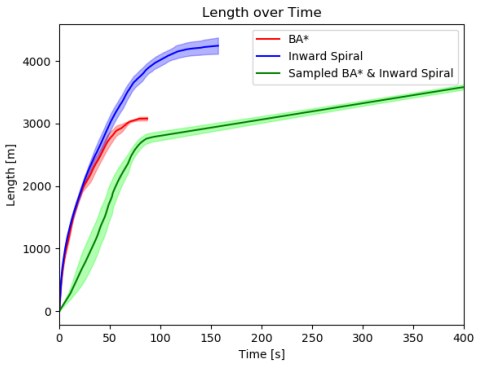
In general, the optimal configurations in Experiment 1 seems to give similar results for other starting points in Experiment 2. Figure 7.5 shows that the Sampled BA* & Inward Spiral is more cost effective than the other algorithms in all environments except for the parking garage. When comparing the costs around 95% coverage with the costs in Experiment 1, see Table 7.5, all costs were slightly higher. BA* was close to the values in Experiment 1 for all environments. Inward Spiral had some lack of robustness in the bridge environment, where the cost in Experiment 2 were much higher than in Experiment 1. In the city crossing the cost were close to the results in Experiment 1, a bit higher in the highway bridge, and significantly higher in the parking garage environment.



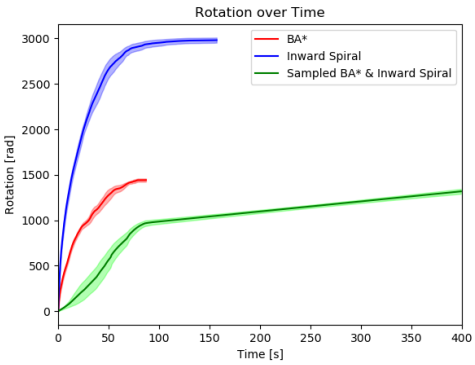
(a) Coverage over computational time



(b) Coverage over computational time. Zoomed view.

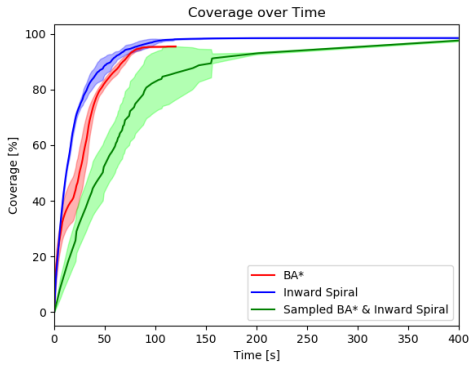


(c) Length over computational time

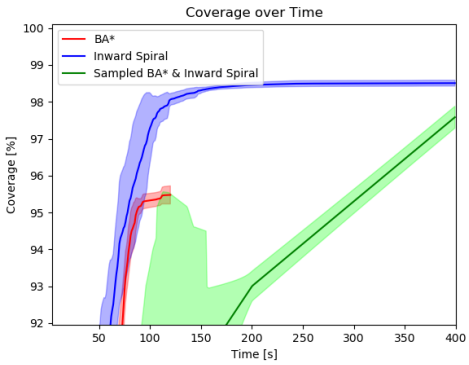


(d) Rotation over computational time

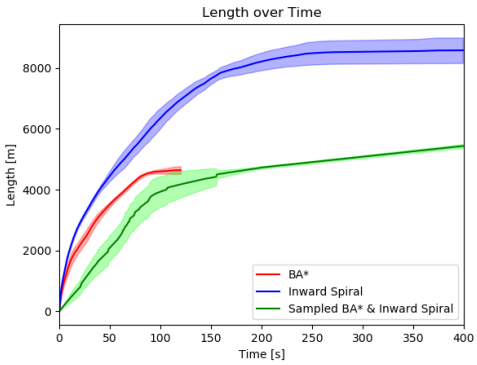
Figure 7.2: Results over computational time from Experiment 2 for Parking garage environment. A path is better if it has higher values in (a) and (b), but lower values in (c) and (d). The graphs in the figures stops when the maximum reachable coverage is reached.



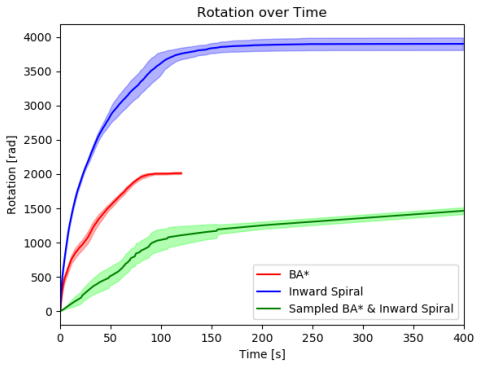
(a) Coverage over computational time



(b) Coverage over computational time. Zoomed view.

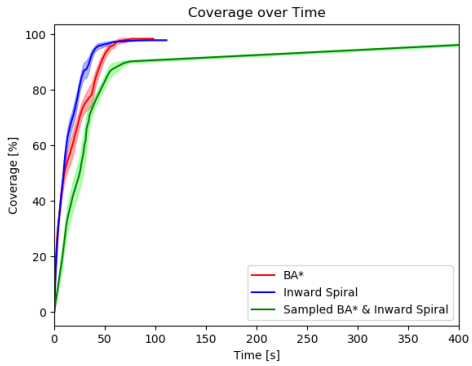


(c) Length over computational time

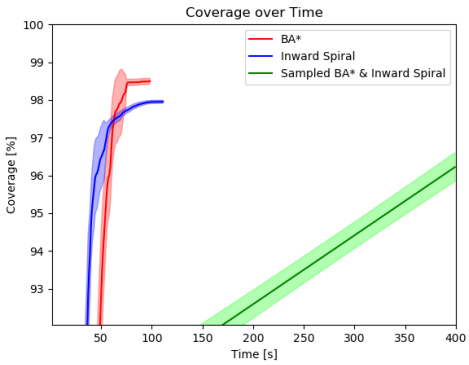


(d) Rotation over computational time

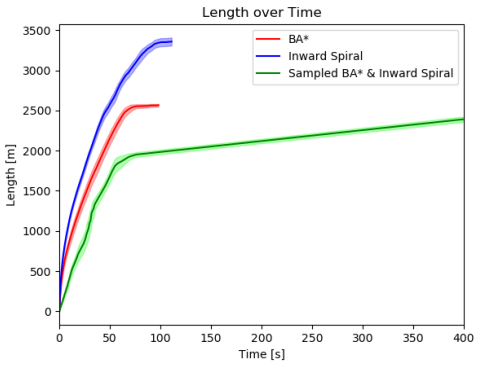
Figure 7.3: Results over computational time from Experiment 2 for Highway bridge environment. A path is better if it has higher values in (a) and (b), but lower values in (c) and (d). The graphs in the figures stops when the maximum reachable coverage is reached.



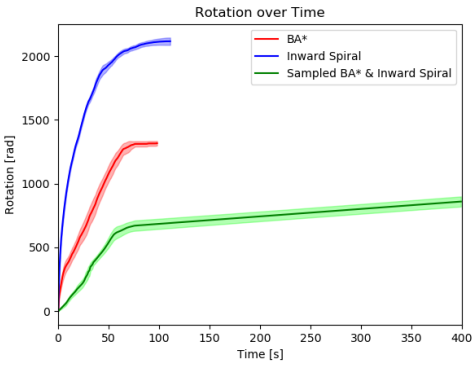
(a) Coverage over computational time



(b) Coverage over computational time. Zoomed view.

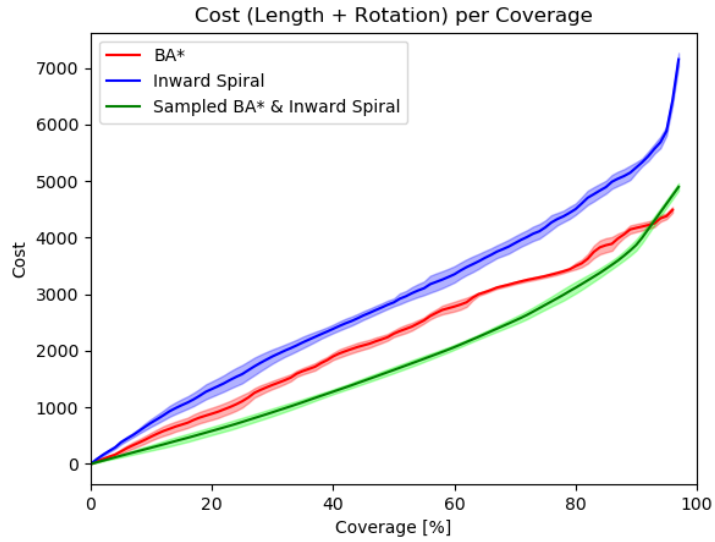


(c) Length over computational time

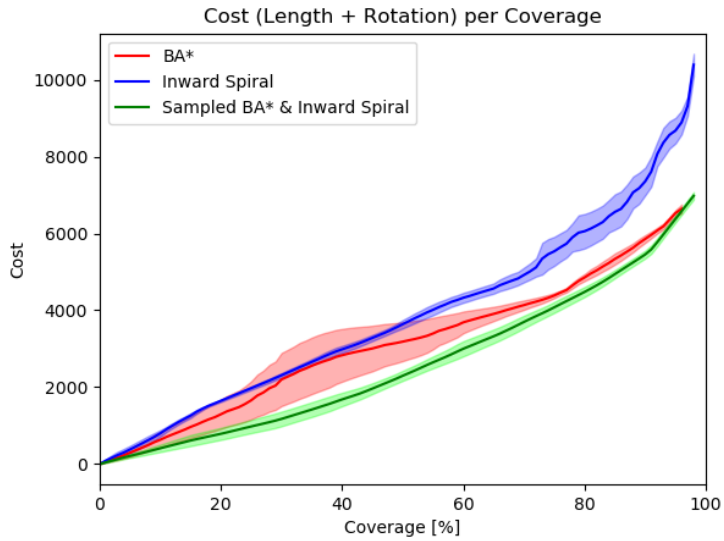


(d) Rotation over computational time

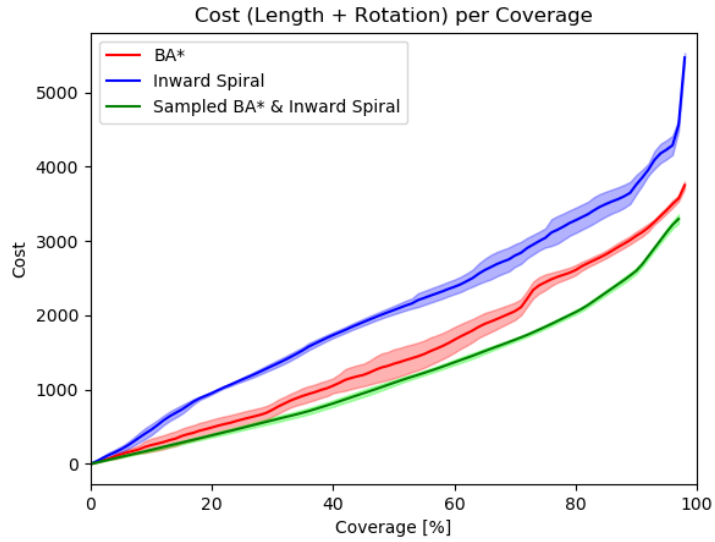
Figure 7.4: Results over computational time from Experiment 2 for City crossing environment. A path is better if it has higher values in (a) and (b), but lower values in (c) and (d). The graphs in the figures stops when the maximum reachable coverage is reached.



(a) Parking Garage



(b) Highway Bridge



(c) City Crossing

Figure 7.5: Performance results in Experiment 2. Mean and 95% confidence interval over 10 random staring locations.

8 Discussion

The focus of this thesis was offline footprint-based CPP that could handle complex multi-floor large-scale environments. Since CPP is NP-hard it is difficult to even find reasonably good feasible solutions, but many algorithms have been developed to solve this problem.

In this work, two well known algorithms have been compared with a new approach in three different scenarios. Making all calculations offline made it possible to optimize the solutions by tuning the parameters of the algorithms to get the best output. An online approach does not require prior knowledge about the environment and would be able to handle changes in the environment. However, it would not be able to optimise the path over the full environment and would most likely be less efficient.

In this chapter results and method are discussed. It also contains some words about the work in a wider context.

8.1 Results

After doing the Terrain Assessment on the environments, see Chapter 4, two experiments were made according to Sections 6.2 and 6.3. The paths shown in Figure 7.1 looks as expected. BA* generates straight back-and-forth paths with movement between dead ends and new starting points. Inward Spiral generated curved paths that follows the borders of the environment. Sampled BA* & Inward Spiral, covered open areas with back-and-forth paths with the optimal driving angle and complex parts of the environment with Inward Spiral paths. Results from the experiments are discussed below.

Experiment 1

The goal of Experiment 1 was to find the best solution that would cover 95% of the area within the time limit. As shown in Table 7.4, Sampled BA* & Inward Spiral gave consistenly better results for each of the three environments. Using HyperOpt, solution paths with less rotation were found and for the Crossing environment a path that was shorter as well. Since

all parameters are automatically optimised, the performance of each algorithm is optimised for the specific environment.

The parameters that gave the best result are presented in Table 7.3. The prior boundaries of λ_{CPP} and $r_{visited}$ are set based on calculations in Section 5.7 with a small margin. In general the boundaries seemed to be reasonable. However, for Inward Spiral, the final parameter value of $r_{visited}$ was consistently very close to the upper border. Possibly, better results could have been achieved with higher upper boundary.

It is unexpected that the optimal $r_{visited}$ for Sampled BA* & Inward Spiral differed quite a lot from the other algorithms, since the same kind of paths are used. A smaller $r_{visited}$ enhances the risk of covering the same point multiple times, but makes the risk of missing spots much less. Since Sampled BA* & Inward Spiral is focused on minimising the cost per coverage, missing spots seems to be more expensive than covering the same points multiple times.

Experiment 2

The goal of Experiment 2 was to evaluate the robustness of the configurations from Experiment 1 for other starting points. In addition, it was used to see how the algorithms perform over time when they does not get stopped until they reach maximum coverage or a high time limit.

Computational time

Since the process of creating the path for BA* and Inward Spiral are similar computational time-wise, the difference between them depends mostly on the process of finding a new starting point. Since BFS, that is used in Inward Spiral, is a simpler algorithm than the backtracking list algorithm of BA*, Inward Spiral is expected to be faster than BA*. In addition to everything that the other algorithms have, Sampled BA* & Inward Spiral has sampling, cost calculations and the Traveling salesman problem algorithm, making it more computationally complex than the others.

Maximum coverage

The definition of reaching maximum coverage is different for the algorithms. BA* stops when no new starting point fulfills the requirements described in Equation 3.2. Finding such starting points is slightly harder than finding new starting points for Inward Spiral. It only requires a new starting point to be uncovered and traversable from current position using BFS. Sampled BA* & Inward Spiral does only reach it's maximum coverage when every traversable point in the environment is explored. Since this task takes a lot of computational time, the time limit was always reached before reaching maximum coverage for Sampled BA* & Inward Spiral .

There are two common main reasons why 100% coverage was not reached by the algorithms in the experiments. First reason is the existence of disconnected islands of coverable points which is a result of the flaws in the Terrain Assessment. The second is due to the geometry complexity of the borders of the environment. The path is made out of lines with a given step size going in 8 different directions. This simple geometry results in missed spots for some parts along the complex border. Both scenarios are shown in Figure 8.1.

As expected, BA* did not reach the same coverage in the Parking garage and Highway bridge environments, since the requirements of a valid new starting point in BA* is higher, making it reach the maximum coverage faster. However, BA* did reach higher coverage anyway for the city crossing. A possible reason is that the city crossing environment is square (less curved

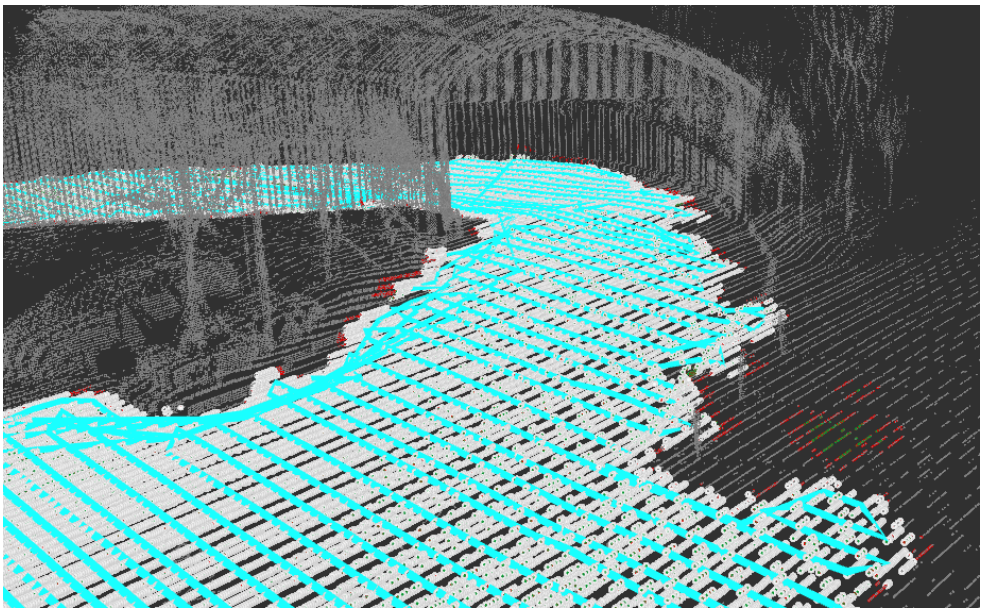


Figure 8.1: Part of the BA* path in the parking garage, showing an unreachable island of coverable points and uncovered points along the border. This shows the reason why 100% coverage were not reached. White points are covered by the path shown as light blue lines. Grey points are inaccessible and red points are missed coverable points.

borders) with a lot of open spaces, which could be beneficial for BA* since it works best in those kind of conditions. In theory, it could be beneficial for the Sampled BA* & Inward Spiral algorithm as well. However, the unexpected low value of $r_{visited}$ for Sampled BA* & Inward Spiral in the city crossing, see Table 7.3, could have made it harder to find segments with acceptable cost per coverage.

Length of path

Covering using spiral motions seems to have fairly the same effectiveness regarding length of the path as BA* for low coverages. When reaching higher coverage percentages the paths gets less structured and more movement between starting points is needed. Since this type of movement does not cover any new points it makes the algorithm ineffective.

Total Rotation

As expected, the curvy paths made the total rotation significantly higher for Inward Spiral compared to other algorithms.

A big strength of Sampled BA* & Inward Spiral is its low total rotation. One possible reason is that the path parts with many turns does not pass the cost per coverage requirement in the Sampled BA* & Inward Spiral algorithm. Since it, unlike BA*, can choose the best driving angle in different parts of the area, it will cover areas in the angle that makes the path goes along the longest side, resulting in less rotation. Another reason is that complex areas are effectively covered by spiral motions in Sampled BA* & Inward Spiral , unlike BA* that needs to cover those with back-and-forth paths. To cover them with back-and-forth paths a lot of turns is needed, resulting in more rotation.

Robustness

Since the Sampled BA* & Inward Spiral involves some randomness with sampling and a lot of parameters, it is expected to be less robust. The lack of robustness of Inward Spiral in the Highway bridge environment was not expected, but a reason could be unfavourable starting positions.

8.2 Method

In general the used method is reliable and should give the same results if the same point cloud, start points and parameters are used. Randomness in the hyper optimization of parameters and in the sampling of positions in Sampled BA* & Inward Spiral can give some variations, but is minimized by making a large amount of evaluation in the parameter optimization.

Three major components of the method were done to improve the validity of the results. Firstly, the experiments were made on multiple point clouds of different environments. Secondly, all parameters in the algorithm were automatically tuned with Bayesian optimization to avoid hand-tuning. Thirdly, the start points in Experiment 2 were randomly sampled to avoid picking biased start points that could result in biased results. There were a few parts of the method that is possible be criticise and a few changes that could have improved validity of the used method. These parts are discussed below.

Algorithm parameters

The tuning of parameters with hyper optimisation is a big strength of the used method which makes the results valid and minimises the human involvement. However, the parameter boundaries for the optimization were still hand tuned. Some boundaries were based on calculations, but most of the boundaries for the parameters in Sampled BA* & Inward Spiral were set after making some initial tests with big boundary intervals. These tests gave many unusable paths that did not reach high coverage within reasonable time. The used boundaries in the experiments were hand tuned to have a high rate of usable paths, but could have been biased.

Motion Planner

The Motion Planner, that handled traversability and shortest path plans for all algorithms seemed reasonable and no problems were found during tests. However, in theory, it has some flaws. Firstly, the parameters defining step size and untraversability threshold, were hand tuned on the parking garage point cloud, but could have been automatically tuned with hyper optimization to make sure they were optimised for any given point cloud. Secondly, the method assumed that the density of the points was evenly distributed over the point cloud. This assumption is not entirely true for the given environments and makes areas with no obstacles but less information undrivable because of low point density. The paths could have been optimised by driving between positions in 3D space, instead of between points, but this would not guarantee that the paths were collision free. On the other hand, the path plans would not be affected by noise in the point cloud as much as with the used method. Fortunately, the used point clouds did not have much noise and the density of the points were fairly even for traversable areas.

Robot

A significant approximation done in this method is the assumption that the robot is a spherical volume that cleans every point inside the sphere. In several ways this is a good approxi-

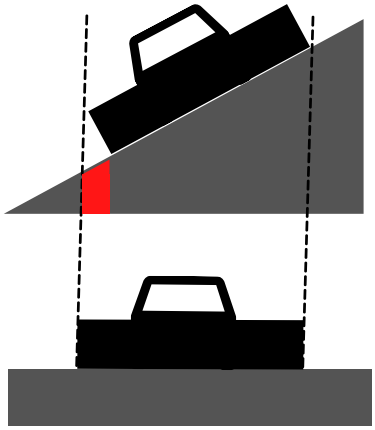


Figure 8.2: Neglecting the inclination of the ground results in skipped area. The range of the sweeper from a bird view perspective is smaller than expected.

mation, but could also differ a lot from a real robot. The real actuator is a square and has two sweep brushes in the front making the clean area in fact a trapezoid. When the robot moves, the cleaning actuator geometry can be approximated with a moving circle with a diameter as big as the distance between the sweep brushes, the robot breadth. By making it spherical does also take tilting of the robot into consideration. When the road is inclined the range of the robot is smaller than on flat floor, seen from a bird-view perspective, see Figure 8.2.

A flaw of the used method is that the planned paths does not take motion constraints of the robot into consideration. For future work there would be interesting to see how the planned paths performs in a real world environment with a real robot.

Coverage

Coverage was measured in number of points. This makes it easy to relate to the used terrain assessment method and does not require additional computations. It is based on the assumption that the point distribution over the point cloud is even, which is not always the case. It would have been better to use a method to measure covered area instead. Since Sampled BA* & Inward Spiral is based on these coverage measurements such change could definitely affect it's performance.

8.3 The work in a wider context

This thesis is a part of the revolution of automation in society. By having robots doing tasks such as cleaning streets and parking garages, it would replace today's drivers. This would take their jobs, but open up possibilities for new work such as maintaining and operating the robot, as well as handling laws regarding automotive driving in urban environments.

Self-driving robots in urban environments opens up many ethical questions, just like automotive driving, which is an active and well discussed topic today. The low speed of the cleaning robot and the fact that it is not transporting humans, makes it less dangerous than

self-driving cars. To avoid putting human health in danger during cleaning of environments such as parking garages, it could be done safely by doing the cleaning at night, when the parking garage is closed for visitors.

9 Conclusion

This chapter contains summarised answers to the research questions in Section 1.3 and ideas for future work.

9.1 Research questions

The aim written in Section 1.2 was achieved. A new offline footprint-based CPP approach called Sampled BA* & Inward Spiral was developed and showed promising performance in different environments compared to existing well known CPP algorithms. The implementation of BA*, Inward Spiral and the new algorithm is applicable directly on 3D point clouds with complex terrain geometry and does not require a 2D-grid representation of the environment. The used method minimizes total rotation and length of the path by automatically tuning algorithm parameters for a given point cloud. This makes the used method and the new algorithm a good choice for Dyno Robotics to use for their sweeping robot, but needs to be tested onboard a real robot.

BA* vs Inward Spiral - coverage, length of path and total rotation over time

Results of the experiments showed that Inward Spiral reached higher coverage than BA* for two out of three environments (1% and 3% higher). However, BA* did reach higher coverage in an environment that were square and open. For both algorithms the coverage is fast in the beginning and slows down when reaching higher coverage. Inward Spiral is faster. The length of path over time is more or less the same in the beginning, but changes significant when reaching higher coverage percentages. Inward Spiral gets less effective than BA*, which makes the final path longer. The total rotation is significantly lower for BA* than Inward Spiral.

Can length of path and total rotation be improved with the new approach for one starting point?

After optimizing the algorithm performances with automatic parameter tuning the new approach, Sampled BA* & Inward Spiral , generated a path with less rotation than BA* (20-40%

reduction) and Inward Spiral (50-60% reduction) for all environments. All algorithms generated paths with similar lengths and the shortest path were different for different environments. However, Sampled BA* & Inward Spiral algorithm generated the path with lowest total cost (between 5-30%), defined as length of path + total rotation, for all environments.

How good does the algorithms perform for other starting points?

In general, the optimal configuration performed similarly on other starting points. The Sampled BA* & Inward Spiral showed least robustness and had an increase in total cost (length of path + total rotation) of 10% and 13% for two environments. BA* and Inward Spiral showed good robustness, except in one environment where the Inward Spiral performed significantly worse (21% increase in cost).

9.2 Future work

The next step is to validate these results in real world environments with a real sweeping robot. Another direction is to compare the new approach with other CPP algorithms and to see how it performs in other environments.

Terrain Assessment is an adjacent field that could be investigated further and put together with the method used in this work. As a suggestion, use a method that automates the method in Chapter 4 and removes all manual tuning. It would be interesting to compare the used method with other approaches for the motion planner and environment representation to find further improvements.

Bibliography

[1] David Applegate, William Cook, and Andr e Rohe. “Chained Lin-Kernighan for large traveling salesman problems”. In: (2000).

[2] Marco Attene, Bianca Falcidieno, and Michela Spagnuolo. “Hierarchical mesh segmentation based on fitting primitives”. In: *The Visual Computer* 22 (Mar. 2006), pp. 181–193. DOI: 10.1007/s00371-006-0375-x.

[3] James Bergstra, Daniel Yamins, and David Cox. “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures”. In: *International conference on machine learning*. PMLR. 2013, pp. 115–123.

[4] S. Bochkarev and S. L. Smith. “On minimizing turns in robot coverage path planning”. In: *2016 IEEE International Conference on Automation Science and Engineering (CASE)*. 2016, pp. 1237–1242. DOI: 10.1109/COASE.2016.7743548.

[5] R. Bormann, J. Hampp, and M. Hägele. “New brooms sweep clean - an autonomous robotic cleaning assistant for professional office cleaning”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4470–4477. DOI: 10.1109/ICRA.2015.7139818.

[6] R. Bormann, F. Jordan, J. Hampp, and M. Hägele. “Indoor Coverage Path Planning: Survey, Implementation, Analysis”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 1718–1725. DOI: 10.1109/ICRA.2018.8460566.

[7] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun. “Principles of Robot Motion : Theory, Algorithms, and Implementations”. In: Intelligent Robotics and Autonomous Agents Ser. MIT Press, 2005, pp. 161–174. ISBN: 9780262255912.

[8] T. Danner and L. E. Kavraki. “Randomized planning for short inspection paths”. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 2. 2000, 971–976 vol.2. DOI: 10.1109/ROBOT.2000.844726.

[9] Sedat Dogru and Lino Marques. “A*-Based Solution to the Coverage Path Planning Problem”. In: Jan. 2018, pp. 240–248. ISBN: 978-3-319-70832-4. DOI: 10.1007/978-3-319-70833-1_20.

[10] Nils Einecke, Jörg Deigmöller, Keiji Muro, and Mathias Franzius. "Boundary wire mapping on autonomous lawn mowers". In: *Field and Service Robotics*. Springer. 2018, pp. 351–365.

[11] B. Englot and F. S. Hover. "Sampling-based sweep planning to exploit local planarity in the inspection of complex 3D structures". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 4456–4463. DOI: 10.1109/IROS.2012.6386126.

[12] Brendan Englot and Franz S. Hover. "Sampling-Based Coverage Path Planning for Inspection of Complex Structures". In: *Proceedings of the Twenty-Second International Conference on International Conference on Automated Planning and Scheduling*. ICAPS'12. Atibaia, São Paulo, Brazil: AAAI Press, 2012, pp. 29–37.

[13] Y. Gabriely and E. Rimon. "Spiral-STC: an on-line coverage algorithm of grid environments by a mobile robot". In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*. Vol. 1. 2002, 954–960 vol.1. DOI: 10.1109/ROBOT.2002.1013479.

[14] Enric Galceran and Marc Carreras. "A survey on coverage path planning for robotics". In: *Robotics and Autonomous Systems* 61.12 (2013), pp. 1258–1276. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2013.09.004. URL: https://www.sciencedirect.com/science/article/pii/S092188901300167X.

[15] H. González-Banos. "A randomized art-gallery algorithm for sensor placement". In: *Proc. 17th ACM Symp. Comp. Geom.* (Jan. 2001), pp. 232–240. DOI: 10.1145/378583.378674.

[16] Felipe Goulart. "Python TSP Solver". In: (2021). URL: https://github.com/fillipe-gsm/python-tsp.

[17] I.A. Hameed, A. la Cour-Harbo, and O.L. Osen. "Side-to-side 3D coverage path planning approach for agricultural robots to minimize skip/overlap areas between swaths". In: *Robotics and Autonomous Systems* 76 (2016), pp. 36–45. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2015.11.009. URL: https://www.sciencedirect.com/science/article/pii/S0921889015002973.

[18] Ibrahim Hameed. "Intelligent Coverage Path Planning for Agricultural Robots and Autonomous Machines on Three-Dimensional Terrain". In: *Journal of Intelligent Robotic Systems* 74 (June 2013). DOI: 10.1007/s10846-013-9834-6.

[19] Keld Helsgaun. "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic". In: *European Journal of Operational Research* 126 (2000), pp. 106–130.

[20] Armin Hornung, Kai Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees". In: *Autonomous Robots* 34 (Apr. 2013). DOI: 10.1007/s10514-012-9321-0.

[21] Jinyong Jeong, Younggun Cho, Young-Sik Shin, Hyunchul Roh, and Ayoung Kim. "Complex urban dataset with multi-level sensors from highly diverse urban environments". In: *The International Journal of Robotics Research* (2019), p. 0278364919843996.

[22] Jian Jin and Lie Tang. "Coverage path planning on three-dimensional terrain for arable farming". In: *Journal of Field Robotics* 28.3 (2011), pp. 424–440. DOI: https://doi.org/10.1002/rob.20388. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.20388. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.20388.

[23] David S. Johnson. "Approximation algorithms for combinatorial problems". In: *Journal of Computer and System Sciences* 9.3 (1974), pp. 256–278. ISSN: 0022-0000. DOI: https://doi.org/10.1016/S0022-0000(74)80044-9. URL: https://www.sciencedirect.com/science/article/pii/S0022000074800449.

[24] Alex Korolov. "Experts predict how AI will look in 2030". In: *Hypergrid Business* (2020). URL: https://www.hypergridbusiness.com/2020/12/experts-predict-how-ai-will-look-in-2030/.

[25] Philipp Krüsi, Paul Furgale, Michael Bosse, and Roland Siegwart. "Driving on point clouds: Motion planning, trajectory optimization, and terrain assessment in generic nonplanar environments". In: *Journal of Field Robotics* 34.5 (2017), pp. 940–984.

[26] Florent Lamiraux and Jean-Paul Laumond. "On the expected complexity of random path planning". In: vol. 4. May 1996, 3014–3019 vol.4. ISBN: 0-7803-2988-0. DOI: 10.1109/ROBOT.1996.509170.

[27] Zhang Ruishu, Zhang Chang, and Zheng Weigang. "The status and development of industrial robots". In: *IOP Conference Series: Materials Science and Engineering* 423 (Nov. 2018), p. 012051. DOI: 10.1088/1757-899x/423/1/012051. URL: https://doi.org/10.1088/1757-899x/423/1/012051.

[28] Vytenis Sakenas, Olegas Kosuchinas, Max Pfingsthorn, and Andreas Birk. "Extraction of Semantic Floor Plans from 3D Point Cloud Maps". In: *2007 IEEE International Workshop on Safety, Security and Rescue Robotics*. 2007, pp. 1–6. DOI: 10.1109/SSRR.2007.4381276.

[29] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. "Taking the human out of the loop: A review of Bayesian optimization". In: *Proceedings of the IEEE* 104.1 (2015), pp. 148–175.

[30] Sebastian Thrun. "Learning metric-topological maps for indoor mobile robot navigation". In: *Artificial Intelligence* 99.1 (1998), pp. 21–71. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(97)00078-7. URL: https://www.sciencedirect.com/science/article/pii/S0004370297000787.

[31] Hoang Viet, Viet-Hung Dang, Md Laskar, and TaeChoong Chung. "BA: An online complete coverage algorithm for cleaning robots". In: *Applied Intelligence* 39 (Sept. 2013). DOI: 10.1007/s10489-012-0406-4.

[32] S. X. Yang and C. Luo. "A neural network approach to complete coverage path planning". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34.1 (2004), pp. 718–724. DOI: 10.1109/TSMCB.2003.811769.

[33] A. Zelinsky, R. Jarvis, J. Byrne, and S. Yuta. "Planning Paths of Complete Coverage of an Unstructured Environment by a Mobile Robot". In: 1993.

[34] Hongmei Zhang, Wei Hong, and Mingjie Chen. "A Path Planning Strategy for Intelligent Sweeping Robots". In: Aug. 2019, pp. 11–15. DOI: 10.1109/ICMA.2019.8816519.