
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# PATH PLANNING ALGORITHMS VISUALISATION

**VIZUALIZACE ALGORITMŮ PRO PLÁNOVÁNÍ CESTY**

**BACHELOR'S THESIS**
**BAKALÁŘSKÁ PRÁCE**


**AUTHOR** **MÁRTON BRÉDA**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. JAROSLAV ROZMAN, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2022**

**Brno University of Technology**
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

# Bachelor's Thesis Specification


Student: **Bréda Márton**
Programme: Information Technology
Title: **Path Planning Algorithms Visualisation**
Category: Artificial Intelligence

Assignment:

1. Study algorithms for path planning algorithms that are used in robotics (road maps, cell decomposition, probabilistic algorithms, etc.). Study Diploma work of Jakub Rusňák.
2. Design your own application or adjust existing one that allows visualisation of path planning algorithms. Consider also with later adding of algorithms.
3. Implement the designed application, including selected algorithms and properly describe the used algorithms.
4. Test the application and create manual for easily adding of other algorithms.

Recommended literature:

- Howie Choset et al., Principles of Robot Motion, 2005, ISN 0-262-03327-5.
- Rusnák Jakub, Vizualizace algoritmů pro plánování cesty, bakalářská práce, FIT VUT v Brně, 2017.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rozman Jaroslav, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

# Abstract

The goal of this paper is to show some of the most important algorithms used in path planning. It also describes the application, that was created to allow people to experiment with these algorithms. For this purpose it uses the library that was introduced by Jakub Rusnák in 2017, which means this is a continuation and possibly extension of his work.



# Abstrakt

Cílem tohoto práce je ukázat některé z nejdůležitějších algoritmů používaných při plánování cest. To také popisuje aplikaci, která byla vytvořena, aby umožnila lidem experimentovat s těmito algoritmy. K tomuto účelu využívá knihovnu, kterou v roce 2017 představil Jakub Rusnák, tzn jde o pokračování a možná i rozšíření jeho práce.



# Keywords

visualization, java, path planning algorithms, roadmap, cell decomposition, probabilistic roadmap, algorithm presentation



# Klíčová slova

vizualizace, java, algoritmy plánování cesty, road mapa, buněčné dekompozice, pravděpodobnostní algoritmy, java, demonstrační aplikace


# Reference


BRÉDA, Márton. *Path Planning Algorithms Visualisation.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jaroslav Rozman, Ph.D.

# Path Planning Algorithms Visualisation

## Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pána Ing. Jaroslava Rozmana, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Márton Bréda
May 6, 2022

# Contents

# Chapter 1

# Introduction

Pathfinding is an important part of the daily operation of many objects. As a mathematical problem, it has been solved in detail from many angles. More important is therefore at this point to pass on this knowledge in a comprehensible form, so other students can continue to build on it. This work tries to describe and demonstrate with examples some of the most important algorithms in this industry.

The goal of this thesis was to implement some of the most important algorithms, that solve path planning problems. To achieve this, an application using the Java language was implemented for the purposes of experimentation with these algorithms. The application requires a user interface, that is intuitive with the main focus on the ability to enable the user to quickly change between the implemented algorithms. The application also needs to enable the user to execute the algorithms step-by-step, enabling a more detailed understanding about the workings of the implemented algorithms.

The structure of the thesis is as follows. Chapter 2 contains a short introduction to the theory of path planning, it introduces the basic concepts to the reader, so the algorithms described in the other chapters can be understood easier. In chapter 3 describes algorithms that generate deterministic roadmaps. Chapter 4 is dedicated to cell decompositions. Chapter 5 explains algorithms that generate probabilistic roadmaps. Chapter 6 is dedicated to the description of the implemented application. Section 6.1 describes the *vizlib* library implemented by Jakub Rusnák, this library is used to help with the visualization of the selected algorithms. Section 6.4 describes how the algorithms were tested, it also contains statistics about the data gathered during testing.

A textbook of planning algorithms was used as the primary source for this work [4], the book [11] was used as the secondary source.

# Chapter 2

# Introduction to path planning

This chapter is dedicated to explaining some of the basic terminology and concepts used in path planning. These terms are used in the rest of the thesis, so it is natural, that the beginning chapter is dedicated to their explanation.

## 2.1 Path planning problem, workspace and configuration space

In [11], Latombe describes the path planning problem as follows:

- $A \subset W$: The robot, it is a single moving rigid object in world $W$ represented in the Euclidean space as $\mathbb{R}^2$ or $\mathbb{R}^3$.
- $O \subset W$: The obstacles are stationary rigid objects in $W$.
- The geometry, the position, and the orientation of $A$ and $O$are known a priori.
- The localization of the $O$ in $W$ is accurately known.

Given a start and goal positions of $A \subset W$, plan a path $P \subset W$ denoting the set of positions, so that $A(p) \cap O = \emptyset$ for any position $p \in P$a long the path from start to goal, and terminate and report $P$ if a path has been found, or $\emptyset$ if no such path exists.

In this thesis, robots are assumed to operate in a planar ($\mathbb{R}^2$) ambient space, this is referred to as the *workspace* denoted as $\mathcal{W}$. In order to construct the workspace of a robot, four different concepts need to be well described:

- The robot's geometry
- The workspace in which the robot moves or acts
- The degrees of freedom of the robot's motion
- The initial and the target configuration in the environment

Path planning problems can be directly solved in the workspace, but motion planning does not usually occur in the workspace. Instead, it occurs in the *configuration space*. The concept of a configuration space was first presented by Lozano-Pérez [12]. It contains all the possible configurations of the robot in a workspace. It is usually denoted by $C$, but in this thesis it is referred to as $Q$, to keep it in line with the theory presented in [4].

In realistic environments the workspace also contains obstacles. An obstacle in the configuration space corresponds to configurations of the robot that intersect an obstacle in the workspace. Due to these obstacles some configurations are *forbidden*. A configuration $c$ is forbidden if the robot intersects an obstacle while positioned at configuration $c$. Now the concept of the *free configuration space* can be defined as the set of all configurations $C$ minus the set of all forbidden configurations. The free configuration space is usually denoted as $C_{free}$, but in this thesis it is referred to as $Q_{free}$, to keep it in line with the theory presented in [4].

## 2.2 Properties of path planners

This section summarizes some of the most important concepts of path planners. In [4] the book the path planners are characterized according to three criteria:

- The task the planner addresses
- The properties of the robot solving the task
- The properties of the algorithm the planner uses

### Task

The book [4] considers four tasks: *navigation*, *coverage*, *localization* and *mapping*. Navigation is finding a collision-free path from one configuration to another. Coverage is passing a sensor over all points in the workspace. Localization is using sensor data to determine the configuration of the robot. Mapping is exploring and sensing an unknown environment to construct a representation that is useful for navigation, coverage, or localization. This thesis deals with algorithms used for navigation and coverage.

### Properties of the Robot

Any object that needs to be moved safely can be the subject of a path planning. The planner is significantly affected by the complexity of the object, the dimension of the configuration space increases, as the complexity of the robot increases. For this reason, the paper considers a simple, autonomous robot so that the reader can focus fully on the planning algorithms themselves. The robot is one-dimensional, its configuration is defined by it's position $(x, y)$.

#### Properties of the path planning algorithms

Path planning algorithms can be divided into multiple groups according to certain criteria.

Depending on the nature of the environment they are designed for, there are planners designed for *static* environments, and *dynamic* environments. A static environment is unvarying, the start and goal configurations along with the obstacles are fixed. In dynamic environments the location of the start, goal and obstacles may vary during the search process.

Depending on whether the robot has a priori knowledge about the environment, a planner can be *global*, if the robot has knowledge about the environment (in form of a map for example). A planner is local, if it does not have a priori information knowledge about its environment. In this case the robot needs a way to sense the locations of the obstacles, and construct a map of the environment during the search process.

Depending on the completeness of the path planning algorithm, it can be *complete*, meaning that if a solution exists it can find it in finite time. As the complexity of the configuration space increases, the computing time for these kinds of algorithms may become too long to be of any practical use. A planner can have weaker forms of completeness, such as *resolution completeness*. It means, that if a solution exists at a given resolution, the planner will find it. Another weaker form of completeness is *probabilistic completeness*. It means that the probability of finding a solution converges to 1 as time goes to infinity.

# Chapter 3

# Roadmaps

Planners usually plan a path from a particular start configuration to a particular goal configuration. In the case, where multiple paths have to be planned in the same configuration space it would make sense to construct a representation of the configuration space, and save it in a data structure, so it can be accessed later to make the construction of future paths quicker. The data structure is referred to as a *map*, the procedure of generating the map is called *mapping*.

This chapter focuses on a class of maps called roadmaps [11]. A roadmap is embedded in the free space and hence the nodes and edges of a roadmap also carry physical meaning. Using a roadmap, the planner can construct a path between any two points in a connected component of the robot's free space by first finding a collision-free path onto the roadmap, traversing the roadmap to the vicinity of the goal, and then constructing a collision-free path from a point on the roadmap to the goal. This can be done, because all roadmaps have the following three properties:

- Accessibility : there exists a path from the start to at least one of the graph nodes of the roadmap, let this node be $q'_{start}$
- Departability : there exists a path from the goal to at least one of the graph nodes of the roadmap, let this node be $q'_{goal}$
- Connectivity : there exists a path between $q'_{start}$ and $q'_{goal}$ for any $q'_{start}$ and $q'_{goal}$

The following sections describe three different types of roadmaps: *visibility maps*, like the Visibility Graph described in section 3.1, *deformation retracts*, like the Generalized Voronoi Diagram described in section 3.2 and *silhouettes*, like Canny's Roadmap Algorithm described in section 3.3.

## 3.1 Visibility Graph

The two defining characteristics of visibility maps are [4]:

- The nodes share an edge, if they are within line of sight of each other
- Every point of the configuration space is within line of sight of at least one node of the visibility graph

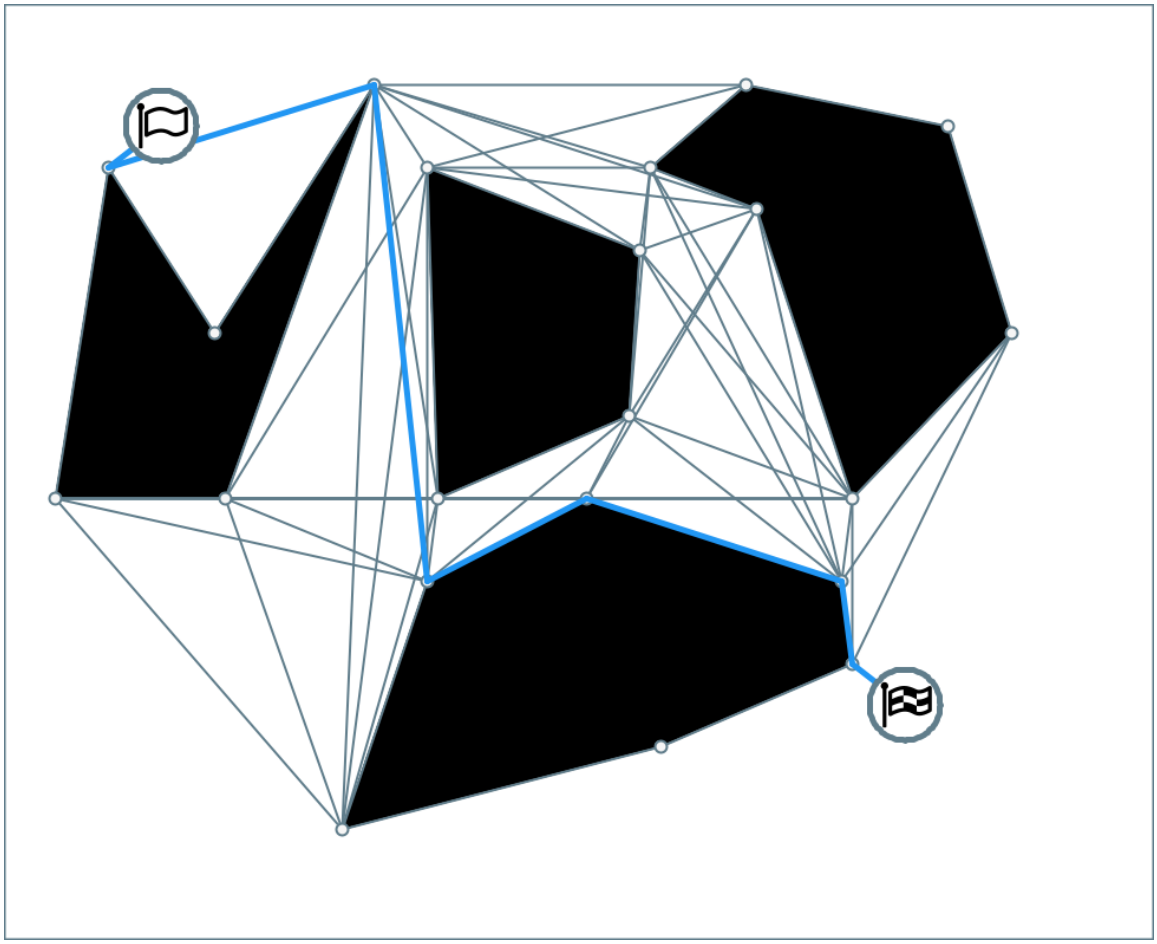


Figure 3.1: The thin lines indicate the edges of the graph. The circles indicate the vertexes, the blue line indicates the path between start and goal.

The simplest visibility map is called the visibility graph. [12] [11]

The standard visibility graph is defined in a two-dimensional polygonal configuration space. The nodes correspond to the vertexes of the polygons, and include the start and goal locations, the edges are straight lines connecting two line-of-sight nodes. Figure 3.1 shows an example of a visibility graph.

The visibility graph has many needless edges, and needless vertexes. The vertexes, that are reflex are unnecessary to include in the graph [4]. A vertex $V$ of a polygon is a reflex vertex if its internal angle is strictly greater than $\pi$ . Otherwise the vertex is called convex. To determine whether a vertex is reflex, one can for example use the following method: take the two lines of the polygon, that the vertex is part of. Determine the center of the two lines, and connect them with a line. Determine the center of this line as well. If this center lies on the outside of the obstacle, the vertex is reflex.

It is also unnecessary to add the lines, that are not *separating lines* or *supporting lines* to the graph [4]. A supporting line is tangent to two obstacles such that both obstacles lie on the same side of the line. A separating line is tangent to two obstacles such that the obstacles lie on opposite sides of the line. To determine whether a polygon lies on one side of a line $l$ the following method can be used. For each of the polygon's edges, calculate the cross product of $l$ and the edge. If all of the cross products are not negative, or all of

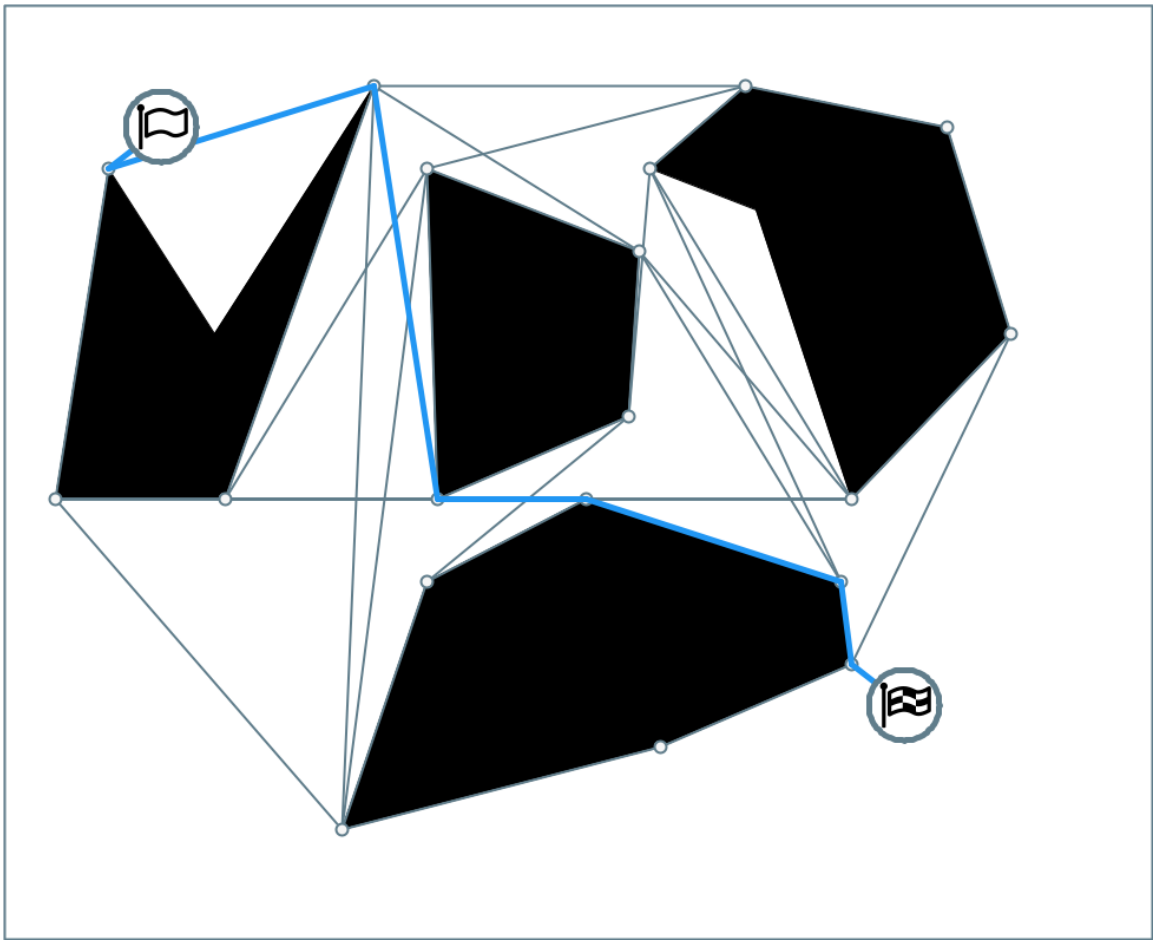


Figure 3.2: The thin lines indicate the edges of the graph. The circles indicate the vertexes, the blue line indicates the path between start and goal.

them are not positive, then all of the edges of the polygon lie or one side of the line, and therefore the whole polygon lies on one side of the line.

The *reduced visibility graph* is constructed from supporting and separating lines. Figure 3.2 shows an example of a reduced visibility graph.

## 3.2 Generalized Voronoi Diagram

First, the thesis gives a definition of the *Voronoi diagram*. Let $S$ denote a set of points, these points are referred to as *sites* in the plane [2]. A *Voronoi region* is the set of points closest to a site [2]. The Voronoi diagram is the set of points equidistant to two sites. The diagram itself sections of the free space into regions, that are closest to a particular site. An example is provided in figure 3.3.

The generalized Voronoi diagram (GVD) is a structure, that divides the configuration space into generalized Voronoi cells (GVCs) around objects. Similar to the ordinary Voronoi diagram, each GVC contains exactly one object, or site, and every point in the GVC is closer to its contained object than to any other object. The generalized Voronoi diagram is the

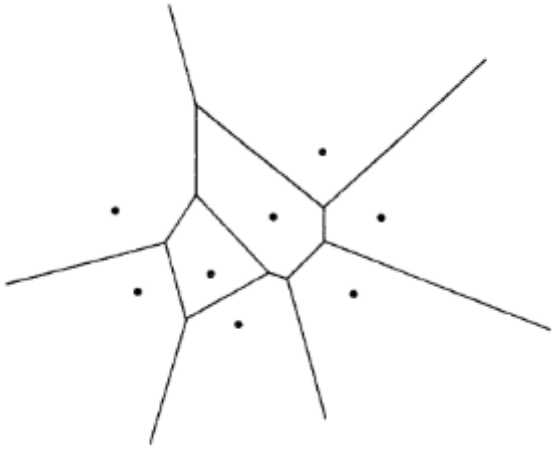


Figure 3.3: Voronoi diagram for eight sites in the plane [2]

boundary of the cell complex, and thus every point on the GVD is equidistant from two or more closest objects [6].

### 3.2.1 Construction of the GVD

In this section several methods for the construction of the GVD is described.

The first method is a sensor based approach. It incrementally constructs the GVD using the range sensors of the robot. Using line-of-sight data, the robot accesses the GVD, and begins tracing an edge until it reaches a *meet point* (a point, that is equidistant to three or more obstacles), or a *boundary point* (a point, where the distance to the closest point is zero). When the robot encounters a new meet point, it marks off the direction from which it came as explored, and then identifies all new GVD edges that emanate from it. From here, the robot explores a new GVD edge until it detects either another meet point or a boundary point. If it detects another new meet point, the above branching process recursively repeats. If the robot reaches an old meet point, the robot travels to a meet point with an unexplored edge associated with it. When the robot reaches a boundary node, it simply turns around and returns to a meet point with unexplored GVD edges. When all meet points have no unexplored edges associated with them, exploration is complete.

The second method makes an assumption, that all obstacles in the configuration space are polygonal. In such space, obstacles have two features, vertexes and edges. The set of points equidistant to two vertices is a line; the set of points equidistant to two edges is a line; and the set of points equidistant to a vertex and an edge is a parabola. The planner can divide the free space into regions, and build the GVD that way.

The third method divides the configuration space into cells, the grid cells that contain obstacles are assigned the number one, the rest are assigned zero. In the first step, all zero-valued cells neighboring one-valued cells are labeled with two. Next, all zero-valued pixels adjacent to two's are labeled with a three, this repeats until all cells are numbered. This is called the *Bushfire algorithm* [4]. The method can be viewed as a wave front passing over the cells of the grid. The wave fronts meet at points where the distance to two different obstacles is the same. These are points on the GVD.

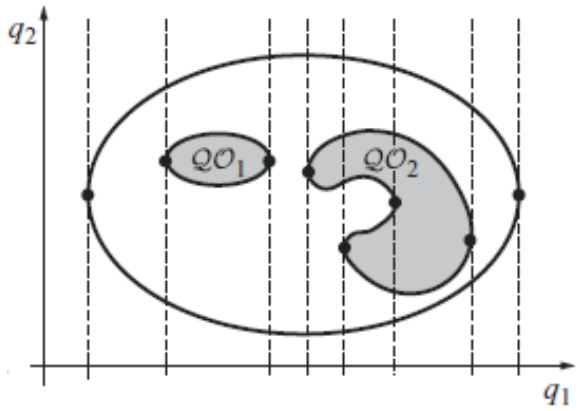



Figure 3.4: Example of the result of sweeping a slice through a two-dimensional configuration space containing two obstacles. The dashed lines represent critical slices, the black dots represent critical points, and the silhouette is highlighted in black

## 3.3 Canny's Roadmap Algorithm

The algorithm is described in great detail in [3]. This thesis only describes it for a two-dimensional configuration space.

The method described in [3] assumes an arbitrary sweep direction, for the sake of the explanation, the slice chosen is parallel to the $y$ axis of the configuration space, and the slice is swept through the configuration space in the $q_1$ direction (the direction towards in which the $x$-coordinate increases in value). As the slice is swept through the configuration space, *extremal points* along the slice are determined for each slice. In this case these extremal points are the points where the slice intersects the boundary of an obstacle. The extremal points of all the slices are the *silhouette curves*. These curves are usually not connected, but one can look at the slices, where the number of silhouette curves changes. These slices are called *critical slices*, and the $x$-coordinates that parameterize them are called are called *critical values*. The points on the silhouette curves where the silhouette curves are tangent to the critical slices are named *critical points*. Normally on critical slices the silhouette algorithm is recursively invoked where the new swept slice now has one less dimension than the critical slice, and this slice is swept in a direction perpendicular to the $q_1$ direction. In this case it would cause the dimension of the slice to drop from two to one. In this case the one-dimensional slice is the silhouette. The result is showcased in figure 3.4.

To connect the start and the goal to the graph, the slices they are part of are treated as critical one-dimensional slices of the initial sweep. The algorithm forms a silhouette network on these slices. The result is showcased in figure 3.5

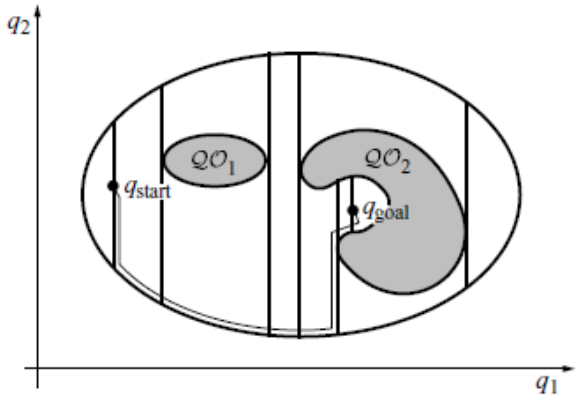
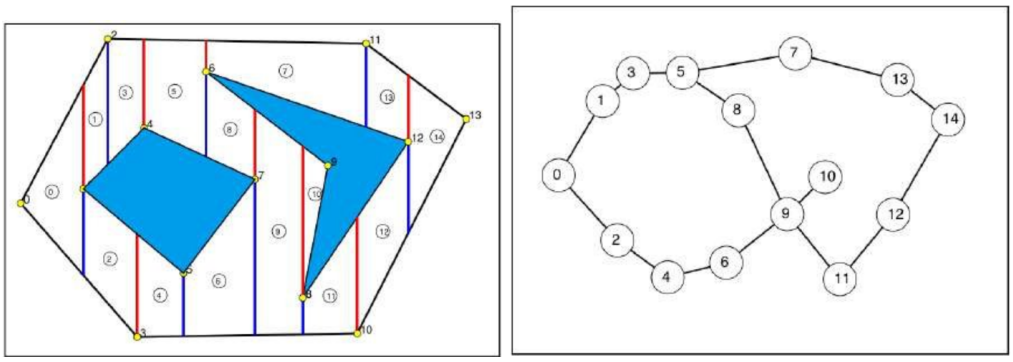



Figure 3.5: Complete silhouette curves traced out with solid lines. A path from start to goal is represented as a thin curve.

# Chapter 4

# Cell Decompositions

In this chapter a different representation of the free space is considered, called an *exact cell decomposition*. These representations divide the configuration space into simple regions called *cells*. The decomposition is exact if, by unifying all the cells, we get the whole configuration space. Shared edges between cells are not globally defined, but depend on local conditions, such as changing the nearest or adjacent obstacle. Two cells are *adjacent*, if they share such an edge. A graph can be constructed, which represents these relationships, a node corresponds to a cell, and edges connect the nodes of adjacent cells. This graph is called an *adjacency graph*. With the decomposition of the configuration space found, it is possible to plan a route between the initial and the target configuration in two steps. First, the cells corresponding to the start and destination are determined. Second, a path between these cells in the adjacency graph is then searched using algorithms such as A* [8] or Dijkstra [5]. An example of a decomposition and adjacency graph can be found on figure 4.1.




(a) The decomposition [13] (b) And the adjacency graph [13]

Figure 4.1: Trapezoidal decomposition example from [13]

However, cell decompositions can also be used for a different purpose. These algorithms distinguish themselves by allowing for the creation of a path, that passes through all points of the configuration space. A path planner, that can construct such a path is called a *Coverage path planner*, and the path created this way is called a *coverage path*. A coverage path can be computed in the following way: first the planner determines an exhaustive walk through the adjacency graph, since each cell has a simple structure, they can be covered by

simple back-and-forth or spiral motions. After each cell is covered in such a way, coverage is achieved.

This chapter describes the most popular cell decomposition, which is the *trapezoidal decomposition* [4].

## 4.1 Trapezoidal Decomposition

Due to its simplicity, the most used decomposition among the cell decompositions, is the trapezoidal decomposition. One could say that dividing the configuration space into triangles is more natural. However, this is not the case and even many algorithms for planar triangulation use trapezoids as their initial decomposition.

This decomposition is limited to a polygonal environment, i.e. an environment in which each object, including the outer boundary of the available space, is represented by a polygon. Algorithms capable of dividing any configuration space are called *Morse decompositions* [1].

### Definition of the Decomposition

For the sake of explanation, assume that each vertex $v_i$ on all of the polygons has a unique $x$ coordinate, i.e., for all $i \neq j$ , $v_{i_x} \neq v_{j_x}$.

To create a decomposition, two half-lines are created at each vertex of the obstacles. One upwards called the upper vertical extension and the other downwards called the lower vertical extension, i.e. to both sides of the $y$ axis. When the half-line hits the edge of an obstacle, a new vertex is created at the intersection point, and a new edge is created between the intersection point and the vertex. For many vertexes this means that they generate a line in only one direction, or no lines at all.

The next step is to determine which cells contain the start and goal, once that is done, the planner can search the adjacency graph for a path between cells. The result of the graph search is a sequence of nodes, not a path. These nodes have to be connected, and an explicit path determined. The trapezoid is a convex set, meaning any two points on the inside of the trapezoid can be connected by a straight line segment, that does not intersect any obstacles. The planner can use this fact, by connecting the midpoints of the vertical extensions to the centroids of each trapezoid. The start and goal can be connected to the graph by drawing a straight line to the vertical extensions midpoints, or the centroid of the trapezoids containing them.

### Construction of the Decomposition

The next issue is the construction of the decomposition. The input for the algorithm is a configuration space containing polygons, represented by a list of vertexes. The first step is to sort the vertexes by their $x$-coordinate in ascending order. This takes $O(n \log n)$ time, and $O(n)$ space, where $n$ is the total number of vertexes. The next step is to determine the vertical extensions. The extensions can be determined by sweeping a sweep line (similar to the slice in Canny's roadmap algorithm section 3.3 5) through the free space stopping at the vertexes, which are sometimes termed *events*. While passing the sweep line, the planner can maintain a list $L$ that contains the "current" edges which the slice intersects.

With the list $L$, determining the vertical extensions at each event requires $O(n)$ time with a simple search, but if the list is stored in an "efficient" data structure like a balanced tree, then the search requires $O(\log n)$ time. It is easy to determine the $y$-coordinates of

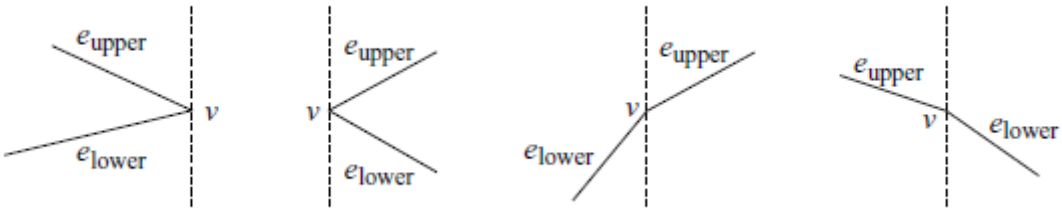


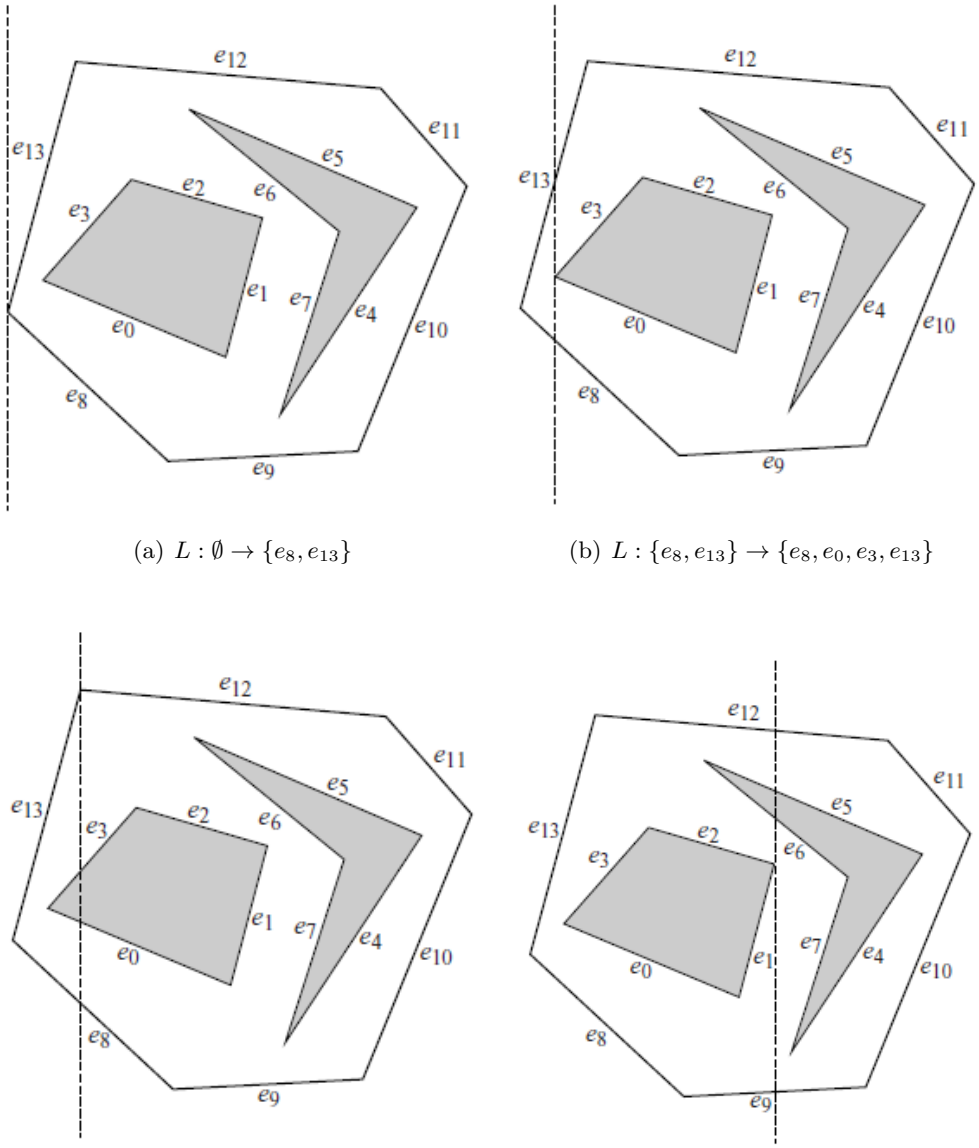

Figure 4.2: The types of events from [4]

the intersection of the line that passes through each vertex $v_i$ and each edge $e_i$ . The trick is to find the appropriate edge or edges for the vertical extensions, i.e., the two edges that v lies between. Let these two edges be called $e_{LOWER}$ and $e_{UPPER}$.

So as long as the "current" list requires $O(\log n)$ insertions and deletions, as balanced trees do, then keeping track of all the edges that intersect the sweep line, i.e., maintaining $L$, requires $O(n \log n)$ time. Let $e_{lower}$ and $e_{upper}$ be the two edges that contain v (these are not $e_{LOWER}$ and $e_{UPPER}$). The "other" vertex of $e_{lower}$ has a $y$-coordinate lower than the "other" vertex of $e_{upper}$. Now, there are four types of events that can occur (figure 4.2 illustrates these events) and the type of event determines the appropriate action to take on the list. These events and actions are:

- $e_{lower}$ and $e_{upper}$ are both to the left of the sweep line
  - delete $e_{lower}$ and $e_{upper}$ from the list
- $e_{lower}$ and $e_{upper}$ are both to the right of the sweep line
  - insert $e_{lower}$ and $e_{upper}$ into the list
- $e_{lower}$ is to the left and $e_{upper}$ is to the right of the sweep line
  - delete $e_{lower}$ from the list and insert $e_{upper}$ into the list
- $e_{lower}$ is to the right and $e_{upper}$ is to the left of the sweep line
  - delete $e_{upper}$ from the list and insert $e_{lower}$ into the list

The figure 4.3 contains examples of the sweep line being swept through a polygonal free space with the corresponding list updates at each event.




(a) $L : \emptyset \rightarrow \{e_8, e_{13}\}$

(b) $L : \{e_8, e_{13}\} \rightarrow \{e_8, e_0, e_3, e_{13}\}$

(c) $L : \{e_8, e_0, e_3, e_{13}\} \rightarrow \{e_8, e_0, e_3, e_{12}\}$

(d) $L : \{e_9, e_1, e_2, e_6, e_5, e_{12}\} \rightarrow \{e_9, e_6, e_5, e_{12}\}$

Figure 4.3: Examples of the sweep line being swept through polygonal free space [4]

# Chapter 5

# Sampling-based algorithms

This chapter introduces the basic concept of Sampling-Based algorithms. Section 5.2 describes the Probabilistic Roadmap Algorithm (from this point on PRM is used as an abbreviation). Section 5.3 describes Single-query Sampling-Based Planners, Expansive-Spaces Trees (from this point on EST is used as an abbreviation) is described in 5.3.1, and Rapidly Exploring Random Trees (from this point on RRT is used as an abbreviation) is described in 5.3.2.

The algorithms described in chapter 3 build roadmaps in the free configuration space. Every algorithm relied on an explicit representation of the geometry of the configuration space. These algorithms become impractical in higher dimensions, as their computation time increases exponentially [4]. Sampling-based algorithms employ a variety of strategies for generating samples (which are collision-free configurations in the configuration space), and for connecting the samples with paths. These samples and connections result in a map, which can be used to solve path-planning problems.

Sampling-based algorithms can be divided into these categories:

- Single-query – for all queries, the algorithm constructs a new graph, which connects the start and goal points. Examples include RRT, explained in chapter 5.3.2, or EST, explained in chapter 5.3.1.
- Multi-query – these algorithms construct a graph in the configuration space once. To search for a path between two points, these algorithms add the two points to the graph, and search for a path between them. This operation is repeated for each subsequent query. Examples include PRM, which is explained in chapter 5.2
- Combined – these algorithms use both approaches, and can be used for multiple searches, and they can use a one-query algorithm internally. Examples include Sampling Based Roadmap of Trees

## 5.1 Sampling-Based planner characteristics

A characteristic of planners described in this chapter is, that they do not construct the boundaries of the configuration space, or represent cells in the configuration space. Instead they rely on procedures, which can determine, whether a configuration of a robot is collision free.

Another characteristic of these planners, is that they have some form of completeness. Completeness requires, that the planner always answers a path-planning query correctly, in

asymptotically bounded time. Complete planners cannot be implemented for high dimension configuration spaces, due to the complexities of the computation required. A weaker form of completeness can be achieved however: if a path exists, the planner will find it after enough time has passed. The PRM has this characteristic [4].

## 5.2 Basic PRM

The PRM was first described in [10]. The PRM algorithm works in two phases. First is the learning phase, during which a roadmap is built in the configuration space; and a query phase, during which the user-generated queries are answered. The reason for this division is, so the planner can capture the connectivity of the configuration space effectively, and the queries can then be answered more efficiently. This section explains the basic form of the PRM.

The roadmap of the basic PRM is represented by a graph $G = (V, E)$. The nodes $V$ correspond to configurations chosen randomly from a uniform distribution from the configuration space. The edges $E$ correspond to paths; these are collision-free paths connecting two configurations from $V$. In their simplest form, these edges are straight lines.

### Construction phase

Algorithm 1 describes the steps for the construction of the roadmap. In the beginning the graph $G = (V, E)$ is empty. Then a random free configuration is sampled, and added to the graph. The sampling is done using a random uniform distribution on the configuration space. The sampling is repeated until $n$ configurations have been added to the graph. For all nodes in the graph the $k$ closest neighbors are selected according to some distance metric *dist* (in this case it is the Euclidean distance between configurations). The algorithm then tries to connect each node to its neighbors (in this case drawing a straight line between the configurations). If the connection is successful (the edge is not in an obstacle) the edge between the two vertexes is added to the graph. Figure 5.1 shows a roadmap constructed in such a way.

**Algorithm 1** Roadmap Construction Algorithm from [4]

**Input**

$n$ : number of nodes to put in the roadmap

$k$ : number of closest neighbors to examine for each configuration

**Output**

A roadmap $G = (V, E)$

```
1:  V ← ∅
2:  E ← ∅
3:  while |V| < n do
4:      repeat
5:          q ← a random configuration in Q
6:      until q is collision-free
7:      V ← V ∪ {q}
8:  end while
9:  for all q ∈ V do
10:     N_q ← the k closest neighbors of q chosen from V according to dist
11:     for all q' ∈ N_q do
12:         if (q, q') ∉ E and Δ(q, q') ≠ NIL then
13:             E ← E ∪ {(q, q')}
14:         end if
15:     end for
16: end for
```

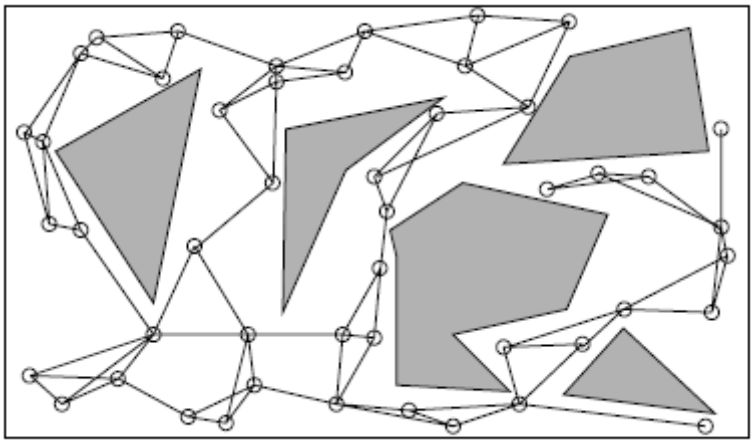


Figure 5.1: Example of PRM after the planning stage. The gray areas are obstacles. Empty circles are nodes. The straight lines are edges. [4]

### Query phase

In this phase the algorithm constructs paths between arbitrary configurations $q_{init}$ and $q_{goal}$. These configurations must be collision-free. This process is illustrated in Algorithm 2. The algorithm connects $q_{init}$ by considering its $k$ closest neighbors according to the

distance metric *dist*, it tries to connect each of them, until one connection is made. The same procedure is repeated for $q_{goal}$. If the connections are successful, the shortest path is determined using an algorithm like Dijkstra [5] or A* [8]. Figure 5.2 shows a path computed by this phase.

**Algorithm 2** Solve Query Algorithm from [4]

**Input**

$q_{init}$ : the initial configuration
$q_{goal}$ : the goal configuration
$k$ : the number of closest neighbors to examine for each configuration
$G = (V, E)$ : the roadmap computed by algorithm 1

**Output**

A path from $q_{init}$ to $q_{goal}$ or failure

$N_{q_{init}} \leftarrow$ the $k$ closest neighbors of $q_{init}$ from $V$ according to *dist*
$N_{q_{goal}} \leftarrow$ the $k$ closest neighbors of $q_{init}$ from $V$ according to *dist*
$V \leftarrow \{q_{init}\} \cup \{q_{goal}\} \cup V$
set $q'$ to be the closest neighbor of $q_{init}$ in $N_{q_{init}}$
**repeat**
    **if** $\Delta(q_{init}, q') \neq NIL$ **then**
        $E \leftarrow (q_{init}, q') \cup E$
    **else**
        set $q'$ to be the next closest neighbor of $q_{init}$ in $N_{q_{init}}$
    **end if**
**until** a connection was successful to the set $N_{q_{init}}$ is empty
set $q'$ to be the closest neighbor of $q_{goal}$ in $N_{q_{goal}}$
**repeat**
    **if** $\Delta(q_{goal}, q') \neq NIL$ **then**
        $E \leftarrow (q_{goal}, q') \cup E$
    **else**
        set $q'$ to be the next closest neighbor of $q_{goal}$ in $N_{q_{goal}}$
    **end if**
**until** a connection was successful to the set $N_{q_{goal}}$ is empty
$P \leftarrow$ shortest path$(q_{init}, q_{goal}, G)$
**if** $P$ is not empty **then**
    **return** $P$
**else**
    **return** failure
**end if**

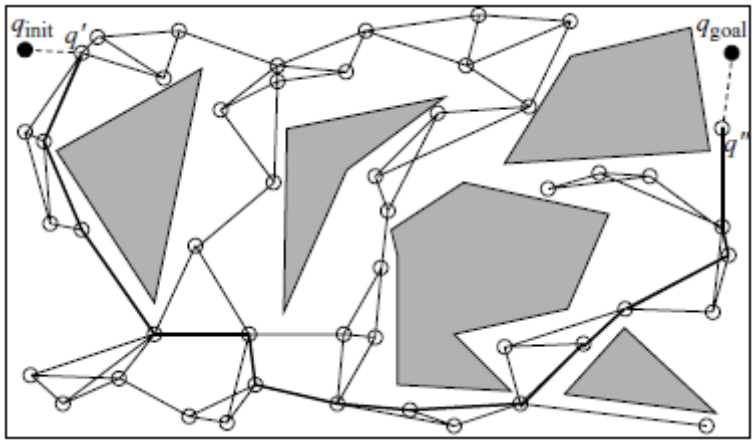



Figure 5.2: Example of PRM after the query stage. The configuration $q_{init}$ is connected to the graph at $q'$, and $q_{goal}$ is connected at $q''$. The found path is highlighted in thick black. [4]

#### Disadvantages of PRM

The algorithm does not give the optimal solution every time. Considering a configuration space with a large number of obstacles situated very close to each other. Assuming the gap between two obstacles is very narrow, and the system generates nodes using a random uniform distribution. The probability of generating nodes between those gaps is small. Increasing the number of nodes in the map does not solve this problem, as it exists because of the chosen distribution. When the system fails to generate a path for such configurations of the space, it cannot be determined whether it is because the path does not exist, or the number of vertexes is too small. This problem is solved by the RRT algorithm, explained in section 5.3.2.

## 5.3 Single-Query Sampling-Based Planners

There are instances in path planning, where the answer to a single query is desired instead of multiple. These queries are best solved using single-query planners, as they attempt to solve a query as fast as possible, and do not focus on exploring the entire configuration space. PRM described in 5.2 can be used as such a planner. It should check periodically if the query can be solved, and proceed to the query phase. The next two sections describe two planners, that were designed for single-query planning, these are ESTs and RRTs. The planners maintain two trees, one rooted at the start, and one rooted at the goal, and then grow the trees towards each other until they can be merged into one. In the construction step a new configuration is selected near the boundaries of the trees, and the planner tries to connect the configuration to some configuration in the tree. In the merging step, the planner tries to connect pairs of configuration selected from both trees, on the success of this step, the two trees became connected at a single point. This means, that a path from the start configuration exists, and now can be found in the merged tree.

To answer a single query efficiently, it is necessary to cover only parts of the configuration space, that are relevant for solving the query. EST and RRT both have a sampling strategies, that focus on exploring yet unexplored areas of the configuration space. This

means, that the generation of new samples is dependent on the generation of the previous samples, the start and goal positions.

### 5.3.1 Expansive-Spaces Trees

#### Construction of the two trees

Let $T_{init}$ and $T_{goal}$ be trees rooted at the start configuration $q_{init}$ and the goal configuration $q_{goal}$ respectively. First a random configuration from one of the trees is selected, let this be $q$, and let the probability of $q$ being selected be $\pi_T(q)$ and the tree it was selected from $T$. The planner then samples a random configuration from the neighborhood of $q$ from a random uniform distribution, let this configuration be $q_{rand}$. The planner attempts to connect $q$ to $q_{rand}$, if the connection is successful $q_{rand}$ is added to $T$, and $(q, q_{rand})$ is added to the edges of $T$. The pseudocode describing the tree construction can be found in algorithm 3 and algorithm 4. Figure 5.3 illustrates this method.

During the roadmap construction of PRM in section 5.2 a new configuration is always added to the graph, without checking whether it can be connected to the existing graph. In EST the new configuration is only added to a tree, if it can be connected to that tree, therefore there is a path between all configurations within a tree.

**Algorithm 3** Build EST Algorithm from [4]

**Input**

$q_0$ : the configuration where the tree is rooted

$n$ : number of attempts to expand the tree

**Output**

A tree $T = (V, E)$, that is rooted at $q_0$ and has $\leq n$ configurations

1: $V \leftarrow \{q_0\}$
2: $E \leftarrow \emptyset$
3: **for** $(i = 1$ to $n)$ **do**
4: $\quad q \leftarrow$ a randomly chosen configuration from $T$ with probability $\pi_T(q)$
5: **end for**
6: **return** $T$

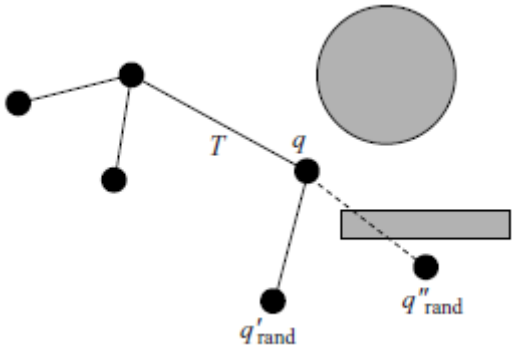



Figure 5.3: Adding a new configuration to EST. $T$ is the tree, $q$ is the configuration to be expanded, $q'_{rand}$ and $q''_{rand}$ are configurations to be added. [4]

**Algorithm 4** Extend EST Algorithm from [4]

**Input**

$T = (V, E)$ : an EST

$q$ : a configuration from which to grow $T$

**Output**

A new configuration $q_{new}$ in the neighborhood of $q$, or NIL in case of failure

1: $q_{new} \leftarrow$ a random collision-free configuration from the neighborhood of $q$
2: **if** $\Delta(q, q_{new})$ **then**
3: $\quad V \leftarrow V \cup \{q_{new}\}$
4: $\quad E \leftarrow E \cup \{(q, q_{new})\}$
5: $\quad$ **return** $q_{new}$
6: **end if**
7: **return** NIL

### Configuration weighting

EST relies on its ability not to over sample a region of the configuration space. This is most pronounced, when sampling in the neighborhood of $q_{init}$ and $q_{goal}$. Because of this, the probability density function $\pi_T$ must be given careful consideration. A function, which generates samples in the neighbourhood of configurations, which are sparse is preferred. In practice, one approach, that works well associates each configuration $q$ with a weight $w_T(q)$, that counts the number of configuration within its neighborhood. If $\pi_T$ is inversely proportional to $w_T(q)$, then the configurations with sparse neighborhoods have a higher chance to be picked for expansion.

### Merging of the two trees

The merging of $T_{init}$ and $T_{goal}$ can be achieved by pushing the construction of $T_{init}$ towards $T_{goal}$. After the planner expanded the tree using the method described in 4, the planner attempts to connect $q$ to its closest $k$ configurations in $T_{goal}$. If a connection is successful,

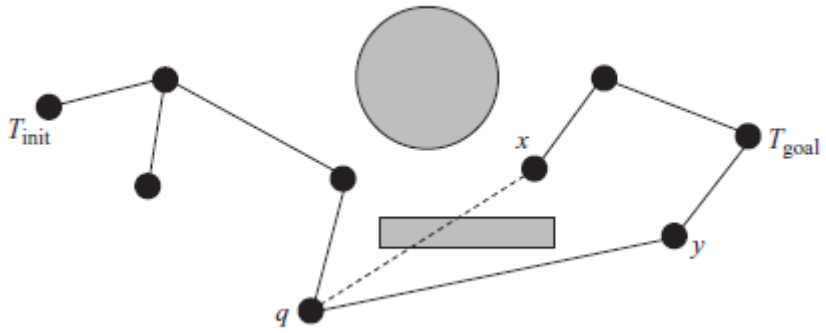



Figure 5.4: Merging two EST trees. The configuration $q$ is just added to $T_{init}$. The planner connected $q$ to $y$ in the second tree $T_{goal}$. [4]

the two trees are merged. Otherwise the trees are swapped, and the process is repeated. This is illustrated in figure 5.4.

Another way of connecting the two trees is by only growing one of them, but instead of generating only one configuration from the neighborhood of $q$, the planner generates $k$ new configurations instead. The planner then attempts to connect these configurations one-by-one to $T_{init}$. After this a new configuration is chose for expansion. First the planner checks, whether this randomly selected configuration can be connected to $q_{goal}$, and if the connection is successful, the two trees are merged.

### 5.3.2 Rapidly-Exploring Random Trees

#### Construction of the two trees

Let $T_{init}$ and $T_{goal}$ be trees rooted at the start configuration $q_{init}$ and the goal configuration $q_{goal}$ respectively. Since both trees are extended, let $T$ denote the tree, that is currently being extended. First, a random collision-free configuration is selected from a uniform random distribution from the configuration space, let this be $q_{rand}$. Then the nearest configuration from $T$ to $q_{rand}$ is found, let this configuration be $q_{near}$. If $q_{rand}$ is closer to $q_{near}$ than $step_size$, the planner check whether a connection between $q_{near}$ and $q_{rand}$ can be established. If that is the case $q_{rand}$ is added to the vertexes of $T$, and the edge $(q_{rand}, q_{near})$ is added to the edges of $T$.

In the case $q_{rand}$ is farther from $q_{near}$ than $step_size$, the planner moves from $q_{near}$ a distance of $step_size$ towards $q_{rand}$, let the configuration the planner arrived at be $q_{new}$. The planner check whether a connection between $q_{near}$ and $q_{new}$ can be established. If that is the case $q_{new}$ is added to the vertexes of $T$, and the edge $(q_{new}, q_{near})$ is added to the edges of $T$, and $q_{new}$ becomes the new $q_{near}$. This process is described in the pseudocode given in algorithms 5 and 6. The process is illustrated in figure 5.5.

a greedier approach can also be considered. The process of stepping towards $q_{rand}$ can be repeated until either $q_{rand}$ is reached, or the planner cannot establish a connection between $q_{new}$ and $q_{near}$. This is described in algorithm 7.

**Algorithm 5** Build RRT Algorithm from [4]

**Input**

$q_0$ : the configuration where the tree is rooted

$n$ : the number of attempts to expand the tree

**Output**

A tree $T = (V, E)$, that is rooted at $q_0$ and has $\leq n$ configurations

```
V ← {q_0}
E ← ∅
for i = 1 to n do
    q_rand ← a randomly chosen free configuration
    extend RRT (T, q_rand)
end for
return T
```

**Algorithm 6** Extend RRT Algorithm from [4]

**Input**

$T = (V, E)$ : an RRT

$q$ : a configuration toward which the tree $T$ is grown

**Output**

A new configuration $q_{new}$ toward $q$, or NIL in case of failure

```
q_near ← closest neighbor of q in T
q_new ← progress q_near by step_size along the straight line in Q between q_near and q_rand
if q_new is collision-free then
    V ← V ∪ {q_new}
    E ← E ∪ {(q_near, q_new)}
    return q_new
end if
return NIL
```

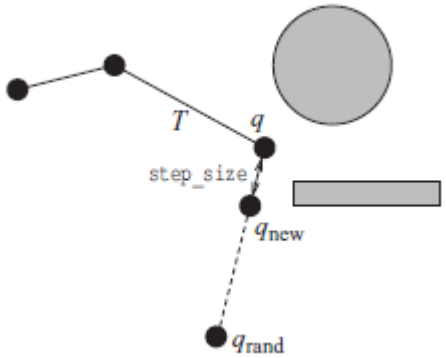



Figure 5.5: Adding a new configuration to RRT. Configuration $q_{rand}$ is selected randomly. Configuration $q$ is the closest configuration in $T$ to $q_{rand}$. Configuration $q_{new}$ is obtained by moving $q$ by $step_size$ toward $q_{rand}$. [4]

**Algorithm 7** Connect RRT Algorithm from [4]

**Input**
$T = (V, E)$ : an RRT
$q$ : a configuration toward which the tree $T$ is grown
**Output**
connected if $q$ is connected to $T$, failure otherwise

1: **repeat**
2: $q_{new} \leftarrow$ extend RRT $(T, q)$
3: **until** ($q_{new} = q$ **or** $q_{new}$ =NIL)
4: **if** $q_{new} = q$ **then**
5: **return** connected
6: **else**
7: **return** failure
8: **end if**

### Sampling bias

In the base version of RRT the planner selects a configuration from a uniform random distribution. This can be improved by considering a sampling function, that is biased towards $q_{goal}$. An extreme approach would be to set $q_{rand}$ to $q_{goal}$ anytime a random configuration is generated. This would introduce too much bias, and the RRT would get stuck, therefore a suitable sampling function is one, that alternates, based on some kind of probability distribution, between uniform sampling and biased sampling. Experimental evidence has shown that setting $q_{rand}$ to $q_{goal}$ with probability $p$, or randomly generating $q_{rand}$ with probability 1 - $p$ from a uniform distribution, works well. Even for small values of $p$, such as 0.05, the tree rooted at $q_{init}$ converges much faster to $q_{goal}$ than when just uniform sampling is used [4].

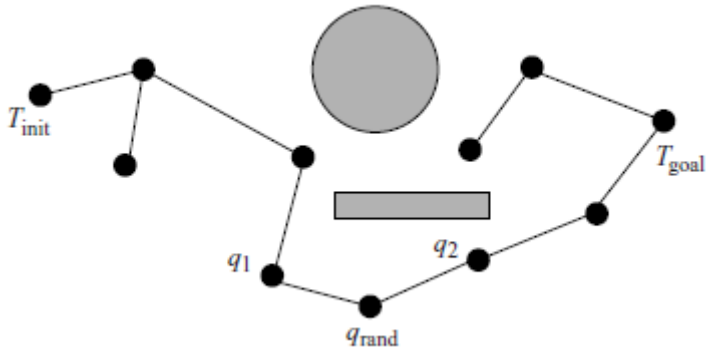



Figure 5.6: Merge two RRTs. Configuration $q_{rand}$ is generated randomly. Configuration $q_1$ was extended to $q_{rand}$. Configuration $q_2$ is the closest configuration to $q_{rand}$ in $T_{goal}$. [4]

### Merging of the two trees

Let $T_{init}$ and $T_{goal}$ be trees rooted at the start configuration $q_{init}$ and the goal configuration $q_{goal}$ respectively. The planner grows both trees toward each other, let $T$ denote the tree, that is currently being expanded. First, a random configuration is generated from the configuration space, let this configuration be $q_{rand}$ . RRT extends $T$ towards $q_{rand}$, and obtains a new configuration $q_{new}$. Then the planner extends the other tree towards $q_{new}$. If the planner reaches $q_{new}$, that means the trees are merged at $q_{new}$, and the algorithm is terminated. Otherwise the two trees are swapped, and the process is repeated $\ell$ times. The algorithm pseudocode for this is presented in algorithm 8. Figure 5.6 illustrates this.

**Algorithm 8** Merge RRT Algorithm from [4]

**Input**
$T_1$ : first RRT
$T_2$ : second RRT
$\ell$ : number of attempts allowed to merge $T_1$ and $T_2$
**Output**
merged if the two RRTs are connected to each other, failure otherwise

```
for i = 1 to ℓ do
    q_rand ← a randomly chosen free configuration
    q_new,1 ← extend RRT (T_1, q_rand)
    if q_new,1 ≠ NIL then
        q_new,2← extend RRT(T_2, q_new,1)
        if q_new,1 = q_new,2 then
            return merged
        end if
        SWAP(T_1, T_2)
    end if
end for
return failure
```

# Chapter 6

# Description of the application

This chapter describes the implemented application demonstrating the algorithms described in the previous chapters. Section 6.1 describes the library used for the implementation of the algorithm. Section 6.2 describes the user interface of the application. Section 6.3 contains specifics about the individual algorithms. Section 6.4 describes, how the application was tested, it also describes the results of the tests.

## 6.1 The Vizlib library

To help with the implementation of the algorithms described in the previous chapters, the *vizlib* library was chosen. This is a library implemented by Jakub Rusnák as part of his master's thesis [9]. This library abstracts parts of the user interface – views, which allow the programmer to quickly show how the algorithms work. There are five different types of views implemented in the library:

- MapView - Visualizes the configuration space 6.1
- CodeView - Contains the pseudocode of the algorithm, it also allows for step-by-step highlighting of a single line 6.2
- ConsoleView - Allows the programmer to display text information about the currently executed step 6.3
- ToolbarView - Allows the user to control the simulation using buttons, or edit the map 6.4
- ParameterView - Allows the user to edit the parameters of the algorithm, also contains presets 6.5

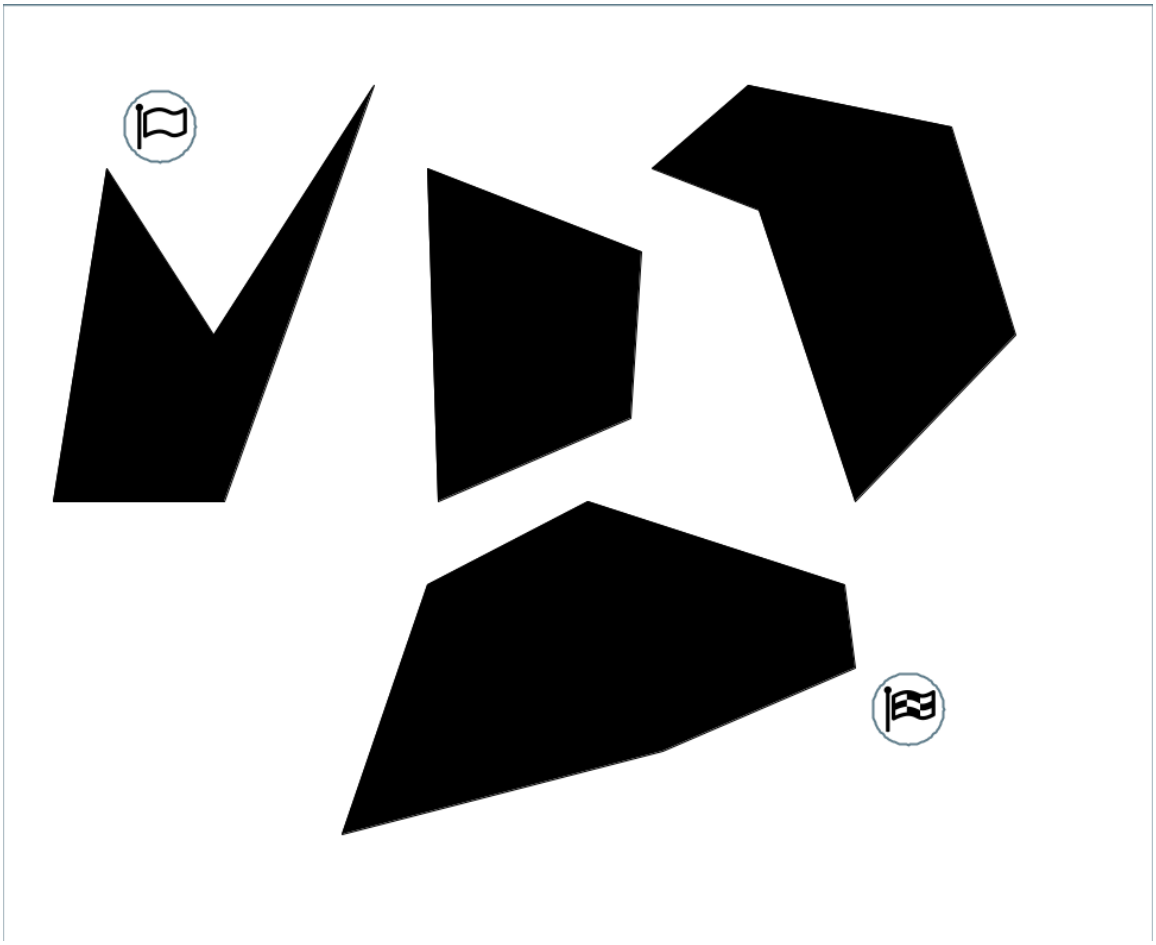


Figure 6.1: MapView

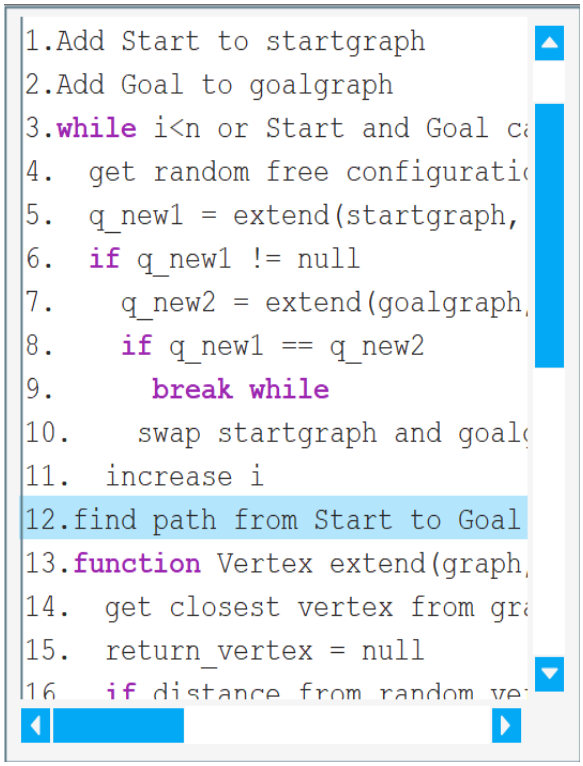



Figure 6.2: CodeView

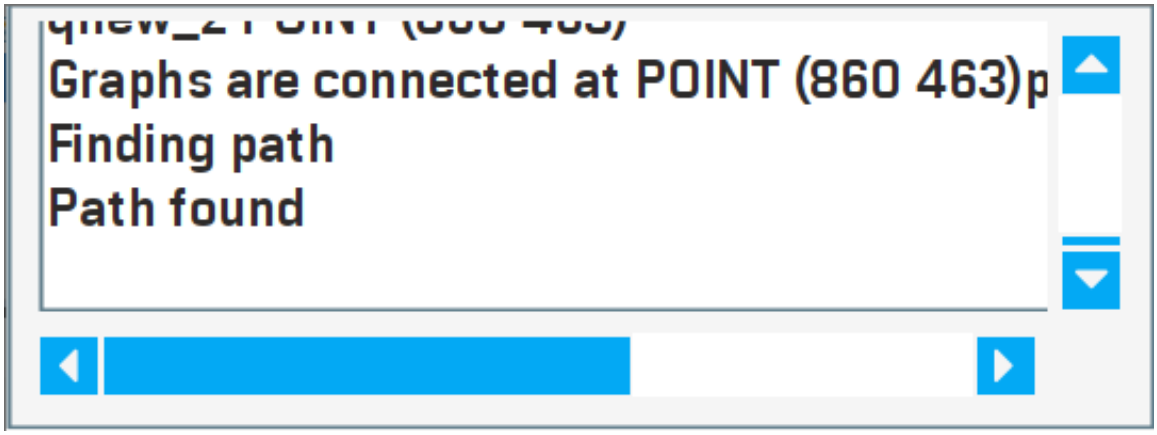



Figure 6.3: ConsoleView


Figure 6.4: ToolbarView

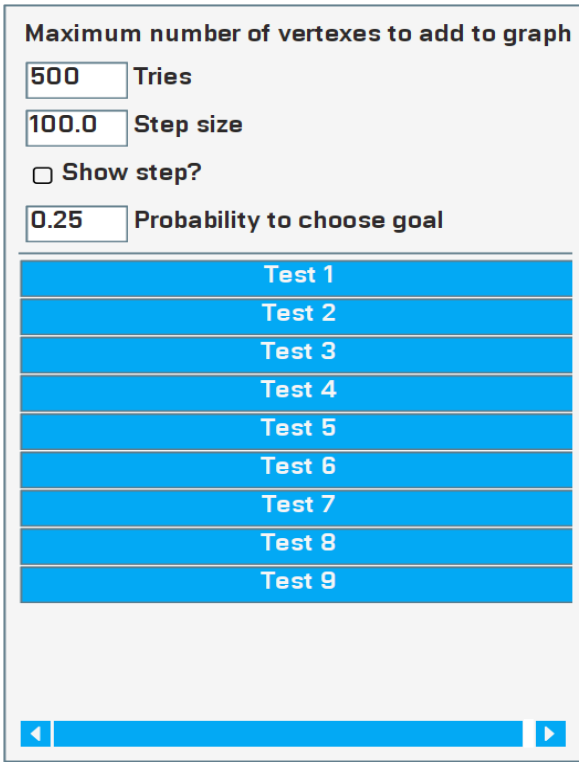



Figure 6.5: ParameterView

The application has two modes: editing and simulation. In editing mode it is possible to create a map with obstacles using the buttons in the *ToolbarView*, or chose one of the presets, and change algorithm parameters in the *ParameterView*. The editing mode is shown in 6.6.

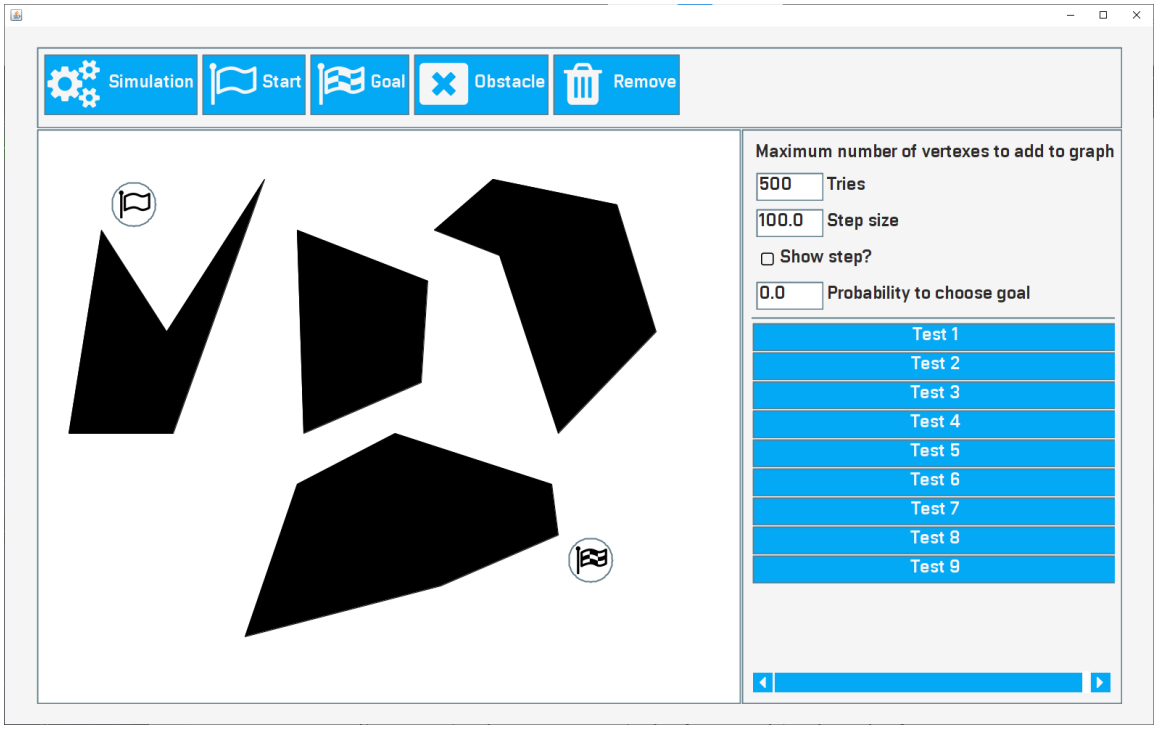



Figure 6.6: Application editing mode

In simulation mode the running of the algorithm itself is visualized. On the right side in the *CodeView* the pseudocode of the algorithm is displayed, and the *ConsoleView* displays information about the currently executed step. The main reason for using this library is to implement the step-by-step execution of algorithms, that is the most important element in regards to this thesis. The simulation mode is shown in 6.7.

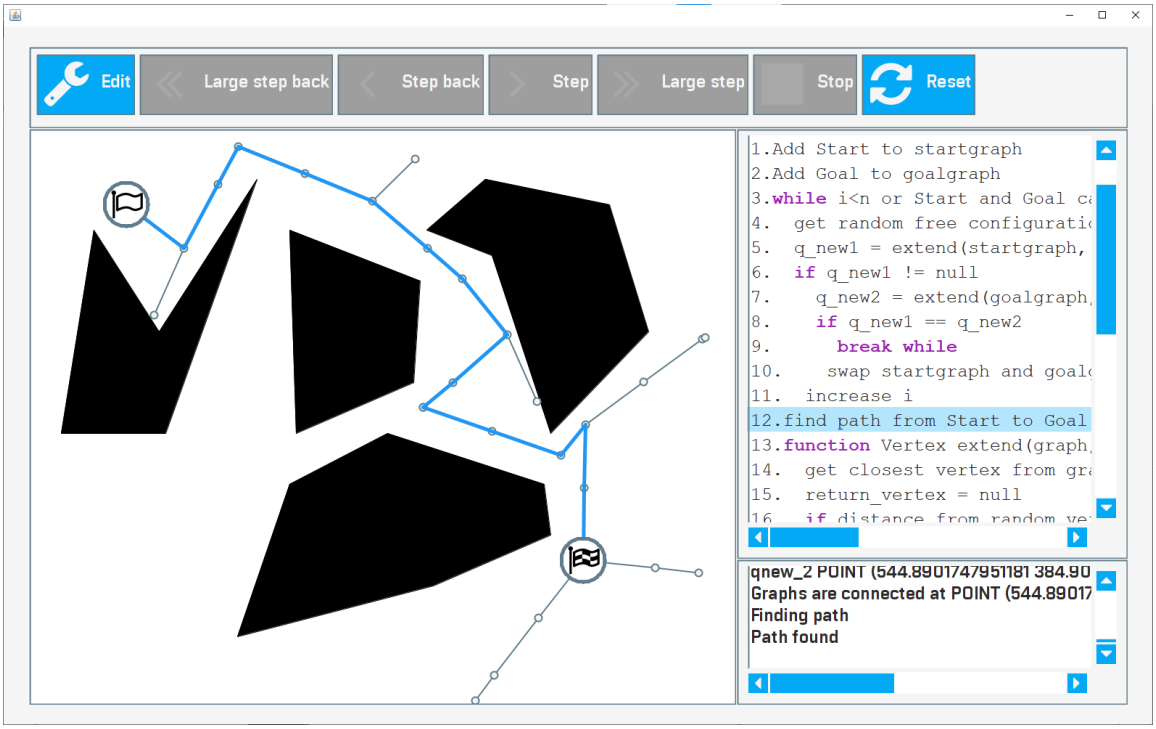



Figure 6.7: Application simulation mode

The work of Jakub Rusnák contains implementations of four path planning algorithm, that fall under the scope of this thesis, these are Visibility Graph, PRM, EST and RRT [9]. These implementations were not used in the making of the application for two main reasons. The first reason is, that they are decentralized, meaning each of the algorithms can only be run individually, the specifics about how this problem was solved can be found in section 6.2. The second reason is, that this thesis specifically deals with the implementation and visualization of path planning algorithms, therefore I decided, that implementing my own version of the algorithms in question is a better solution. The algorithms can be implemented in such a way, so that the first problem is solved at the same time.

## 6.2 Application user interface

During the design process for the user interface, multiple designs were considered, this section describes why was the final one chosen over the others.

The original idea was for the whole application to be displayed in one window. This approach has led to problems however. The view responsible for the changing of the algorithms would have had to be synchronised across all of them. This was not feasible due to the design of the *vizlib* library. In the library, when an object of type *mainPanel* is created, the views the *mainPanel* has registered for use have to be specified when calling its constructor. It is impossible to construct all *mainPanel* objects simultaneously, the view responsible for the changing of the *mainPanels* would therefore be different at the construction of each *mainPanel*. The view would also have a different name in each of the *mainPanel* implementations, leading to further complications. The *mainPanels* would try to access the view simultaneously, this leads to *ConcurrentModificationException* being thrown at every *mainPanel* change.

Due to the complications described above I have decided on a different design. The changing of algorithms is handled in a separate window from the window, that displays the algorithms. This design solves the problems mentioned in the above paragraph. An example of the two application windows is shown in 6.8.

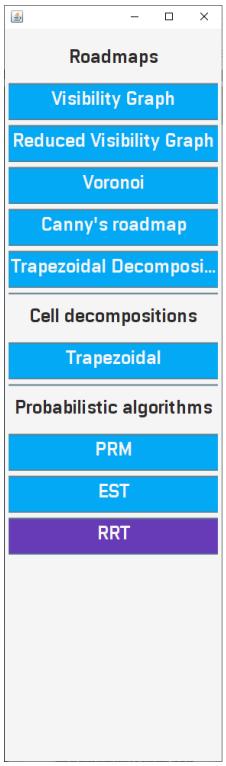



Figure 6.8: Second window of the application

## 6.3 Specific notes about individual algorithms

During the implementation of the algorithms, great care was taken to ensure, that they resemble their theoretical description described in the previous chapters. This was done to allow seamless transition from theory to practice and simple experimentation with algorithms, which are described in textbooks. The subsections of this section are dedicated to differences from the descriptions, and to the explanations of the necessity of these differences.

### Voronoi Diagram

The algorithm described in section 3.2 traverses the entire configuration space to make a map of it. This means, that the implemented algorithm must calculate whether the current point of the configuration space is part of the Voronoi diagram or not. In order to lessen the number of steps needed to construct the graph, a grid was imposed on the configuration space. The algorithm calculates, whether there is point in the cell, that is part of the Voronoi diagram. The size of each cell is 5 pixels * 5 pixels, this size was determined experimentally, with this size the algorithm does not get stuck inside smaller holes, and runs approximately 25 times faster. The planner also constructs the graph slice-by-slice

similar to Canny's roadmap algorithm described in 3.3, this leads to further reductions in computing time.

The method described in chapter 3.2 also requires the configuration space to be bounded in some way. The *vizlib* library does not have a function to make this possible. This is solved by adding obstacles to the edges of the configuration space for each of the presets. The user is free to move these bounds as they see fit. The other restriction imposed onto the algorithm, is that it only works within the bounds of the configuration space, but it does not treat these bounds as obstacles. If the user deletes all the boundaries, and only has a single obstacle in the configuration space, all points of the configuration space are equal distance apart from other obstacles (as the is only one), and are therefore part of the Voronoi diagram.

### Canny's Roadmap algorithm

The algorithm described in section 3.3 considers slices for all values of the $x$ coordinate of the configuration space. The algorithm implemented only considers every fifth slice of the configuration space, as well as the slices containing the vertexes of obstacles, this simplification can be done, due to the fact, that all obstacles are polygonal. This can reduce the time to construct the graph by up to a factor of 5 (if every slice contains a vertex, there is no change in computation time). By only leaving out slices, that do not contain vertexes, it is ensured, that the algorithm does not generate a graph, that intersects obstacles, and by visiting every slice containing a vertex, it is ensured, that the points of the graph are connected.

The same statements about the bounds of the configuration space described in the previous subsection apply to this algorithm as well.

### Basic Probabilistic Roadmap

The algorithm described in section 5.2 is a multi-query algorithm. The implemented algorithm, however is the single-query version of PRM, this change was necessary due to the fact, that the *vizlib* library only allows the user to set a single start, and a single goal point [9].

### Expansive-Spaces Trees

In order to implement a graph with weighted nodes described in the Configuration weighting section of 5.3.1, a weighted graph is required. The *vizlib* library does not have such graph, so it had to be implemented. The graph is implemented in the file *WeightedGraph.java*, and the weighted vertexes are implemented in *WeightedVetex.java*. The weight calculation is done by the function named *recalculateWeights()*.

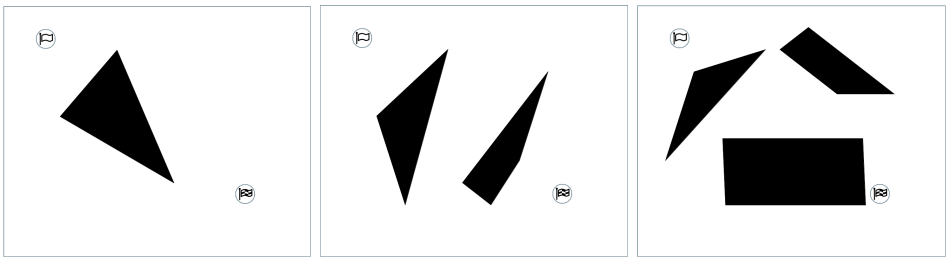
## 6.4 Testing the application

In this chapter, the comparison and testing of the algorithms is described. Testing was done using a batch for hand generated maps. There are nine maps in total, they contain obstacles increasing in number (from left to right on 6.9), and increasing complexity (from top to bottom on 6.9). These maps can be found as presets for each algorithm, in the respective algorithm's *ParameterView*. Figure 6.9 illustrates these maps.

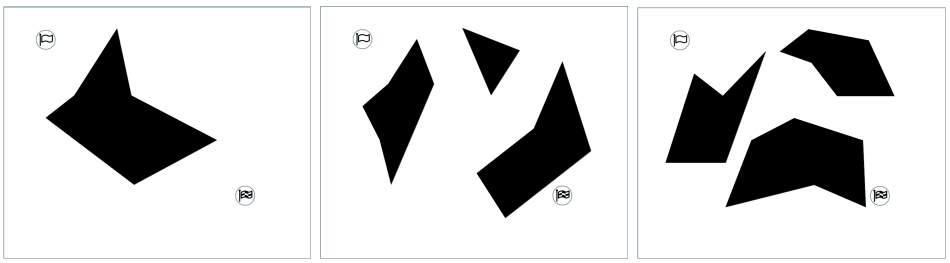
In this chapter I have attempted to answer the following questions:

- Which of the deterministic roadmap algorithms generates the shortest path?
- Which of the probabilistic roadmap algorithms generates the shortest path?
- How do deterministic and probabilistic roadmaps compare in terms of path length?
- What effect does the different sampling methods of EST and RRT have on the number of nodes in the generated graph?
- Given equal amounts of random vertexes PRM or RRT has a higher success rate?
- What effect does the probability of choosing goal instead of a random configuration in RRT have on the number of random configurations needed to reach the goal?

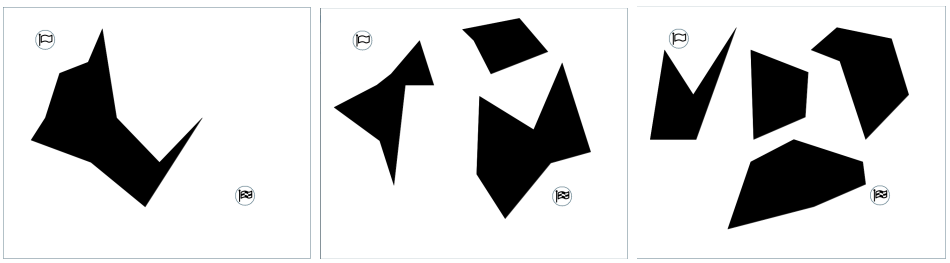
Unless otherwise stated, to search the graphs, the A* algorithm is used with Euclidean distance as the heuristic, as it finds the paths faster than Dijkstra's algorithm, for known start and goal positions [7].



(a) Test1 (b) Test2 (c) Test3



(d) Test4 (e) Test5 (f) Test6



(g) Test7 (h) Test8 (i) Test9

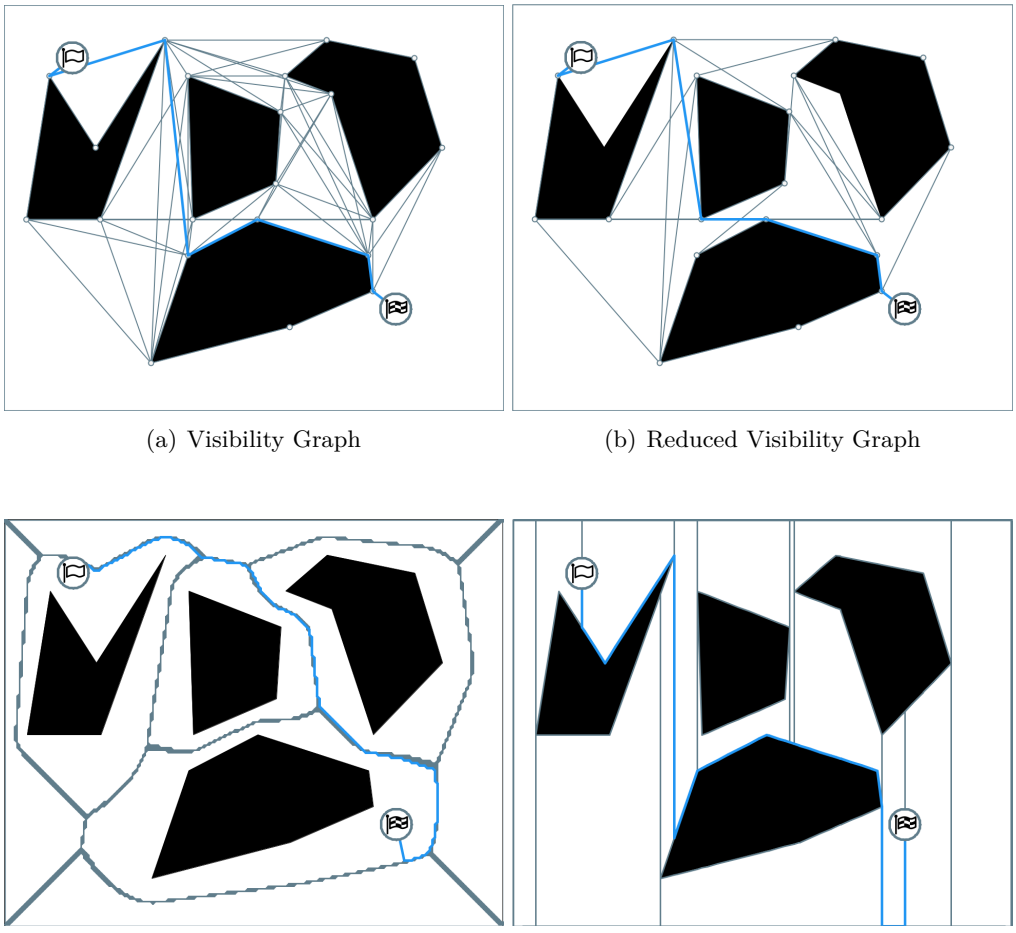
Figure 6.9: The maps the tests were conducted on

### Deterministic algorithm comparison

The algorithms compared in this section are the following:

- Visibility Graph
- Reduced Visibility Graph
- Generalized Voronoi Diagram
- Canny's Roadmap Algorithm

In the context of navigation for deterministic algorithms, one of the most important indicator of the algorithm's performance is the length of the path found. Figure 6.10 illustrates the paths found on Test9, and the table 6.1 contains the results of all tests.



(a) Visibility Graph

(b) Reduced Visibility Graph

(c) Generalized Voronoi Diagram

(d) Canny's Roadmap Algorithm

Figure 6.10: Maps generated using Test9

The results show, that the Reduced Visibility Graph provides the shortest path, followed by the Visibility Graph, followed by the Generalized Voronoi Diagram, followed by Canny's Roadmap Algorithm. This confirms the relation between the Visibility Graph and the Reduced Visibility Graph. These algorithms move near the walls of the obstacles, in practice this may not always be desirable. Canny's Roadmap Algorithm avoids the obstacles inside

| Tests | Visibility Graph | Reduced Visibility Graph | Generalized Voronoi Diagram | Canny's Roadmap Algorithm |
|---|---|---|---|---|
| Test1 | 912.36 | 912.36 | 1083.34 | 1516.0 |
| Test2 | 1150.67 | 1150.67 | 1600.34 | 1516.0 |
| Test3 | 1260.95 | 1260.95 | 1245.77 | 1848.34 |
| Test4 | 1114.1 | 895.37 | 1211.14 | 1516.0 |
| Test5 | 1463.66 | 1306.49 | 1492.22 | 1780.57 |
| Test6 | 1303.95 | 1303.95 | 1214.26 | 1941.62 |
| Test7 | 1009.91 | 1009.91 | 1176.19 | 1859.74 |
| Test8 | 1237.59 | 1311.87 | 1459.03 | 2052.46 |
| Test9 | 1221.76 | 1127.43 | 1334.7 | 2053.32 |
| Σ | 10674.95 | 10279.0 | 11817.01 | 16084.05 |

Table 6.1: Results of the tests comparing the lengths of paths found.

the bounds of the configuration space, and follows the bounds instead, and the Generalized Voronoi Diagram aims to avoid going near the walls as much as possible. The results also show, that the more obstacles there are, the longer the paths are on average, and the same goes for the complexity of the obstacles.

### Probabilistic algorithm comparison

The algorithms compared in this section are the following:

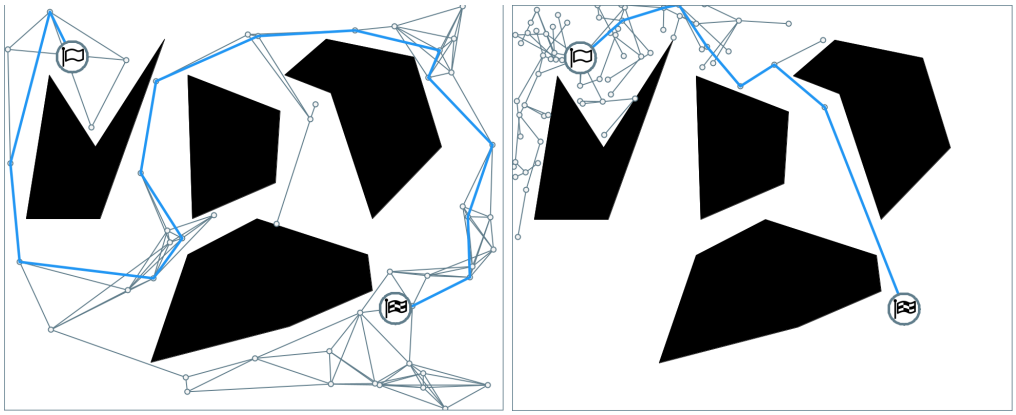
- Basic PRM
- EST
- RRT

In the context of navigation for probabilistic algorithms, one of the most important indicator of the algorithm's performance is the length of the path found. Figure 6.11 illustrates a path found on Test9, and the table 6.2 contains the results of all tests.

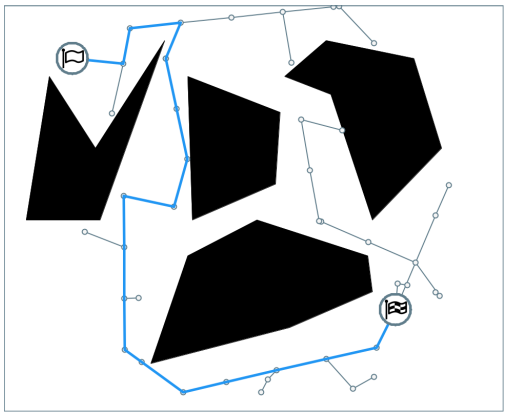
The algorithms were ran 100 times for each test using the following parameters:

- Basic Probabilistic Roadmap - the number of nodes generated was set to 50, the maximum number of connections per node was set to 5, in case of failure, the algorithm was ran again
- Expansive-Spaces Trees - the maximum distance between vertexes was set to 100 pixels, and the random number of nearby nodes generated was set to 5
- Rapidly-Exploring Random Trees - the maximum distance between vertexes was set to 100 pixels, the maximum number of tries was ignored, so the algorithm always produces a path, and the probability to select the goal instead of a random configuration was set to 0.0.

The tests show, that the EST and RRT algorithms generate significantly shorter paths compared to PRM, this confirms, that guided sampling reduces the length of the generated paths. In environments with less obstacles EST performed the best, in environments with



(a) Probabilistic Roadmap (b) Expansive-Spaces Trees



(c) Rapidly-Exploring Random Trees

Figure 6.11: Maps generated using Test9

| Tests | Basic Probabilistic Roadmap | Expansive-Spaces Trees | Rapidly-Exploring Random Trees |
|---|---|---|---|
| Test1 | 1530.02 | 1007.76 | 1189.24 |
| Test2 | 1514.04 | 1423.04 | 1417.22 |
| Test3 | 1444.15 | 1465.08 | 1261.48 |
| Test4 | 1359.93 | 1210.56 | 1287.25 |
| Test5 | 1558.14 | 1515.08 | 1459.28 |
| Test6 | 1468.0 | 1439.16 | 1307.6 |
| Test7 | 1375.67 | 1216.04 | 1396.16 |
| Test8 | 1553.59 | 1565.28 | 1510.48 |
| Test9 | 1538.88 | 1446.12 | 1418.11 |
| $\Sigma$ | 13342.42 | 12288.12 | 12246.82 |

Table 6.2: Results of the tests comparing the average lengths of paths found.

more number of obstacles RRT generated the shortest paths. The complexity of the obstacles increased the length of paths, and had the most significant effect on the EST algorithm. During testing two additional observation were made. The first is, that the PRM fails more

often, the more obstacles are in the configuration space. The second is, that the complexity of the configuration space disproportionately impacted the runtime of EST.

### Comparison of the deterministic and probabilistic algorithms

The results from the previous two sections confirm, that deterministic algorithms generate shorter paths, than their probabilistic counterparts. Canny's roadmap algorithm is an interesting outlier in the comparisons, as it has generated paths, that are on average 1.2 times longer than even PRM. The deterministic algorithms consistently generated shorter paths across 8 of the 9 test cases, except in Test2, where the generated a path in the Generalized Voronoi Diagram goes in-between the two obstacles instead of going around them. Later investigation revealed, that this is due to the A* algorithm, that was used to find the path.

### Comparison of sampling strategies

This test compares the two sampling strategies used in the EST and RRT algorithms. The EST uses a sampling strategy, that generates samples near the tree rooted in the start point, and RRT generates samples from a uniform random distribution, and tries to extend the tree towards the sample. The number of nodes in each of the generated graphs was counted, and the results averaged for each testing map. Table 6.3 contains the results.

The algorithms were ran 100 times for each test using the following parameters:

- Expansive-Spaces Trees - the maximum distance between vertexes was set to 100 pixels, and the random number of nearby nodes generated was set to 5
- Rapidly-Exploring Random Trees - the maximum distance between vertexes was set to 100 pixels, the maximum number of tries was ignored, so the algorithm always produces a path, and the probability to select the goal instead of a random configuration was set to 0.0.

| Tests | Expansive-Spaces Trees vertexes | Rapidly-Exploring Random Trees vertexes | Ratio of vertexes generated |
|---|---|---|---|
| Test1 | 21.88 | 18.25 | 1.2 |
| Test2 | 77.8 | 23.56 | 3.3 |
| Test3 | 107.72 | 23.96 | 4.5 |
| Test4 | 48.76 | 27.44 | 1.78 |
| Test5 | 129.04 | 27.44 | 4.7 |
| Test6 | 121.44 | 27.68 | 4.39 |
| Test7 | 49.68 | 36.68 | 1.35 |
| Test8 | 127.52 | 47.88 | 2.66 |
| Test9 | 99.4 | 36.52 | 2.72 |
| average | 87.03 | 29.93 | 2.9 |

Table 6.3: Results of the tests comparing the average number of vertexes in the graphs.

The tests show, that the sampling strategy used by RRT generates trees, that have fewer nodes in them on average by a factor of 2.9, this is to be expected as is RRT a configuration is only added to the tree, if it is situated in the direction of the randomly generated configuration. This trend becomes more pronounced, the more complex the configuration space is, the number of objects has a bigger effect on this than their complexity.

### PRM and RRT success rate comparison

This section compares the success rates of PRM and RRT, given the same number of randomly generated vertexes. Both algorithms utilize the same sampling strategy, but use the samples in a different way. The PRM algorithm always adds them to its generated roadmap, while the RRT extends its current roadmap towards them. The table 6.4 contains the results.

The algorithms were ran 100 times for each test using the following parameters:

- Basic Probabilistic Roadmap - the number of nodes generated was set to 50, the maximum number of connections per node was set to 5.
- Rapidly-Exploring Random Trees - the maximum distance between vertexes was set to 100 pixels, the maximum number of tries was set to 50, and the probability to select the goal instead of a random configuration was set to 0.0.

The reason for the low number of tries per run is, so the differences between the two algorithms become more pronounced.

| Tests | Basic Probabilistic Roadmap success rate | Rapidly-Exploring Random Trees success rate |
|---|---|---|
| Test1 | 1.0 | 1.0 |
| Test2 | 0.86 | 0.95 |
| Test3 | 0.81 | 0.95 |
| Test4 | 0.91 | 0.91 |
| Test5 | 0.85 | 0.89 |
| Test6 | 0.83 | 0.93 |
| Test7 | 0.79 | 0.84 |
| Test8 | 0.74 | 0.61 |
| Test9 | 0.49 | 0.46 |

Table 6.4: Results of the tests comparing the success rates for the algorithms

The test show, that for most cases RRT produces results with higher consistency than PRM. The paths generated by the RRT were also shorter than those generated by PRM. The cases of Test8 and Test9 are outliers from this trend, but this test did not take into account, that adding a node to the graph of PRM a more complex operation, than adding a node to the graph of RRT. For PRM the 5 closest neighbors have to be determined first, before the edge can be added, while RRT only determines the closest neighbor once.

### Effect of the introduction of a bias into the probability density function of RRT

This section attempts to test what effect does the probability of choosing goal as a random configuration in RRT have on the number of random configurations needed to reach the goal. For the remainder of the section, this probability is referred to as $p$. The test conducted in the previous section showed, that from all available configurations the algorithm had the smallest probability of finding a path on Test9, therefore this is the preset, that is used for this test. The table 6.5 contains the results.

The algorithms was ran 100 times for each tested value of $p$, the limit on the number of generated random configurations was ignored, the maximum distance between nearest vertexes was set to 100.0.

| $p$ | average number of random configurations | median of random configurations |
|---|---|---|
| 0 | 67.02 | 57 |
| 0.05 | 75.54 | 52 |
| 0.1 | 86.28 | 80 |
| 0.15 | 74.52 | 51 |
| 0.2 | 75.58 | 60 |
| 0.25 | 104.46 | 82 |

Table 6.5: Results of the tests containing the number of average tries, and the medians for each number of tries for each $p$

The dispersion of the results was very big, so the medians of the numbers of generated random configurations are also compared. The average values do not match the predictions outlined in [4]. The median values however follow the assumptions outlined in [4] with one exception for $p = 0.1$, but this can be attributed to the low number of tests conducted. From the test, two observations can be made. First, the value of $p$ indeed has an effect on the generated graph, as stated in [4]. Second, there is an optimal value of $p$ for this configuration, and this number is between 0.05 and 0.1. At the value $p = 0.25$ the average and median both start to become exponentially bigger, and start to approach infinity as $p$ approaches 1.0.

# Chapter 7

# Conclusion

In this thesis, path planning algorithms were studied. A total of nine algorithms were implemented. Five for generating deterministic roadmaps, one for generating a coverage path and three for generating probabilistic roadmaps.

In addition, an application was created in Java that allows step-by-step execution of these algorithms, so the user can form a better understanding of these algorithms. A study was also created from the results obtained during the testing of these algorithms.

The study shows, that among the deterministic algorithms the Reduced Visibility Graph produces the shortest paths for the test data. Among the probabilistic algorithms Expansive-Spaces Trees (EST) produced the shortest paths for a small number of obstacles, as the number of objects increases, the Rapidly-Exploring Random Trees (RRT) algorithm produces shorter paths. The deterministic algorithms on average produce shorter paths than the probabilistic algorithms, with the exception of Canny's roadmap algorithm, which produced longer paths than every other algorithm. The different sampling methods of EST and RRT were compared. This comparison has proven, that RRT's sampling method generates graphs with fewer nodes. The success rate of the Probabilistic Roadmap Algorithm (PRM) and RRT was also compared. The algorithms were given the same amount of samples. The test results show, that on average RRT has a higher success rate, and generates shorter paths. The effect of the probability of choosing goal instead of a random configuration in RRT has on the number of random configurations needed to reach the goal was also studied. The tests show, that the optimal probability to chose the goal is between 0.05 and 0.15. Probabilities larger than 0.2 increased the number of random configurations needed.

Within the study of the described approaches, there are frequent examples of environments with an external obstacle, an obstacle limiting the outer boundary of the available space. The *vizlib* library does not allow for creation of such obstacles, a workaround was developed to solve the problem, because extending the library with this function is not trivial. The next step in this work would be, to increase the robustness of the implemented algorithms, and to extend the library by the mentioned with the probability of adding external obstacles. The application should further be extended by the implementation of algorithms from other authors. Implementing the Opportunistic Path Planner, the Boustophedon Decomposition and the Sampling-Based Roadmap of Trees algorithms found in [4] would be a good idea.

# Bibliography


[1] Acar, E. U., Choset, H., Rizzi, A. A., Atkar, P. N. and Hull, D. Morse decompositions for coverage tasks. *International Journal of Robotics Research*. April 2002.

[2] Aurenhammer, F. Voronoi Diagrams—a Survey of a Fundamental Geometric Data Structure. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. sep 1991, vol. 23, no. 3, p. 345–405. DOI: 10.1145/116873.116880. ISSN 0360-0300. Available at: https://doi.org/10.1145/116873.116880.

[3] Canny, J. F. *The Complexity of Robot Motion Planning*. Cambridge, MA, USA: MIT Press, 1988. ISBN 0262031361.

[4] Choset, H. *Principles of Robot Motion: Theory, Algorithms, and Implementation*. Prentice Hall of India, 2005. ISBN 9780262033275.

[5] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*. 1959, vol. 1, p. 269–271.

[6] Edwards, J., Daniel, E., Pascucci, V. and Bajaj, C. Approximating the Generalized Voronoi Diagram of Closely Spaced Objects. *Computer graphics forum : journal of the European Association for Computer Graphics*. may 2015, vol. 34, no. 2, p. 299–309.

[7] Goyal, A., Mogha, P., Luthra, R. and Sangwan, N. PATH FINDING: A* OR DIJKSTRA'S? *International Journal in IT & Engineering*. 2014, vol. 2, p. 1–15.

[8] Hart, P. E., Nilsson, N. J. and Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*. 1968, vol. 4, no. 2, p. 100–107. DOI: 10.1109/TSSC.1968.300136.

[9] Jakub, R. *Vizualizace algoritmů pro plánování cesty*. Brno, CZ, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis-file/18368/18368.pdf.

[10] Kavraki, L., Svestka, P., Latombe, J.-C. and Overmars, M. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*. 1996, vol. 12, no. 4, p. 566–580. DOI: 10.1109/70.508439.

[11] Latombe, J.-C. *Roadmap Methods*. Springer US, 1991. ISBN 978-1-4615-4022-9. Available at: https://doi.org/10.1007/978-1-4615-4022-9_4.

[12] Lozano Pérez, T. and Wesley, M. A. An Algorithm for Planning Collision-Free Paths among Polyhedral Obstacles. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. oct 1979, vol. 22, no. 10. DOI: 10.1145/359156.359164. ISSN 0001-0782. Available at: https://doi.org/10.1145/359156.359164.

[13] Petr, J. *Autonomous helicopter control by a mobile phone with android for precision agriculture*. Praha, CZ, 2018. Bakalářská práce. Czech Technical University in Prague. Available at: https://www.researchgate.net/publication/329372517_Autonomous_helicopter_control_by_a_mobile_phone_with_android_for_precision_agriculture.

[14] The Eclipse Foundation. *Eclipse Installer 2022-03 R* [Available at https://www.eclipse.org/downloads/packages/installer (2022-05-01)].

# Appendix A

# Storage medium contents

- **application** – folder for demo application
  - **PathPlanning.jar** - binary file for the application
  - **src** - folder containing source code of application
  - **examples** - folder containing pictures of the algorithms in editing mode and simulation mode
  - **vizlib** - folder containing the vizlib library necessary for the installation
- **text** – folder for documentation
  - **BP_xbreda00.pdf** – documentation file
  - **BP_xbreda00_print.pdf** – documentation file for print
  - **src** – folder containing source code for the documentation

# Appendix B

# Editing and running the application using Eclipse

To run the application, the current version of Java is required.

If it is required to edit or compile the source code, it can be done by using Eclipse, and making a new build. In this case, the installation of the *vizlib* library is also required, it can be found in the vizlib folder of the submitted CD. Its installation process is described in detail in the work of Jakub Rusnák [9]. The short version is as follows:

1. Installation of Eclipse. Installation instructions can be found for example here [14]
2. Installation of the *vizlib* library [9]
3. Import the .jar file found on the CD described in A
4. Run the program