ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

JOHN VON NEUMANN FACULTY OF INFORMATICS

THESIS

OE-NIK	Student’s name:	**Petrovikj, Kristina**
2023	Student’s registration number:	**T/007180/FI12904/N**

Óbuda University
John von Neumann Faculty of Informatics
Software Engineering Institute

THESIS WORK SPECIFICATION

Name of the student:	**Kristina Petrovikj**
Identification number of the student:	T/007180/FI12904/N
Neptun's code:	[redacted]

Title of the thesis:

Computational Geometry of Robot Motion

A robotmozgás geometriája

Internal supervisor	Dr. László Csink
External supervisor:	
Deadline:	15 December 2022
Subjects of the final exam:	Computer Architecture Cloud service technologies and IT security specialization

The task:

Robotics has become an essential interdiscplinary branch of computer science and engineering. Motion planning is an important research topic in robotics that has many applications.
The aim of this thesis work is to give a concise summary of the theoretical background of robot motion based on current literature concentrating on geometric algorithms.
In the framework of this project, a 2D visual robot simulation environment will be created demonstrating geometric algorithms for commonly occuring visuality problems.

The thesis must contain:

- review of current literature in geometric robot vision,
- analysis of visual robot simulation techniques,
- system design of a simple geometric robot motion planning,

Place of stamp

.....
Dr. Zoltán Vámossy
head of institute

This specification is valid until: **15 December 2024**
(According to OE TVSz 55.§)

I consider the thesis suitable for submission:

.....
external supervisor

.....
internal supervisor

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

STUDENT'S DECLARATION

I the undersigned student hereby declare that this thesis is the result of my own work; I have disclosed the references and tools used in an identifiable manner. The results indicated in my thesis completed may be used by the university and the institution announcing the task for their own purposes free of charge.

Dated: Budapest, December 9, 2022

K. Petrovic

student's signature

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

CONSULTATION LOG

Student's name: Neptun code: Specialty:

Kristina Petrovikj [redacted] Cloud service stechnologies and IT security

Phone number: Mailing address (e.g. domicile):

[redacted]

Degree project / thesis[1] title in Hungarian:

A robotmozgás geometriája:

Degree project / thesis[2] title in English:

Computational Geometry of Robot Motion

Institutional supervisor: External supervisor:

Dr. László Csink

Occ.	Date	Content	Signature
1.	11 FEB 2022	DISCUSSION ON TOPIC SELECTION	
2.	11 MARCH 2022	LITERATURE OVERVIEW	
3.	11 APRIL 2022	PROGRESS ISSUES	
4.	10 MAY 2022	SKETCH OF CONTENT	

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Degree project I / Degree project II (BSc) or Thesis 1 / Thesis 2 / Thesis 3 / Thesis 4[3], and is allowed to proceed for reporting / defense 4[4].

Dated: Budapest, 10 May 2022

.....

institutional supervisor

[1] underline as appropriate
[2] underline as appropriate
[3] underline as appropriate
[4] underline as appropriate

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

CONSULTATION LOG

Student's name:	Neptun code:	Specialty:
Kristina Petrovikj	[redacted]	Cloud service stechnologies and IT security

Phone number: Mailing address (e.g. domicile):

[redacted]

Degree project / thesis[5] title in Hungarian:

A robotmozgás geometriája:

Degree project / thesis[6] title in English:

Computational Geometry of Robot Motion

Institutional supervisor: External supervisor:

Dr. László Csink

Occ.	Date	Content	Signature
1.	12 SEP 2022	REVIEW AND ANALYSIS OF POSSIBLE APPROACHES	
2.	10 OCT 2022	SELECTION OF THE METHOD FOR SOLUTION	
3.	7 NOV 2022	IMPLEMENTATION AND TESTING	
4.	6 DEC 2022	ANALYSIS OF THE IMPLEMENTATION	

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Degree project I / Degree project II (BSc) or Thesis 1 / Thesis 2 / Thesis 3 / Thesis 4[7], and is allowed to proceed for reporting / defense 4[8].

Dated: Budapest, 6 December 2022

.....

institutional supervisor

[5] underline as appropriate
[6] underline as appropriate
[7] underline as appropriate
[8] underline as appropriate

CONTENT

1 INTRODUCTION

Robotics has become an essential interdisciplinary branch of Computer Science and Engineering. Robot Motion is an important research topic in Robotics that has many applications. It consists of many interconnected disciplines that altogether make the robot's motion possible. Path planning i.e., Navigation is one of them.

The significance of the robot's mobility is amply apparent. This work's major interest is the focus on the path-planning algorithms that are currently known. These algorithms are based on geometry and have a mathematical foundation. Simply put, it implies that they are provided in the form of an algorithm as a solution to a geometric issue in the plane or space. Naturally, it can also be provided in higher configuration dimensions, but I concentrate on 2D.

In this thesis work, I give a concise summary of the theoretical background of the robotic path planning algorithms based on current literature and a proposal for an optimized algorithm, which I name the Optimized Rapidly Exploring Random Tree Star algorithm, or short Optimized RRT*.

In the framework of this project, a 2D visual robot simulation environment consisting of common visuality problems is created using MATLAB. It demonstrates the performance of the novel Optimized RRT* algorithm. Alongside the implementation, a comparison with its predecessors is given, confirming the overall optimized performance of the proposed algorithm. Further on, I talk about the application of the proposed algorithm and its improvement potentials which are good material for further research.

2 COMPUTATIONAL GEOMETRY

Some geometric issues gave rise to the area of computational geometry in the 1970s. It may be characterized as the systematic study of geometric object algorithms and data structures, with an emphasis on asymptotically fast precise algorithms. The difficulties presented by the geometric issues attracted a lot of scholars. The path from the initial conceptualization of the problem to effective and beautiful solutions has frequently been long, with many challenging and subpar intermediate outcomes. A large variety of effective geometric algorithms available today are also reasonably simple to comprehend and use. Early algorithmic solutions to many geometric problems were either sluggish or challenging to comprehend and use. Numerous new algorithmic strategies have been created recently that have enhanced and sped up many of the older methods. Even though it is a new research field, its roots date back to ancient times. To achieve good solutions for geometric algorithms, one must pay attention to two elements. The first is a deep grasp of the problem's geometric aspects, and the second is the correct use of algorithmic approaches and data structures. So, comprehending the problem's geometry and knowing the correct algorithmic tools are of crucial importance [1].

2.1 Computational complexity

Computational complexity is fundamental to computational geometry and is one of the most crucial characteristics to pay attention to while examining the geometric algorithm, since it is one of the key objectives for optimizing it. Big O notation is a type of mathematical notation that expresses how a function limits itself when its argument goes towards a certain value or infinity. Big O notation categorizes functions based on their rates of growth, allowing for the representation of several functions with the same growth rate using the same notation. The reason the growth rate of a function is also known as the order of the function is why the letter O is used. Big O notation often only gives an upper bound on the function's growth rate when describing a function.

For instance, there might be days and seconds of calculation between O(n2) and O(nlogn). Based on this, the difference can be drastic.

In computer science, the act of figuring out an algorithm's computational complexity the amount of time, storage, or other resources needed to perform it is referred to as algorithm analysis. Typically, this includes creating a function that links the quantity of an algorithm's input to the number of steps it requires (time complexity) or the number of storage places it uses (its space complexity). When the values of a function are small or expand slowly in comparison to the amount of the input, the method is efficient.

2.2 Application domains

Computational geometry has many application domains. One of the most notable are the following:

- Robot Motion

  Robot Motion is one of the crucial aspects of the robotics field, that is responsible for the motion of the robot in a given environment. The robot should be able to autonomously move in the environment and make decisions on its own when the navigation is taken in account. In the next chapter Robot Motion is explained more in details.

- Geographic Information Systems

  Maps are a crucial tool for travelers since they provide a wealth of information. They provide information about the locations of tourist sites, including minor lakes, highways, and railway lines. They may be frustrating, too, as it can be challenging to obtain the proper information. For example, even when you know roughly where a tiny town is located on a map, it may still be challenging to find it. Geographic information systems overlay maps in order to make them easier to read [1].

- Computer Graphics

  The field of computer graphics focuses on using computers to create visuals. Modern digital photography, cinema, video games, cell phone and computer displays, and many specialized applications all use computer graphics as a key technology. There has been a lot of development in specialized hardware and software, and computer graphics technology now powers the majority of devices' screens. It is a modern and expansive field in computer science. Verne Hudson and William Fetter, two Boeing computer graphics experts, first used the expression in 1960. It is frequently referred to as "CG" or "computer-generated imagery" in the context of movies (CGI). Computer science research focuses on the technical elements of computer graphics [2].

- CAD/CAM

  The combination of computer-aided design (CAD) with computer-aided manufacturing (CAM) is known as CAD/CAM (CAM). Powerful computers are needed for each of these. Designers and draftsmen can benefit from CAD software, and CAM reduces personnel expenses in the production process [3].

3 ROBOT MOTION

Robot motion is the field of robotics responsible for navigating the robot from a departure to a destination point. The creation of autonomous robots, which are robots that can be instructed on what to do without being instructed on how to do it, is one of robotics' ultimate objectives. This necessitates a robot's ability to arrange its own movements among other things. A robot must have some understanding of the area it is going through to be able to plan a motion. A mobile robot moving around in a plant, for instance, must be aware of where impediments are. A floor plan can reveal some of this data, such as the locations of walls and equipment. The robot will have to rely on its sensors for more information. It ought to be able to recognize barriers that aren't shown on the floor layout, such as humans. In the process of successfully moving in an environment, the robotic system deals with four tasks: navigation, coverage, localization, and mapping. Robotics depends critically on finding a solution to this so-called motion planning problem. The design and analysis of motion planning algorithms pose a unique set of challenges in mechanics, control theory, computational and differential geometry, and computer science since actions in the real world are subject to physical laws, uncertainty, and geometric limitations [4].

- Navigation

  In the navigation task, the robot is in search of a collision-free path from one point to another point in the configuration space. Different environments mean different issues; thus, the need of efficient path planning algorithms arises. The environment can be static or dynamic. Static means that the environment does not change, whereas dynamic means that the environment can change. There can be obstacles in the given environment. The obstacles again, can be static or dynamic. A case can be that they can randomly appear and disappear. Due to these settings, we can have optimal and not optimal motions, i.e., algorithms.

- Coverage

  The issue of covering every point in a place with a sensor or instrument, as in demining or painting, is known as coverage. An algorithm is complete if it finds an existing path in a finite time, and in case the path does not exist then the algorithm returns that in a finite time.

- Localization

  The challenge of localization is to interpret sensor data for the robot's setup using a map. With the help of different sensors that are attached to the robot, together with an odometer that can be placed in the robot’s wheels and is responsible for measuring the distance passed so far, the robot will be able to localize itself and thus can successfully move around and do the intended tasks autonomously. The robot can have a map in advance or can be placed in an unknown environment

and it must build the map of the environment before it can localize itself.

- Mapping

The challenge of investigating and feeling an uncharted region to create a representation that may be used for localization, coverage, or navigation is known as mapping [1].
As in SLAM (simultaneous localization and mapping), localization and mapping may be combined, and this is useful in case of an unknown environment [4].

4 WORK SPACE AND CONFIGURATION SPACE

4.1 Work space

Let R be a point robot that moves around in a 2-dimensional environment, or as we call it – a workspace. The environment consists of a set S = {P_1,..., P_t} of obstacles. For the sake of simplicity let us assume that R is a simple polygon. With the help of a translation vector, we specify a placement i.e., the configuration of the polygon robot [1]. We denote the robot translated over a vector (x, y) by R (x, y). For example, if the robot is a polygon with the following vertices: (0,2), (-1,1), (1,1), (-1, -2), and (1, −2), then the vertices of R (4,2) will be (4,4), (3,3), (5,3), (3,0), and (5,0), respectively. In this way, a robot can be specified by giving the vertices of R (0,0). In this case, we translate over the vector (0,0) [1].

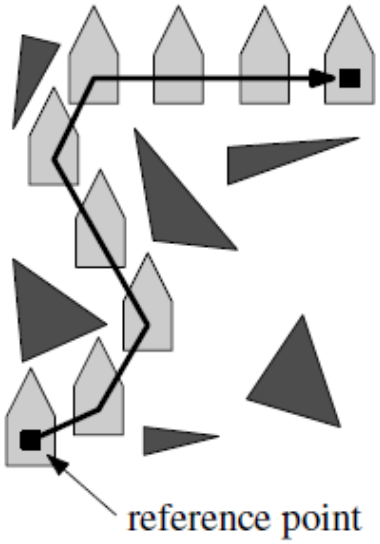

Figure 1
Work space [1, p. 286]

Consider the above from the standpoint of a reference point. The most understandable situation is when R's interior area contains the origin, which is denoted by (0, 0). In this case, we have that by a definition, this location is referred to as the robot's reference point. (Figure 2) [1]. If the robot is at the designated location, we may specify a placement for R by simply giving the reference point's coordinates. The following describes it better. R (x, y) specifies that the robot's reference point is at, therefore (x, y). The reference point might theoretically be a location outside the robot that is connected to it by an intangible stick instead of having to be within the robot itself. This location is the genesis of R (0,0) [1].

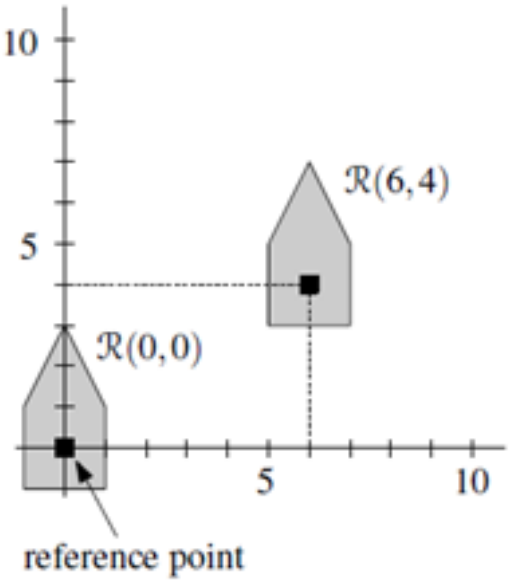

Figure 2
Reference point [1, p. 284]

Now, let us suppose that the robot can change its orientation by rotation, for example around its reference point (see Figure 3). In that case, we will need an extra parameter, φ, to specify the orientation of the robot. Let R (x, y, φ) represents the robot with its reference point at (x, y), which is rotated counterclockwise through an angle φ. So, what is specified at the start is R (0,0,0) [1].

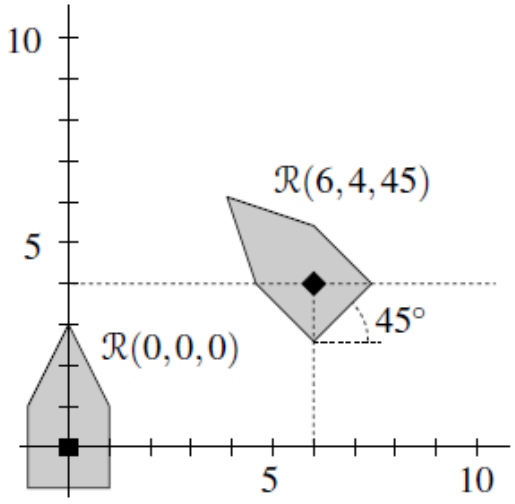

Figure 3
Changed orientation by rotation [1, p. 284]

4.2 Degrees of freedom (DOF)

In general, a robot's placement is decided by a set of variables whose total number is equal to the robot's DOF. This number is two for planar robots that can only translate, and three for those that can rotate as well as translate. We obviously require more parameters for a robot in three dimensions: a translational robot in $\boldsymbol{R^3}$ has three degrees of freedom, and a robot with six degrees of freedom that can freely move and rotate in $\boldsymbol{R^3}$ [1].

4.3 Configuration space

The parameter space of a robot R is usually called its configuration space [1]. It is denoted by C(R). A particular placement R(p) of the robot in the workspace matches a point in the configuration space. When we do a translation and rotation of the robot in the plane, the configuration space is 3-dimensional [1].

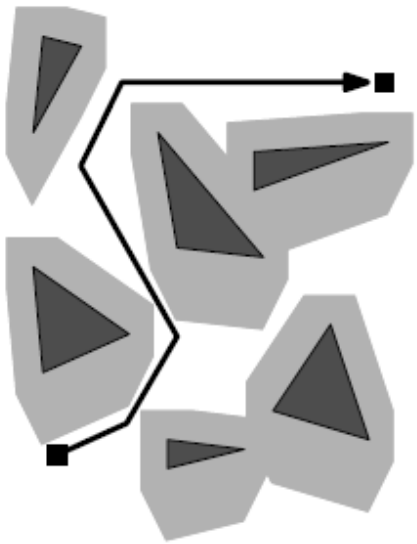

Figure 4
Configuration space [1, p. 286]

4.4 Free and forbidden configuration space

Every point in the configuration space corresponds to a particular location of a real robot in the workplace, and each point represents a polygonal robot in the workspace [1]. One may characterize the robot's location by supplying values for the variables that have an influence on it or by specifying a point in the configuration space. Of course, not every scenario can be fulfilled. The spots in S where the robot passes over a barrier are not allowed and cannot be accepted [1]. The part of the configuration space containing these points is called a forbidden configuration space [1]. It goes by C_{forb} (R, S). The rest of the configuration space, the placements where the robot does not intersect any obstacle, is called the free configuration space, and it goes by C_{free} (R, S) [1]. Let us restrict the environment to a large box denoted by B. The free configuration space C_{free} now is represented by the part of *B* that does not have any obstacle:

$C_{free} = B \setminus \bigcup_{i=1}^{t} \boldsymbol{P_i}$ [1].

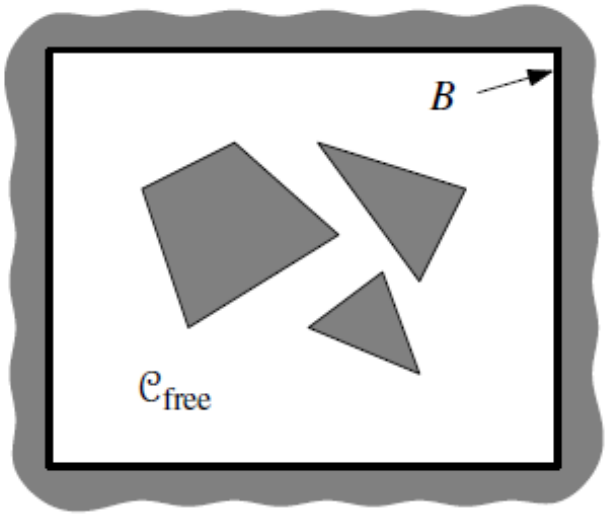

Figure 5
Free and forbidden CS [1, p. 287]

5 KNOWN ALGORITHMS

The existing navigation planners consisting of many algorithms are a great base to do some research on and give thought to improvement and use cases. What we already have discovered, has led us to success thus it is a promising step to continue from there.

Having the mindset, that everything can be improved, and nothing has a limit, I am diving into the research of what already exists, with the hope that some parameters can be improved and changed, leading to an overall optimization.

I researched three groups of path-planning algorithms: Bug algorithms, grid-based and sampling-based algorithms.

5.1 Bug algorithms

The Bug algorithms are sensor-based algorithms. This means, that when implemented, there are sensors attached to the robot that collect data and the algorithm uses this data for accurate implementation. They are one of the simplest planners but nevertheless have interesting and challenging issues. The Bug1 [4] and Bug2 algorithms [5] are the first versions of the bug family. They assume that the robot is a point robot, and it moves through the environment by detecting obstacles with a contact sensor or a zero-range sensor. If the robot has the nonzero sensor property, then an updated Bug algorithm called the Tangent Bug [6] is introduced. It uses the sensor information to find a shorter path to the destination. The Bug Algorithms work in the following manner: they move straightforwardly on a (straight) line and follow a boundary (wall).

5.1.1 Bug1

The Bug1 algorithm exhibits two behaviors: motion-to-goal and boundary-following [4]. The robot is represented by a point, and it circumnavigates the obstacles until motion-to-goal is allowed once again. At all times it moves on a straight line and follows the obstacles' boundary.

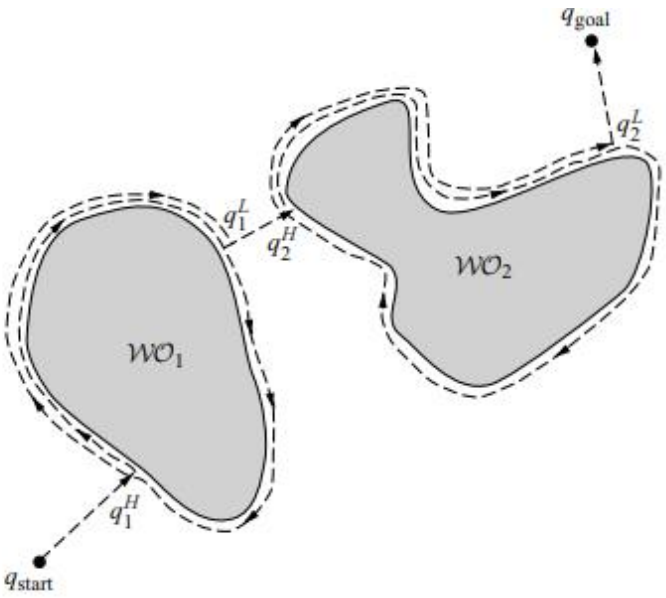

Figure 6
The Bug1 algorithm reaches the goal [4]

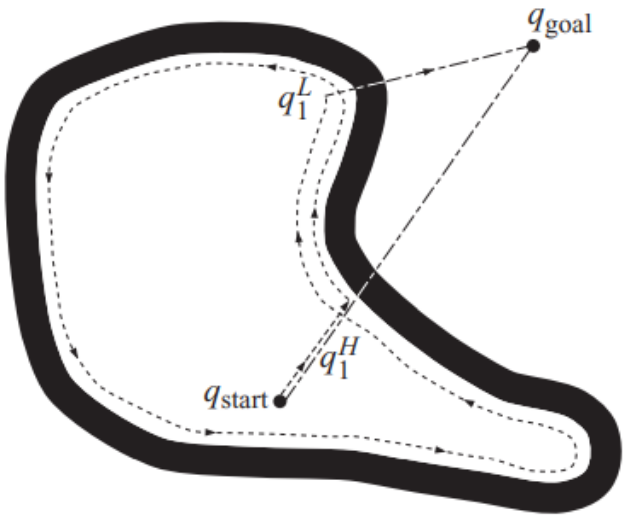

Figure 7
The target is unreachable [4]

Input: A point robot with a tactile sensor
Output: A path to the q_{goal} or a conclusion no such path exists

while Forever **do**
  repeat
    From q_{i-1}^{L}, move toward q_{goal}.
  until q_{goal} is reached **or** an obstacle is encountered at q_i^H.
  if Goal is reached **then**
    Exit.
  end if
  repeat
    Follow the obstacle boundary.
  until q_{goal} is reached **or** q_i^H is re-encountered.
  Determine the point q_i^L on the perimeter that has the shortest distance to the goal.
  Go to q_i^L.
  if the robot were to move toward the goal **then**
    Conclude q_{goal} is not reachable and exit.
  end if
end while

Table 1
Bug1 algorithm [4]

5.1.2 Bug2

The Bug2 [5] algorithm is very similar to Bug1. As well, it exhibits just two behaviors, the motion-to-goal one and the boundary-following one. During the first behavior, when it moves towards the goal it moves on the m-line, but in Bug2 the m-line connects the starting and the end points, q_{start} and q_{goal} respectively. In this way, it remains fixed. The second behavior, i.e., is triggered only when the robot encounters an obstacle. But again, it differs here in the case of Bug2 compared to Bug1. In Bug2, the point robot circles the obstruction until it arrives at a new position on the m-line that is nearer the objective than the point of first contact with the wall of the obstruction. [7] Afterward, it moves on toward the goal, and if meets an obstacle again, it repeats again the process. If the point it meets the obstacle at, is the same point as before, then in this case there is no path to the goal.

After this comparison, it seems that Bug2 is a more effective algorithm than Bug1 because the robot does not have to entirely circumnavigate the obstacles.

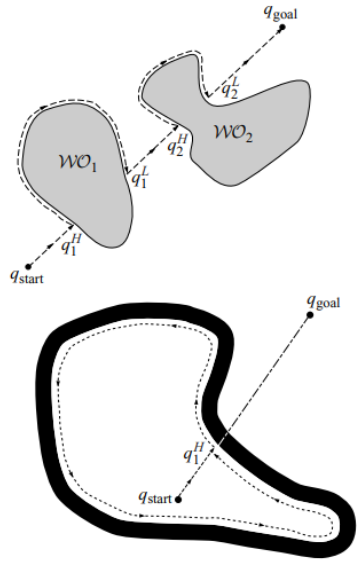

Figure 8
(Top) The Bug2 algorithm finds a path to the goal. (Bottom) The Bug2 algorithm reports failure. [4]

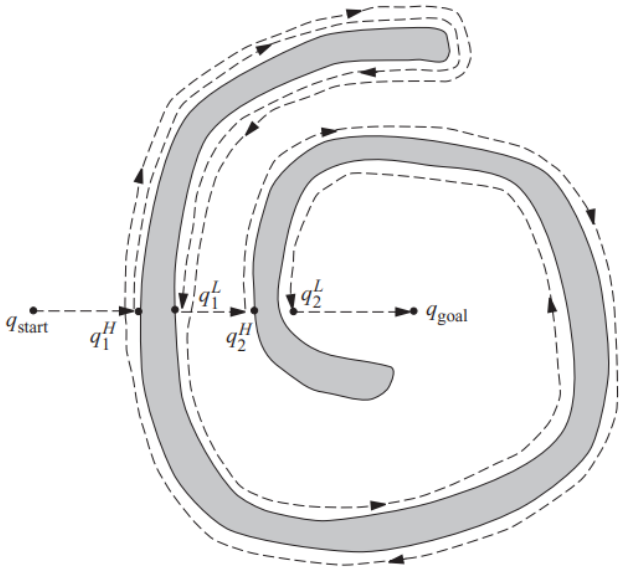

Figure 9
Bug2 algorithm [4]

Input: A point robot with a tactile sensor
Output: A path to q_{goal} or a conclusion no such path exists

while True **do**
  repeat
    From q_{i-1}^{L}, move toward q_{goal} along m-line.
  until
  q_{goal} is reached **or**
  an obstacle is encountered at *hit point* q_i^H.
  Turn left (or right).
  repeat
    Follow boundary
  until
   q_{goal} is reached **or**
   q_i^H is re-encountered **or**
   m-line is re-encountered at a point m such that
    $m \neq q_i^H$ (robot did not reach the hit point),
    $d(m, q_{\text{goal}}) < d(m, q_i^H)$ (robot is closer), and
    If robot moves toward goal, it would not hit the obstacle
  Let $q_{i+1}^{L} = m$
  Increment i
end while

Table 2
Bug2 algorithm [4]

5.1.3 Tangent Bug

Tangent Bug is an upgrade on Bug2 in that it uses a range sensor with 360-degree infinite orientation resolution to find a quicker route to the target [4], [6]. Below is the pseudocode explained, followed by some illustrative examples. WOi in Figure 10, is an abbreviation of the obstacles' workspace.

Input: A point robot with a range sensor
Output: A path to the q_{goal} or a conclusion no such path exists

1: **while** True **do**
2: **repeat**
3: Continuously move toward the point $n \in \{T, O_i\}$ which minimizes $d(x, n) + d(n, q_{goal})$
4: **until**

- the goal is encountered **or**
- The direction that minimizes $d(x, n) + d(n, q_{goal})$ begins to increase $d(x, q_{goal})$, i.e., the robot detects a "local minimum" of $d(\cdot, q_{goal})$.

5: Chose a boundary following direction which continues in the same direction as the most recent motion-to-goal direction.
6: **repeat**
7: Continuously update d_{reach}, $d_{followed}$, and $\{O_i\}$.
8: Continuously moves toward $n \in \{O_i\}$ that is in the chosen boundary direction.
9: **until**

- The goal is reached.
- The robot completes a cycle around the obstacle in which case the goal cannot be achieved.
- $d_{reach} < d_{followed}$

10: **end while**

Table 3
Tangent Bug [4]

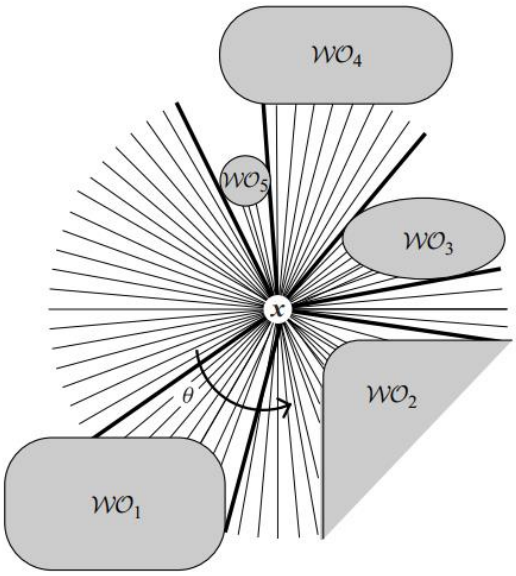

Figure 10
Rays [4]

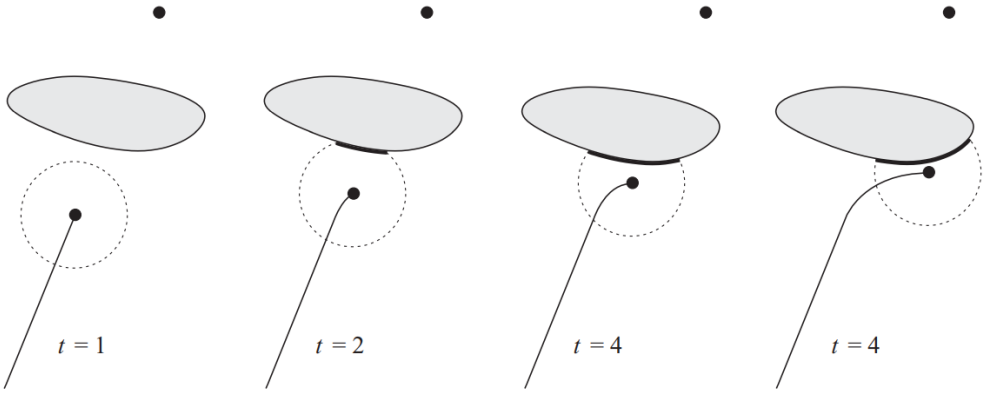

Figure 11
Motion-to-goal behavior for a robot with a finite sensor range [4]

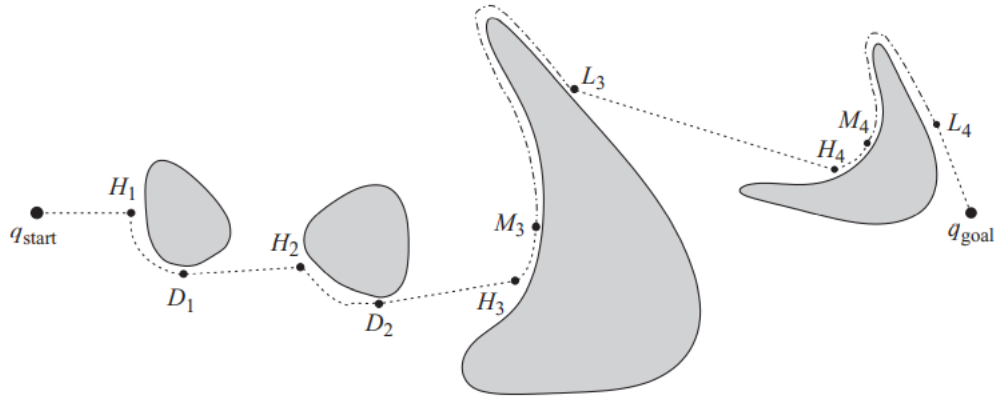

Figure 12
The path generated by Tangent Bug with zero sensor range [4]

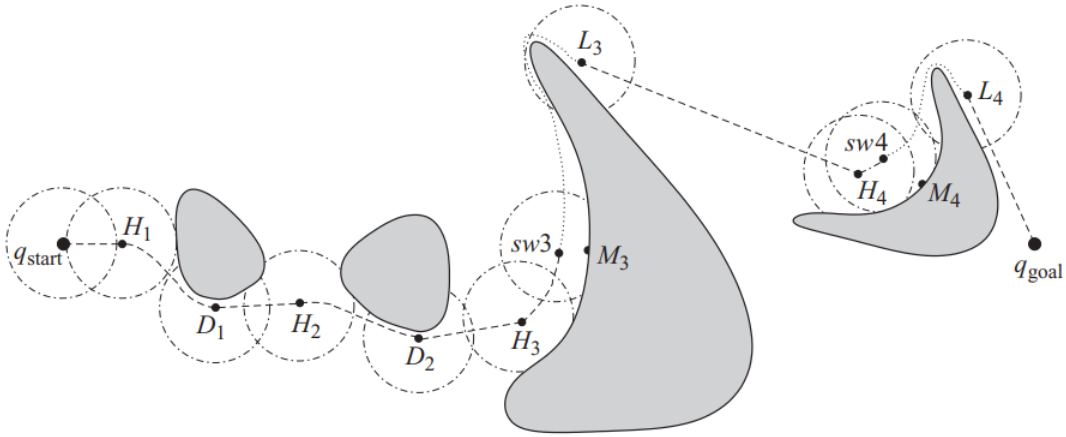

Figure 13
Path generated by Tangent Bug with finite sensor range [4]

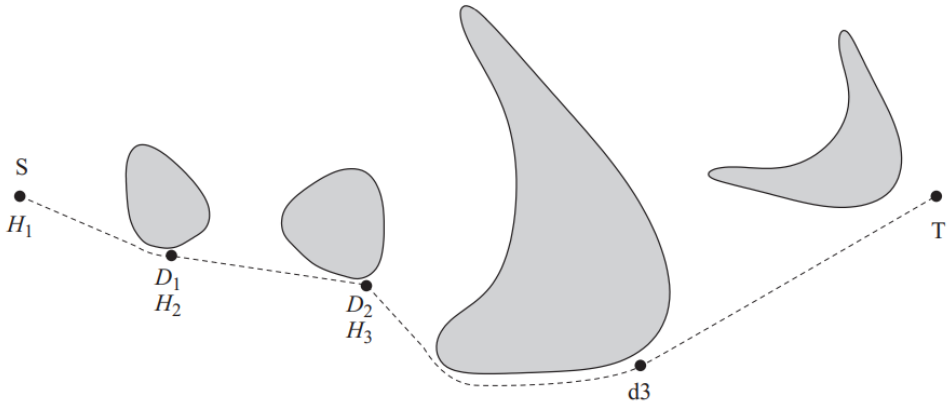

Figure 14
Path generated by Tangent Bug with infinite sensor range [4]

5.2 Dijkstra's algorithm

Dijkstra's algorithm is used for situations where we need to find the single-source shortest paths on a weighted and directed graph, given as G = (V, E), where V is the set of vertices and E is the set of edges, where all the edge weights are not negative.

The algorithm is based on a set S that contains the vertices whose final shortest-path weights from the source s have already been found out. It constantly selects the vertex u ∈ V – S with the minimum shortest-path estimate, it adds u to S and relaxes all edges leaving u. The following implementation is based on a min-priority queue Q of vertices, where the key is their d value [8].

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5       do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for each vertex v ∈ Adj[u]
8               do RELAX(u, v, w)
```

Table 4
Dijkstra's algorithm [8]

The algorithm modifies edges as shown in Figure 15. In Line 1 we have the initialization of d and π values. Line 2 initializes the set S to be empty. At all times when it is the start of the while loop, it maintains the equation Q=V-S from lines 4 to 8. In Line 3 we have an initialization of the minimum priority queue Q, so it contains all the vertices of the graph. Note, that because at the beginning S is empty, the equation is true only after the third line. At every iteration of the while loop of lines 4–8, a vertex u is taken from Q = V − S and added to the set S, so we have satisfied the equation. During the first iteration, u=s. Lines 7 and 8 modify each edge u-v (leaving u), so it updates the estimate d[v] and the predecessor π[v] if the shortest path to v can be somehow improved by passing through u. The algorithm always chooses the closest vertex in V-S to add to the set S. Dijkstra's algorithm computes shortest paths. The important aspect to be shown is that each time a vertex u is added to set S, we have d[u] = δ (s, u). The implementation of the min-priority queue affects how long Dijkstra's algorithm takes to complete.

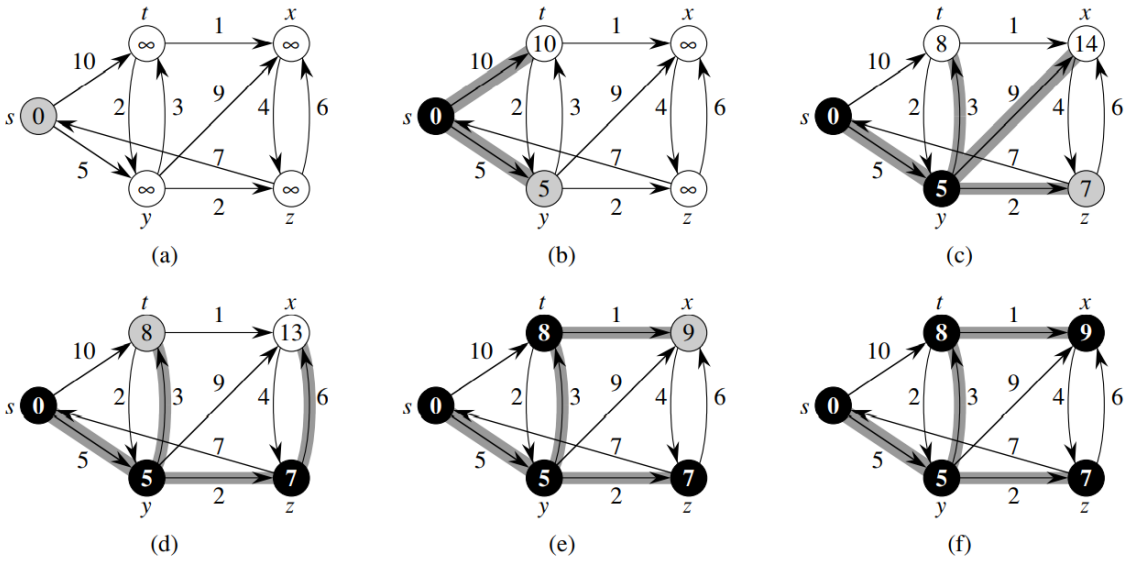

Figure 15
The execution of Dijkstra's algorithm [8]

5.3 A*

It might happen that the metric we want to optimize i.e., improve is not the shortest-path length. Other metrics can be time, safety, energy, or maybe a combination of all of them.

Looking at graph-based search algorithms, one of the objectives is to optimize the number of nodes that are visited. Here heuristics play an important role. They can lead to the optimal path or maybe close to it.

An example can be, choosing the next node such that it has the shortest Euclidean distance to the goal. This can be preferred simply because the such node has the highest possibility of getting closer to the destination. This is just an assumption, a good guess and there is not any guarantee this will work. We assume this kind of heuristics is based on local information [4].

The A* algorithm is again a graph-based algorithm. We can simply say that the A* is an optimized Dijkstra algorithm. The difference is that in the case of A* we have a chosen heuristic, so the graph search is optimized. The algorithm returns an optimal path only if the heuristic is good. For example, if the environment is a grid, then the graph represents the grid, and a good heuristic could be the Euclidean distance to the destination [4].

The following A* algorithm uses the priority queue data structure.

The priority queue simply contains the nodes sorted by their priority, which is determined by the sum of the distance traveled in the graph thus far from the start node together with the heuristic.

Input: A graph
Output: A path between start and goal nodes

1: **repeat**
2: Pick n_{best} from O such that $f(n_{best}) \leq f(n), \forall n \in O$.
3: Remove n_{best} from O and add to C.
4: If $n_{best} = q_{goal}$, EXIT.
5: Expand n_{best}: for all $x \in \text{Star}(n_{best})$ that are not in C.
6: **if** $x \notin O$ **then**
7: add x to O.
8: **else if** $g(n_{best}) + c(n_{best}, x) < g(x)$ **then**
9: update x's backpointer to point to n_{best}
10: **end if**
11: **until** O is empty

Table 5
The execution of A algorithm [8]*

A* ’s algorithm input is a graph. The nodes can be embedded into the free space of the robot so they can be seen as coordinates. Edges correspond to adjacent vertices and have values corresponding to the cost required to traverse between the adjacent vertices.

A*’s algorithm output is a back-pointer path, which is a sequence of nodes starting from the goal and going back to the start position. Let’s denote 2 additional data structures, an open set O and a closed set C, correspondingly. Set O will be a priority queue data structure and the closed set C will contain all the previously processed vertices.

Star(n) is the set of nodes that are neighbors to n. c (n1, n2) is the length of edge connecting n1 and n2. g(n) is the total length of a back pointer path from n to $q_{start.}$

h(n) is the heuristic cost function, which returns the approximated cost of the shortest path from n to q_{goal} . Then the function [f (n) = g(n) +h(n)] is the estimated cost of the shortest path from q_{start} to q_{goal} through n.

5.4 D*

The D* algorithm [9] is a version of the A* algorithm, just in a dynamic environment. means that the edge cost parameters can change during the motion.

So far, the discussed algorithms were good examples for static environments, but in real-life situations, most of the time, the robot is dealing with a changing, dynamic environment [4].

D∗ runs a modified Dijkstra algorithm backward. The modification involves updating a heuristic and a minimum heuristic function.

5.5 Grassfire algorithm

The grassfire algorithm is a graph-based algorithm. More specifically, it is used for path planning in a grid-like environment. A grid is a lattice graph.

The algorithm uses a breadth-first approach. It is complete, so it returns a path if there is one, otherwise, it states that a path does not exist. The computational complexity increases linearly, i.e., O(V) where V is the number of nodes in the graph [10].

Step 1: Start
Step 2: For node =1 to *n* in the configuration field
    2.1: The distance of the node is initialized to infinity
Step 3: Create an empty list
Step 4: Initialize the distance of the goal node to zero
    4.1: add goal to list
Step 5: while list not empty
    5.1: Let current=first node in list
    5.2: Remove current from list
    5.3: For each node *n* that is adjacent to current
        5.3.1: if *n*.distance=infinity
            5.3.1.1: *n*.distance=current.distance+1
            5.3.1.2: add *n* to the back of the list
Step 6: Stop

Table 6
Grassfire algorithm

To move towards the destination from any node simply move towards the neighbor with the smallest distance value, breaking ties arbitrarily.

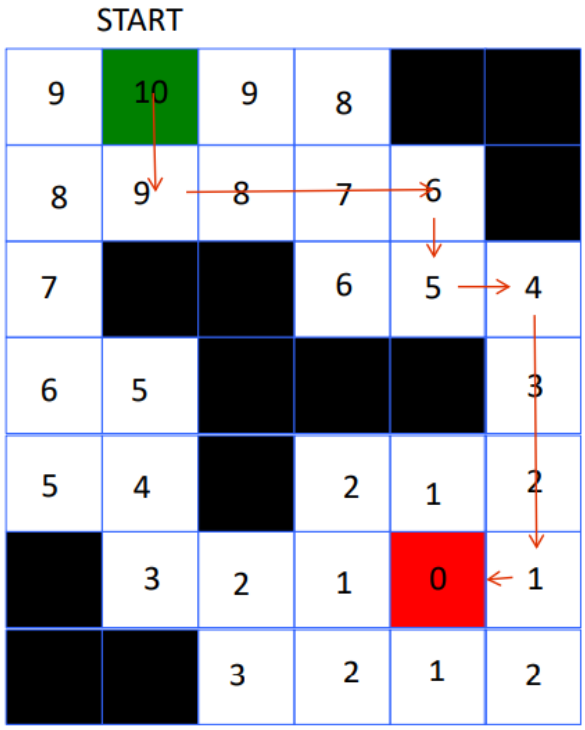

Figure 16 Path exists [10]

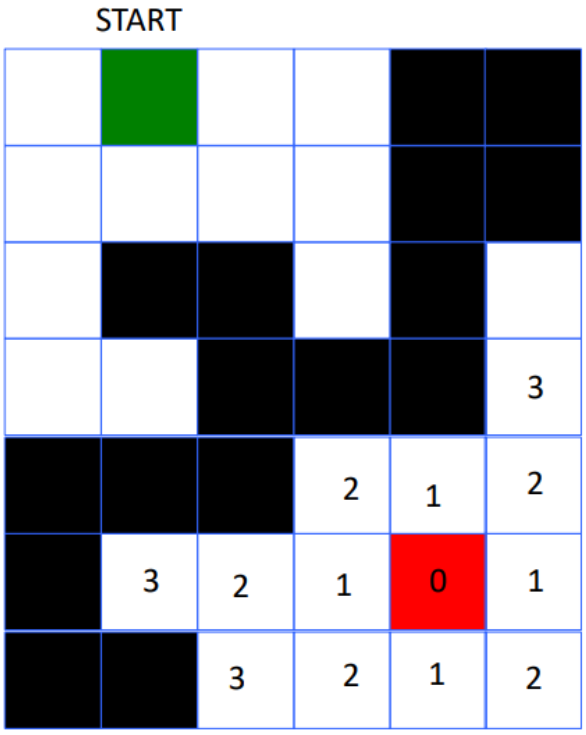

Figure 17 Path does not exist [10]

5.6 Probabalistic Roadmap (PRM)

To find solutions to path-planning issues, sampling-based approaches use several techniques for creating samples (collision-free configurations of the robot) and linking the samples with routes.

Random or probability sampling is the easiest of all sampling methods to apply to C-spaces.

The probabilistic roadmap (PRM) [11] is a sampling-based planner, that operates in two phases. The first one is a learning phase in which the roadmap is constructed by storing the collision-free configurations as nodes and the possible connections between them as edges in a graph. The second phase is when the goal and destination are substituted as the corresponding nodes in the graph and a path between them is being searched. The PRM can be used to solve high-dimensional path planning problems (5–12 degrees of freedom).

5.7 Rapidly exploring random tree (RRT)

The idea behind this algorithm is to construct a search tree incrementally so that when the number of nodes approaches infinity, it densely covers the space. For this approach, a dense sequence of random samples is used for the construction of the search tree, thus a dense covering of the space is obtained [12], [13].

Points are created at random and connected to the nearest node that is accessible. A verification that the vertex resides outside of an obstruction must be done each time a vertex is generated. Chaining the vertex to its nearest neighbor must also avoid impediments, as well. When a node is produced inside the desired region or a limit is reached, the algorithm is finished [13].

The way the randomly produced vertex is chained can also be changed. Calculating the vector that creates the shortest path between the new vertex and the nearest edge is one approach. A new node is linked to the randomly generated vertex and added to the edge at the junction. As an alternative, a chain of discretized nodes can be connected from the vertex to the nearest node. This approach requires more points to be saved but takes less calculation and is easier to implement [13].

RRT generates very cubic graphs. Given that nodes are connected to their closest neighbors, this is predicted. These graphs' structural characteristics reduce the likelihood of discovering an ideal route. The two triangular legs are crossed rather than taking the hypotenuse between the two points. Clearly, this is a greater distance. RRT* addresses the uneven pathways and cubic nature produced by RRT [13].

The algorithm's speed and application are advantages. RRT is rapid when compared to other path-planning methods. Finding its nearest neighbor is the algorithm's most expensive step since the length of this procedure increases with the number of created vertices [13].

The RRT may reach every point in the space arbitrarily since it is dense in the limit (with probability one). RRTs were introduced as a single-query planning algorithm that efficiently covers the space between q_{init} and q_{goal} [4].

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$;

Table 7 RRT

5.8 RRT*

RRT may be further refined so that, after discovering a first path, the new RRT* algorithm keeps looking for a more ideal one. This indicates that the method will yield the shortest, most ideal path if the tree has an infinite number of vertices. However, this is obviously not feasible. In any case, it suggests that after some time, it returns a more ideal path than the initial one discovered. The only difference between RRT* and RRT is the two increases to RRT* [13]. First, RRT* records the separation between each vertex and its parent vertex. This is referred to as the cost () of the specific vertex. A set of vertices within a certain radius of the newly discovered node in the graph are then investigated. If a node is located that has a lower cost () than the proximal node, the cheaper node will take its place. The addition of fan-shaped twigs to the tree structure demonstrates the influence of this feature. The RRT cubic structure is removed [13].

The second is the actual rewiring of the tree. A vertex is linked to its cheapest neighbor after which the neighbors are once again looked at. It is determined whether rewiring neighbors to the newly added vertex would reduce their cost. If the cost does decrease, the neighbor is rewired to the newly added vertex. The road is smoother thanks to this feature. Paths made by RRT* are exceedingly straight [13]. Additionally, its graphs differ significantly from RRT's in several key ways. The structure of RRT* is extremely helpful for finding an ideal path, especially in a crowded field of obstacles. Compared to RRT, the graph finds shorter pathways by wrapping itself around objects. The first graph may still be used even if the destination changes because it shows the most direct route to most places in the region [13].

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
        $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
      if $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$ **then** $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;

*Table 8 RRT**

6 OPTIMIZED RRT*

RRTs, or rapidly exploring random trees, are used often in motion planning because they are effective at solving single-query issues. RRTs are extended to the challenge of finding the best solution by optimum RRTs (RRT*s), which asymptotically determine the best route from the starting state to each state in the planning domain. This behavior is not only ineffective but also at odds with the single-query nature of their architecture [14]. The subset of states that potentially enhance a solution can be characterized by a prolate hyper-spheroid for issues that aim to decrease path length. Direct sampling of this subgroup helps us narrow the search. One way to optimize RRT* is to sample the states that lead to the optimal solution directly. In this way, the new, improved algorithm shows the same convergence and solution quality as RRT*, but it is more optimized, thus it returns a more optimal, i.e., shorter path in less time. This Optimized RRT* can be used in future developments for more advanced motion planning algorithms. The big advantage of this new algorithm is that it does not require additional parameters to be handled [14].

This enables them to directly consider kinodynamic limitations and scale with issue size more efficiently; nevertheless, the end consequence is a less-strict completeness guarantee. RRTs are probabilistically complete, which ensures that as the number of iterations approaches infinity, the likelihood of discovering a solution, [14] if one exists, approaches unity. Prior efforts to concentrate on RRT and RRT* have focused on repeated searches, sample biasing, heuristic-based graph trimming, and heuristic-based sample rejection [14].

Table 9 shows the pseudocode of the Optimized RRT*. The lines in red are the additional lines that make the Optimized RRT* different from the RRT*.

Optimized RRT* reduces the planning issue to subsets of the original domain using heuristics. Because it cannot focus the search when the related prolate hyper-spheroid is greater than the planning issue, it is intrinsically dependent on the cost of the existing solution [14].

Algorithm: Optimized RRT* (X_{START}, X_{GOAL})

```
V ← {x_start};
E ← ∅;
X_soln ← ∅;
T = (V, E);
for iteration = 1 ... N do
    c_best ← min_{x_soln ∈ X_soln} {Cost(x_soln)};
    x_rand ← Sample(x_start, x_goal, c_best);
    x_nearest ← Nearest(T, x_rand);
    x_new ← Steer(x_nearest, x_rand);
    if CollisionFree(x_nearest, x_new) then
        V ← ∪{x_new};
        X_near ← Near(T, x_new, r_RRT*);
        x_min ← x_nearest;
        c_min ← Cost(x_min) + c · Line(x_nearest, x_new);
        for ∀x_near ∈ X_near do
            c_new ← Cost(x_near) + c · Line(x_near, x_new);
            if c_new < c_min then
                if CollisionFree(x_near, x_new) then
                    x_min ← x_near;
                    c_min ← c_new;
        E ← E ∪ {(x_min, x_new)};
        for ∀x_near ∈ X_near do
            c_near ← Cost(x_near);
            c_new ← Cost(x_new) + c · Line(x_new, x_near);
            if c_new < c_near then
                if CollisionFree(x_new, x_near) then
                    x_parent ← Parent(x_near);
                    E ← E \ {(x_parent, x_near)};
                    E ← E ∪ {(x_new, x_near)};
        if InGoalRegion(x_new) then
            X_soln ← X_soln ∪ {x_new};
return T;
```

Table 9
*Optimized RRT**

The SAMPLE algorithm is the direct sampling from the solution states [14].

1 **if** $c_{\max} < \infty$ **then**
2 $\quad c_{\min} \leftarrow \left|\left|\mathbf{x}_{\text{goal}} - \mathbf{x}_{\text{start}}\right|\right|_2$;
3 $\quad \mathbf{x}_{\text{centre}} \leftarrow \left(\mathbf{x}_{\text{start}} + \mathbf{x}_{\text{goal}}\right)/2$;
4 $\quad \mathbf{C} \leftarrow \texttt{RotationToWorldFrame}\left(\mathbf{x}_{\text{start}}, \mathbf{x}_{\text{goal}}\right)$;
5 $\quad r_1 \leftarrow c_{\max}/2$;
6 $\quad \{r_i\}_{i=2,\ldots,n} \leftarrow \left(\sqrt{c_{\max}^2 - c_{\min}^2}\right)/2$;
7 $\quad \mathbf{L} \leftarrow \text{diag}\left\{r_1, r_2, \ldots, r_n\right\}$;
8 $\quad \mathbf{x}_{\text{ball}} \leftarrow \texttt{SampleUnitNBall}$;
9 $\quad \mathbf{x}_{\text{rand}} \leftarrow \left(\mathbf{C}\mathbf{L}\mathbf{x}_{\text{ball}} + \mathbf{x}_{\text{centre}}\right) \cap X$;
10 **else**
11 $\quad \mathbf{x}_{\text{rand}} \sim \mathcal{U}\left(X\right)$;
12 **return** $\mathbf{x}_{\text{rand}}$;

Table 10
Sample algorithm

7 IMPLEMENTATION IN MATLAB

I chose MATLAB as a software for the implementation of the algorithms due to its visual characteristics and advantages. It gives a clear simulation result that is understandable and clear for further analysis.

I implemented RRT, RRT*, and Optimized RRT*.

The simulation results are the following.

7.1 Implementation of RRT algorithm

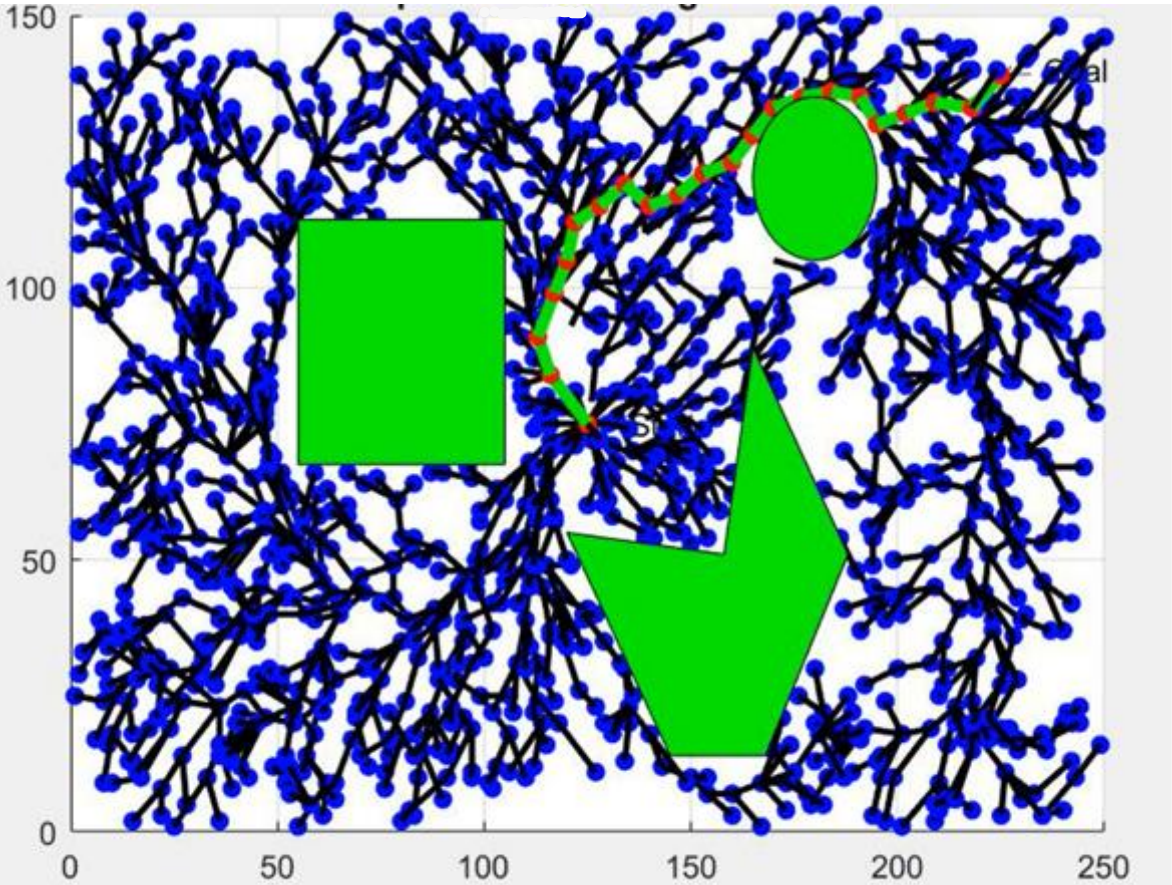

Figure 18
The output of my implementation of RRT

This is the original Rapidly Exploring Random Tree algorithm, that randomly grows a tree-like structure and then connects a path from a start position to a destination position.

The total time for finding a path is around 29.3 seconds.

7.2 Implementation of RRT* algorithm

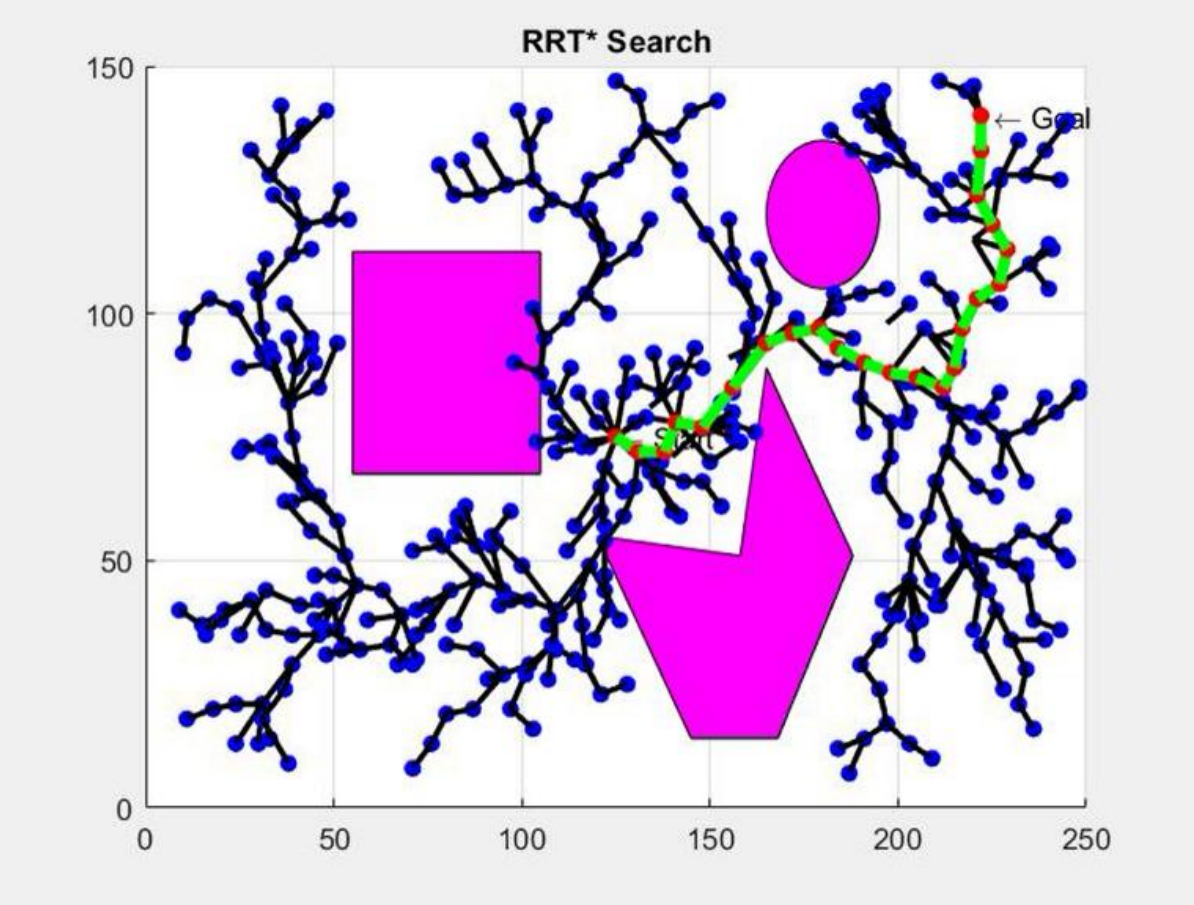

Figure 19
*The output of my implementation of RRT**

This is the implementation result of the RRT* algorithm. We can see that it updates the optimal path when it finds a better one. This is a computationally expensive process. Thus, the following Optimized RRT* plays an important role in the family of sampling-based algorithms.

The total time for finding a first path is 13.5 seconds. But the process continues, and it finds a better path after 58 seconds.

7.3 Implementation of Optimized RRT* algorithm

Here is my implementation result of the novel Optimized RRT* algorithm.

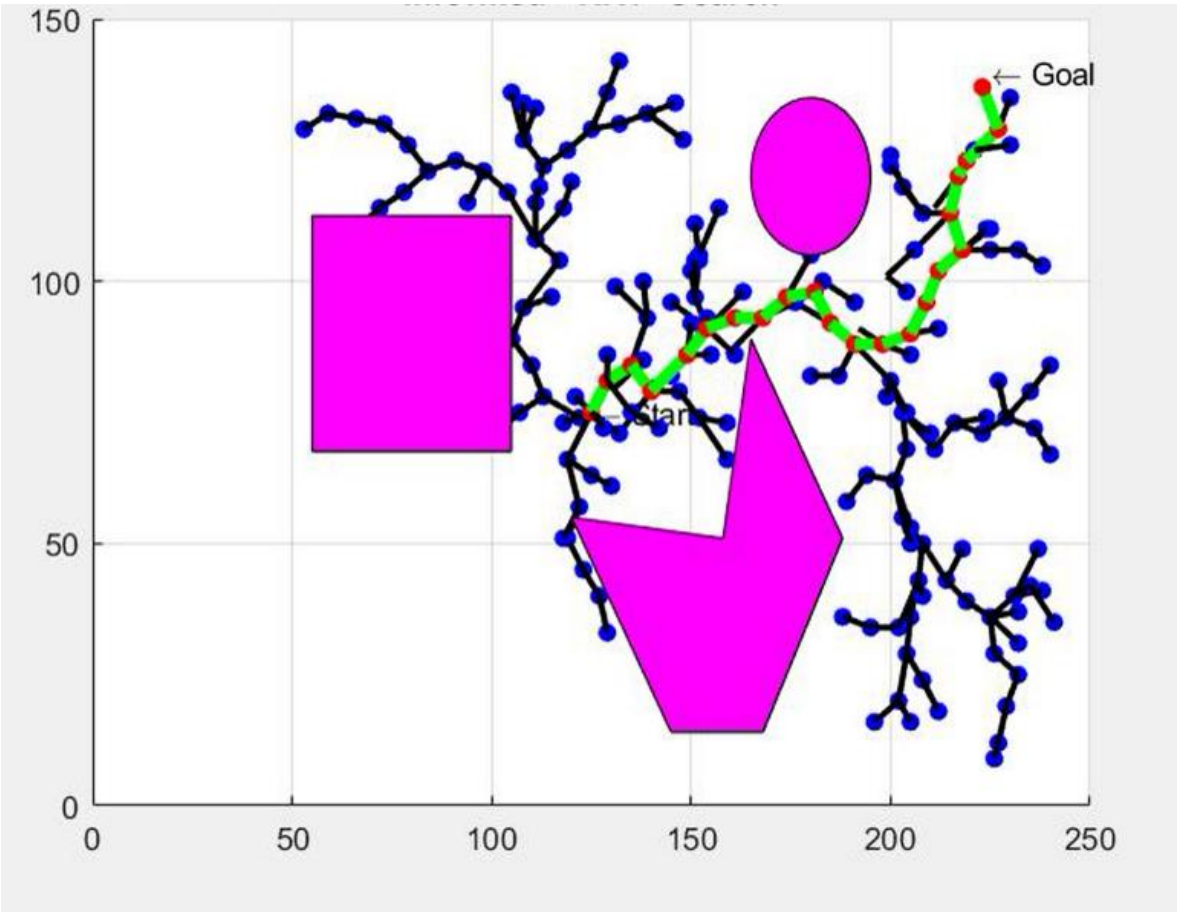

Figure 20
*The output of my implementation of Optimized RRT**

This optimized version is better than the original RRT* algorithm, as now it only searches a new, better path from a direct sampling from the states. Thus, it is more computationally cheap, and it finds an optimal path faster.

The total time for finding a path is around 6.5 seconds. The time for finding a more optimal path is 32 seconds. It has a better performance compared to the RRT*.

The Optimized RRT* behaves exactly like RRT* until the first solution is found by the algorithm. Afterward, it only samples directly from the subset of states led by a heuristic to lead to an improved solution.

8 TESTING AND COMPARISON

I tested RRT* and Optimized RRT* for the same environment, and I got the following results:

- Optimized RRT* performs better for both, finding the first path from the departure to the destination point, and the optimal solution. In Figure 21 it is shown that RRT* has nearly 60% successful performance, whereas Optimized RRT* for the first solution has nearly 90%.
- Optimized RRT* has almost 90% successful performance for finding the best solution, compared to the 51% of the one RRT*. If given more time, the performance of Optimized RRT* will tend to reach 100% faster than RRT*'s.

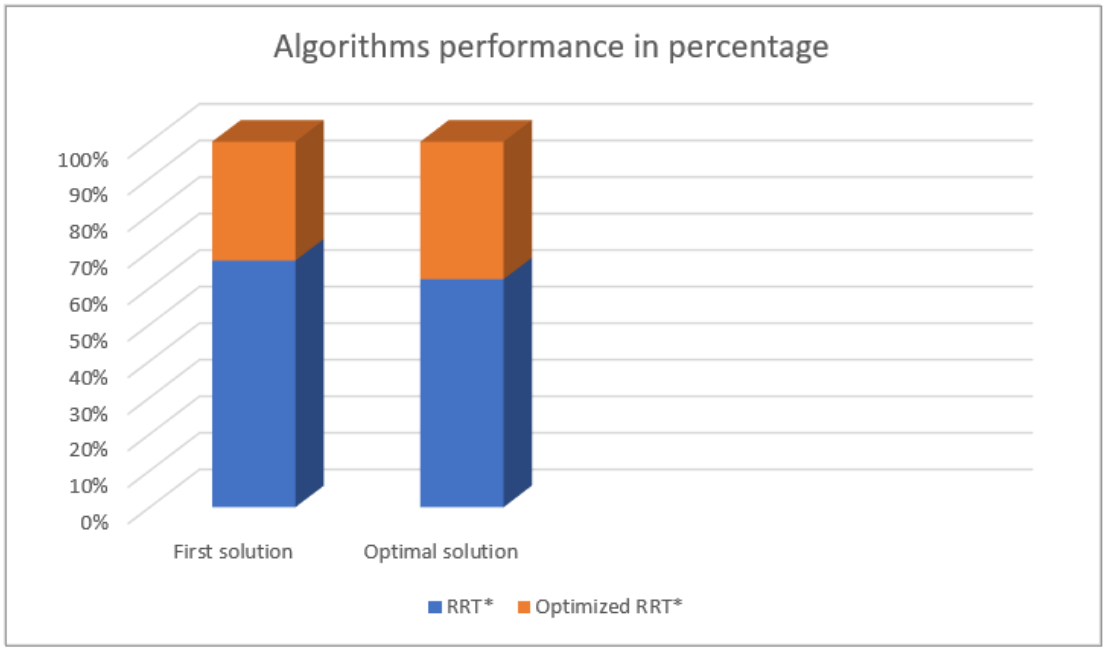

Figure 21
Algorithms performance in percentage

- Optimized RRT* finds the first solution in less than half the time needed for RRT* to find the first solution.
- Optimized RRT* needs a bit more than half the time of RRT* to find a better solution than the first one.

These results confirm that Optimized RRT* is more optimal than RRT*, as it returns a solution much faster. Of course, accuracy should be tested and measured separately. These results are based on time (in seconds).

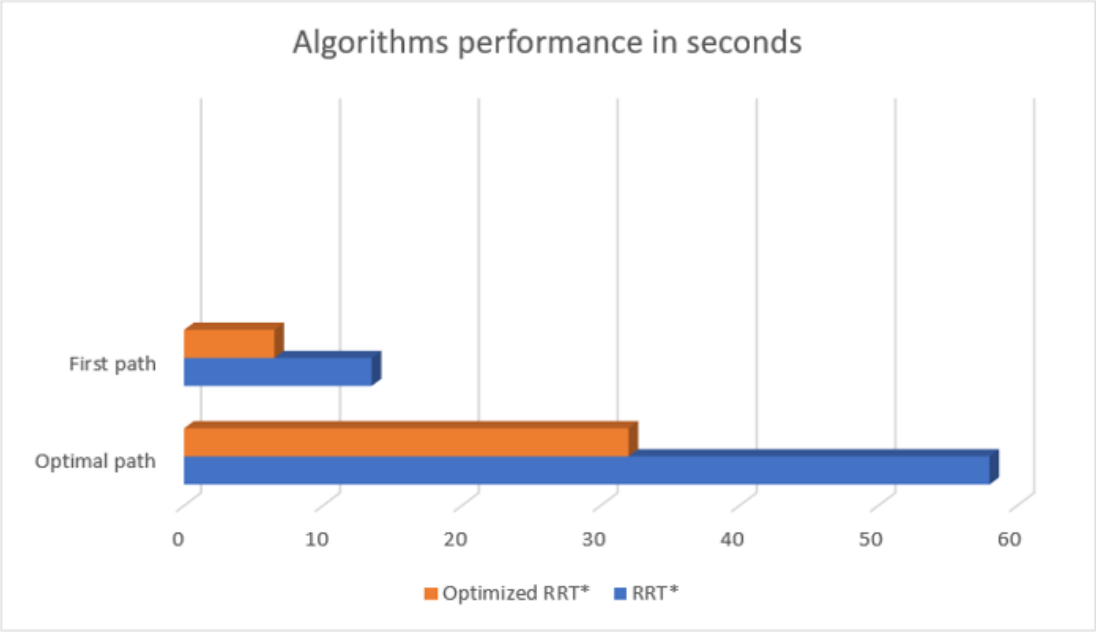

Figure 22
Algorithms performance in seconds

- Figure 23 shows the solution cost vs the CPU time. Both algorithms will run until they find a solution of the same cost (blue dot). The Optimized RRT* algorithm is performing better than the RRT*, i.e., it is computationally cheaper.

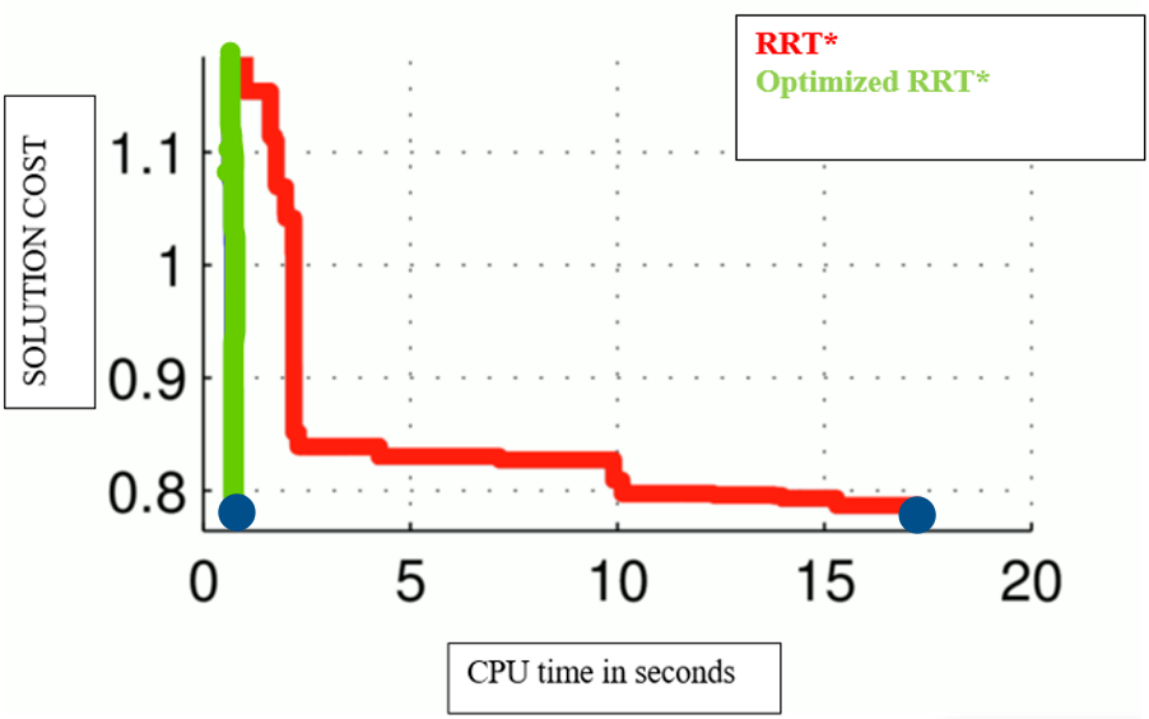

Figure 23
Solution Cost vs CPU

9 APPLICATION, IMPROVEMENT POTENTIALS, AND FUTURE WORK

9.1 Application

The Optimized RRT* algorithm is a sampled-based algorithm, an improvement of RRT*, which further is an improvement of RRT. This means, that the Optimized RRT* is a good option for a higher dimensional configuration state of an unknown environment. This property of the algorithm makes it highly functional and useful as there are no computational costs or other constraints. Its probabilistic completeness means that the probability of a solution (if existent) is going to be returned. This is possible when the number of nodes approaches infinity.

9.2 Improvement potentials

Heuristics are used by Informed RRT* to reduce the planning issue to portions of the original domain. Because it cannot focus the search when the related prolate hyper-spheroid is greater than the planning issue, it is intrinsically dependent on the cost of the existing solution. Like that, it can only reduce the subset to the lower bound specified by the best possible answer. We are looking for methods to concentrate the search without requiring an initial answer. These methods gradually expand the search subset. They prioritize the first search for inexpensive solutions by doing this [14].

9.3 Future work

Optimized RRT* needs still to be tested, so more precise and accurate results can be given. Improvement research and experiments should be done, to verify that the proposed algorithm is a good alternative to a dynamic environment, which is constantly changing. Further work and testing should be done, to become optimized enough to a point to be applicable for a real-time scenario. I plan to conduct more research to develop an algorithm for coordinated behavior in the navigation of several mobile robots.

10 SUMMARY

Robot motion planning is one of the crucial parts of the field of robotics. As the aim of robotics is to create an autonomous robot that will be able to move and perform the wanted tasks on its own, it is of immense importance to make the robot move efficiently and successfully from a starting point to a given destination point. This research area plays a significant role nowadays as it has many applications. One of the most promising applicable fields is planetary explorations, autonomous cars, and microbiology. Of course, these are just three of the many more examples.

It is truly clear how important robot motion is. The main interest of this work is the focus on the currently known path planning algorithms. These algorithms have a mathematical background, more specifically they are based on geometry. This means, in simple words, that they are given in an algorithm form as a solution to a geometric problem in the plane or space. Of course, it can also be given in a higher configuration dimension, but I focus on 2D.

This work starts with the basis of robot motion planning, the needed background, and well-explained terms of this field. Then I dive into the many algorithms that are being applied. I analyze each one of the most important path-planning algorithms, compare them with each other and give their specifications and the scenario in which they are most often used. As a result, with the analysis and comparison, I suggest some improvements and ideas in a form of an optimized algorithm based on my research.

In this work, the novel optimized path planning algorithm called Optimized Rapidly Exploring Random Tree Star (Optimized RRT*) is introduced and explained. It is followed by its implementation in a static environment that contains unmovable obstacles. The implementation is done in MATLAB software. Two other planning algorithms from the same family are implemented as well. After the testing, I give the results in graphs showing the new algorithm’s performance. This thesis work finishes with the improvement potentials and future work.

11 SUMMARY IN HUNGARIAN

A robotok mozgástervezése a robotika egyik kulcsfontosságú része. Mivel a robotika célja egy olyan autonóm robot létrehozása, amely képes önállóan mozogni és végrehajtani a kívánt feladatokat, óriási jelentőséggel bír, hogy a robot hatékonyan és sikeresen mozogjon egy kiindulási ponttól egy adott célpontig. Ez a kutatási terület napjainkban jelentős szerepet játszik, mivel számos alkalmazással rendelkezik. Az egyik legígéretesebb alkalmazható terület a bolygókutatás, az autonóm autók és a mikrobiológia. Természetesen ez csak három példa a sok további közül.

E munka fő érdekessége, hogy a jelenleg ismert pályatervező algoritmusokra összpontosít. Ezek az algoritmusok matematikai háttérrel rendelkeznek, pontosabban a geometrián alapulnak. Ez leegyszerűsítve azt jelenti, hogy algoritmus formájában egy síkbeli vagy térbeli geometriai probléma megoldásaként adódnak. Természetesen magasabb konfigurációs dimenzióban is megadható, de én a 2D-re koncentrálok.

A munka a robotok mozgástervezésének alapjaival, a szükséges háttérrel és a terület jól magyarázható fogalmaival kezdődik. Ezután számos alkalmazott algoritmust tárgyalok. Elemzem a legfontosabb pályatervező algoritmusok mindegyikét, összehasonlítom őket egymással, megadom a specifikációikat és azt a forgatókönyvet, amelyben leggyakrabban alkalmazzák őket. Az elemzés és az összehasonlítás eredményeként a kutatásom alapján néhány javítást és ötletet javaslok egy optimalizált algoritmus formájában.

Ebben a munkában az új, optimalizált útvonaltervezési algoritmus, az úgynevezett optimalizált RRT* (Optimized Rapidly Exploring Random Tree Star) kerül bemutatásra. Ezt követi a megvalósítása statikus környezetben, amely mozdíthatatlan akadályokat tartalmaz. A megvalósítás MATLAB szoftverben történik. Ugyanebből a családból két másik tervezési algoritmust is megvalósítok. A tesztelés után az eredményeket grafikonok formájában adom meg, amelyek az új algoritmus teljesítményét mutatják. A dolgozat a fejlesztési lehetőségekkel és a jövőbeli tervekkel zárul.

12 REFERENCES

[1] M. Berg, M. Kreveld, O. Cheong and M. Overmars, Computational geometry., Springer, 1997.

[2] "ACM," [Online]. Available: www.acm.org . [Accessed 10 11 2022].

[3] R. Metz, *"A 'New' Face In CAD/CAM",* The New York Times, 1981.

[4] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki and S. Thrun, Pronciples of robot motion., 2005: MIT Press.

[5] V. Lumelsky and A. Stepanov, "Path planning strategies for point mobile automaton moving amidst unknown obstacles of arbitray shape," *Algorithmica,* pp. 403-430, 1987.

[6] I. Kamon, E. Rimon and E. Rivlin, "Tangentbug: A range-sensor based navigation algorithm.," *Int. Journalof Robotics research,* pp. 934-953., 1998.

[7] Y. V. Chavan, P. Y. Chavan and A. Nyayanit, "Obstacle detection and avoidance for automated vehicle: a review.," *J Opt 50,* pp. 46-54, 2021.

[8] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to algorithms., MIT Press: 2001.

[9] A. Stentz, "Optimal and efficient path planning for unknown and dynamic environments.," *International Journalof robotics and Automation,* pp. 1-42, 1995.

[10] C. Taylor, "Computational motion planning.," [Online]. Available: https://courses.edx.org/Week_9_Graph_Based_Planning_Method.pdf. [Accessed 5 11 2022].

[11] L. E. Kavraki, P. Svestka, J. Latombe and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces.," *IEEE Transactions on Robotics and Automation,* pp. 566-580, 1996.

[12] S. LaValle, Planning algorithms, Cambridge University Press: 2006.

[13] T. Chinenov, " Robotic Path Planning: RRT and RRT*, Medium," [Online]. Available: https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378. [Accessed 5 12 2022].

[14] J. Gammell, S. Srinivasa and T. Barfoot, "Informed RRT: optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic.," in *IEEE/RSJ Intnl Conference on Intelligent Robots and systems,* 2014.

[15] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning.," *The Internationa ÍJournal of Robotics Research,* pp. 846-894, 2011.

[16] C. Taylor, "Computational Motion Planning," [Online]. Available: https://courses.edx.org/Week_9_Graph_Based_Planning_Method.pdf. [Accessed 5 11 2022].

13 LIST OF FIGURES

14 LIST OF TABLES