Hamburg University of Technology
Vision Systems

Prof. Dr.-Ing. R.-R. Grigat

A Graph Based Path Planning Approach for the RoboCup Standard Platform League

Research Project

Felix Warmuth

April 2, 2019

Statutory Declaration

I, Felix Warmuth, born on 24.04.1990 in Eindhoven, the Netherlands, hereby declare on oath that I compiled this research project, submitted to Hamburg University of Technology (TUHH), on my own. I have used only the declared sources and auxiliaries.

Place and Date	Signature

Contents

List of Figures **iii**

List of Tables **1**

List of Algorithms **1**

1 Introduction **2**
- 1.1 Motivation 3
- 1.2 Scope 4
- 1.3 Path Planning Concept and Terminology 5
- 1.4 Existing approaches 9
  - 1.4.1 Probabilistic Road Maps 9
  - 1.4.2 Rapidly Exploring Random Trees 10
  - 1.4.3 Visibility Graph 12
  - 1.4.4 Potential Field 13
- 1.5 Requirements 14

2 Prerequisites **17**
- 2.1 Calculation of Tangents 17
- 2.2 A-Star Graph Search (A*) 20
  - 2.2.1 Heuristic control 20

3 Algorithm **22**
- 3.1 Basic Concept 22
- 3.2 Data-types 27
  - 3.2.1 Input-Vector 27
  - 3.2.2 Graph 28
  - 3.2.3 Search Algorithm 28
  - 3.2.4 Output, the trajectory 29
- 3.3 Design 29
  - 3.3.1 Initialization 29

3.3.2 The Search Loop 33
3.3.3 Graph - Exploration of neighbors 36
3.3.4 Visibility check 40
3.4 Correctness 42
3.4.1 Initialization 42
3.4.2 The Search, modified A-Star 42
3.4.3 The Graph 43
3.5 Complexity 43

4 Implementation 45
List of Code Snippets 45
4.1 Development Framework and Visualization 45
4.1.1 Pygame and Matplotlib 46
4.1.2 Networkx as a graph visualizer 46
4.2 Python Prototype Code Snippets 46
4.2.1 cost(u: Vertex, v: Vertex, edge_type: EdgeType) 46
4.2.2 connected(u: Vertex, v: Vertex) 47

5 Evaluation 50
5.1 Comparison 51
5.2 Testing 54

6 Conclusion and Outlook 55
6.1 Further Work 56

Bibliography 57

List of Figures

1.1 NAO 2
1.2 Top4 Foul Statistic 2018 3
1.3 Global- and relative coordinate system 6
1.4 Work space example 7
1.5 Configuration space example 7
1.6 Example situation with 4 obstacles. 9

1.7 Probabilistic Road Map example 10
1.8 Visibility graph example. 12
1.9 Potential field, attractive and repulsive part 15
1.10 Detour example, Potential Field Approach 16

2.1 Tangent lines, point to circle. 17
2.2 Tangents line - two circles with same radii 18
2.3 Construction of tangents line - two circles with same radii 18
2.4 Construction outer tangents, different radius 19
2.5 Construction inner tangents, same radius 19

3.1 Basic Path Planning Problem. 22
3.2 Field Visualization of State 1 23
3.3 Graph Structure State 1 23
3.4 Field Visualization - State 2 24
3.5 Graph Structure - State 2 24
3.6 Field Visualization - State 3 25
3.7 Graph Structure - State 3 25
3.8 Full Sized Field Visualization - Final State 26
3.9 Visualization of Final Path with a rect as an obstacle. 27
3.10 Enlarged View of previous figure. 27
3.11 edges data type example 28
3.12 Example, Obstacle overlap, blocked arcs. 30
3.13 Shift of obstacle. 31
3.14 Example of start vertex inside of blocked area. 32
3.15 Destination inside of blocked arc, deletion approach. 32
3.16 Destination inside of blocked arc, shift approach. 33
3.17 Complex environment. 36
3.18 Intermediate vertex, it's neighbors and tangent edges. 37
3.19 Multi visiting an obstacle, blocked arcs, candidate and pair vertices. . . 40
3.20 Toughing edges are considered as visible. 41
3.21 Distribution of needed iterations. 44

4.1 List of angle placement scenarios regarding a blocked arc. 48

5.1 Reference path, proposed approach 50
5.2 PRM, final path, showing detour 51
5.3 Potential field, 3D, reference situation 52
5.4 Potential field, 2D, reference situation with contour lines 53

List of Tables

3.1 Observed number of function calls, mean over all 2500 runs. 44

5.1 Distance compartment, direct vs proposed approach. 54

List of Algorithms

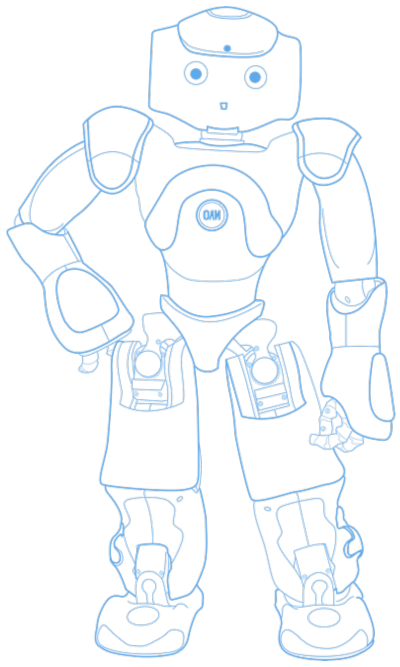
1 Rapid Exploring Random Trees 11
2 Sample A* pseudo code 21
3 Plan the path, modified A-Star 34
4 The Cost Function 35
5 SkipIfPossible, Can the pair vertex be added. 35
6 Get neighbors of vertex 38
7 Populate obstacle 39

Chapter 1: Introduction

This work is part of the research activity of team HULKs (Hamburg Ultra Legendary Kickers), the RoboCup Standard Platform League (SPL) team at the Technical University of Hamburg (TUHH). The subject is the result of analyzing the experience during SPL competitions regarding path planning. It proposes a specialized approach to solve path planning issues and improve the trajectory shape. The introduction chapter shows a picture of the target hardware, a motivation why path planning is a game changer as well as a short overview of other existing approaches. Followed by a section dedicated to the requirements of path planning approaches in a soccer game and discussing the drawbacks of the presented methods.

The RoboCup is an annual robot competition that was conceived in 1995 and first held in 1997. RoboCup has its roots in robot soccer but it is growing. Many additional stages of the competition such as "RoboCupRescue", "RoboCup@Home" and "RoboCupJunior" besides the succor leagues exist. The RoboCup World Championships are held annually at different locations world-wide. The Standard Platform League (SPL) is characterized by the fact that participating teams work on the same hardware and compete in soccer games with each other. Changes to the hardware are not allowed resulting in the focus on software development. The rule set ensures autonomous behavior and limits the inter-team communication of each robot. The used hardware is a NAO, programmable humanoid robot developed by Aldebaran Robotics, a French robotics company which was acquired by SoftBank Group in 2015 and rebranded as SoftBank Robotics. The most recent NAOv6 is 57.3 cm tall and weighs 5.5 kilograms. A 1.91 GHz Intel Atom E3845 processor and two HD cameras are integrated in the head.

Figure 1.1: *NAO*



1.1 Motivation

An important and basic requirement for any moving agent is to be able to successfully navigate the surrounding environment. This fundamental need is a requirement especially on autonomous robots. In the use case of the SPL, quick movement on the field is one critical winning precondition in order to get hold of the ball quickly. The player who is first at the ball has a playing advantage. Observation and analysis and previous matches results show that reaching the ball fast seems a good idea. The rules include statements to avoid ignorant playing and disrespectful move which lead to broken hardware. In the intend to reach the ball pushing an opponent might result in a foul. Only in the immediate duel for the ball, little touches between opponent robots are allowed. Fouls are immediately punished by the human referee. These sanctions strongly influence the game. They are expressed by accumulating time penalties. Is case the referee issues a penalty the punished robot gets benched for a certain time. These time penalties should be avoided as far as possible.

It is obvious that a team missing one or multiple robots has a decreased chance to score a goal. Figure 1.2 shows the executed foul penalties of the semifinalists of the recent RoboCup Championship 2018 in Canada. The final placements of the team regarding the Championship is decreasing from left to right. Nao Team HTWK became Champion, B-Human 2nd place and so on. The black dashed line shows the average prosecuted pushings of the top teams during the event. Notice that the number of called penalties rise on the lower tiers. Only outlier is HTWK. This fact is caused by their tactic used during the competition. In short terms, their tactic was: "Walk the ball into the goal". The other teams use a more conservative tactic and are at least comparable. One can see a correlation between called pushings and achieved rank. Team HULKs, where i participate, did most of them all. TJArk did 25% less than and B-Human just did 40% of Team HULKs with a total of 29 pushings called during the event. The motivation to improve this game changing issue is the goal of this work. These time penalties should be not caused by bad path planning.

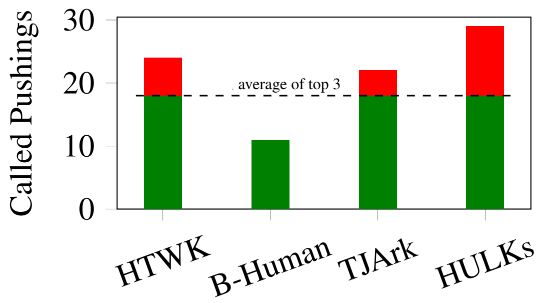


Figure 1.2: *Official called fouls of all semifinalists at the World Championships of RoboCup 2018, Montreal, CAN. Data: [Com18b]*

The observation shows that the current used path planning seems to not avoid penalty calls strong enough. These problems should be solved with this work. Therefore a whole rework form scratch is done. The focus lies on a robust and fairly simple path planning. Main goal is to avoid additional penalties but still taking the shortest route towards the desired position. the requirements are deeply touched in section 1.5

1.2 Scope

This section covers the scope of the work. As mentioned at the beginning of the introduction chapter, this work is part of the research of the HULKs. The HULKs framework implementation is based on modules, each handles a certain task. The overall process and data flow can be roughly split into 3 parts. The perception, decision and execution part. There are the data acquisition modules handling all kind of perceptions tasks, analyzing the robots neighborhood with all available sensors, like camera, sonar and even touch sensors. This data is than filtered using filter modules resulting among others is a map of the landscape and the robots position. Information is also exchanged with team mates. This information is than passed to the processing modules which decide how to react on intercepted world state. For example where to move next on the field. In the end, the motion modules take care of the physical motions done by the robot, from high level movements like walking or turning the head to just moving a joint with a certain force to a certain angle. This workflow is repeated in loops.
The path planning tackles only a small problem of the whole. It is supposed to find the shortest path between two locations. The two locations are the start and the destination position. The start locations is always the positions of the robot in global coordinates. The destination is given by a processing module in the brain part of the implementation. It decides where to go when. The path planning integrates between the high-level decision and the following motion modules. It is just outputting a path which should be followed. The resulting path should avoid obstacles in a meaningful way. If the destination is inside of an obstacle, the path sill leads to it and not stop somewhere intermediate. If the destination is not reachable, surrounded by obstacles, a feedback should be given to the deciding modules.
The problem which should be solved is narrowed down to a path planning problem.

1.3 Path Planning Concept and Terminology

In robotics, the term path planning is the process defining a path which the robot has to move based on a given situation. Another term sometimes used anonymously is motion planning. Using an example makes distinguishing between motion and path planning easy. For example: Someone wants to travel from Hamburg to Munich by car, the GPS navigation would be the path planning with high-level instructions like, "turn left in 2km". The driving itself, steering, acceleration and breaking would be the motion planning part. Motion planning tries to follow the path established by the path planning process considering the physical circumstances of the vehicle.
The path planning process can be optimized regarding different goals like, finding shortest path (distance) or to minimizing the travel time. Constrains could be avoiding sharp turns and keeping a certain distance from any obstacle any time. Depending on the application further constrains can be needed to be taken into account. Popular example is multi destination path planning, with interim locations, shown in the Traveling Salesman Problem. Often path planning must get repeated as often as possible, because the status of the given situation, information environment changes over time. Therefore the path selected at the start of the journey can be wrong or not feasible anymore.
The earlier introduced application of autonomous robot soccer, the path planning process takes place on each individual robot. The control of detailed agitation of the robot, e.g. moving its legs, turning, keeping balance, etc. is no part of path planning.
As soon as the environment around the robot is crowded by other robots or obstacles, a collision-free trajectory is difficult to archive. Formulating a path planning problem leads to the need of necessary information, like:

- Actual location of the robot.
- Desired target location.
- Size and shape of the robot.
- Description of the world, e.g. locations of obstacles.

The expected behavior of the robot must be formulated as well. To make resulting behavior comparable, specific and measurable requirements need to be established. Each requirement needs to be judged in their relative importance. The optimal result should be a movement from start to destination that the robot can follow under the constrains not touching any obstacle at all, be short in distance and fast in time.
Describing the environment can be done in relative coordinates of the robot itself or with help of a global coordinate system. The HULKs framework uses right-handed coordinate systems. In case of path planning, the z-axis is going to be ignored, so we assume that the robot and the world are only 2d. The global coordinate systems origin will be

found at the center point of the field. The orientation depends on the playing direction of the current match. By definition the own goal is located on the negative half-space of the x-axis. Figure 1.3 shows how the global coordinate system is aligned regarding the field in a state of playing from left to right. Also an example of a relative coordinate system inside of the global coordinate system is illustrated. The relative coordinate systems are located inside of each robot and the x-axis points forwards and the y-axis to the left. Further in this work the path planning will take place in global coordinate to make examples comprehensive and clear.

The path planning problem needs be formally described. To simplify the description it is not given in the real world, it gets transformed to another space. This transformation gets rid of any robot properties like shape or move-ability constrains and is called *configuration space* (C-space). The *configuration space* represents the space which can be reached by the robot. The physical shape of the robot as well as the dimensions of freedom (DoF) have impact on the *configuration* space. A robot's *configuration* specifies information about the robot such as position, proportions and joint angles. Therefore, the *configuration* space has exactly as many dimensions as the robot has dimensions of freedom.

In case of the path planning problem in this work the configuration space is very simple. By making simplifications regarding the robot's shape and movability the configuration only contains the position, therefore *configuration* space will have 2 dimensions and the robot will be represented as a point. The real proportions and shape of the robot are included in the size of obstacles, generally such intense simplification is not the case. The shape of the robot is simplified to a circle. The circles introduce a rotation invariant behavior. It doesn't matter what orientation the robot is in, a circle stays a circle.

Using these simplifications the transformation from working space to *configuration* space can be shown as a picture. Figure 1.4 and 1.5 illustrate the transformation. As illustrated the obstacle becomes larger. The size increase is half of the robot's diame-

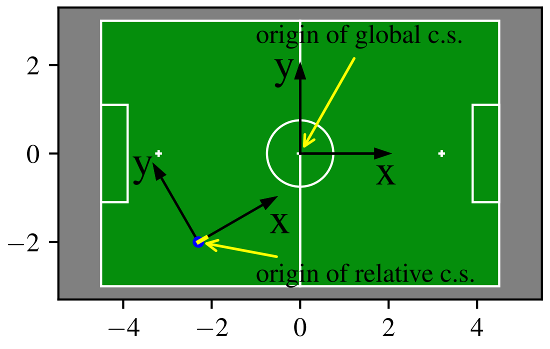


Figure 1.3: *Origin of the global coordinate system (c.s.) is located at center point, pointing to opponent goal. Relative c.s. placed at center of the robot.*

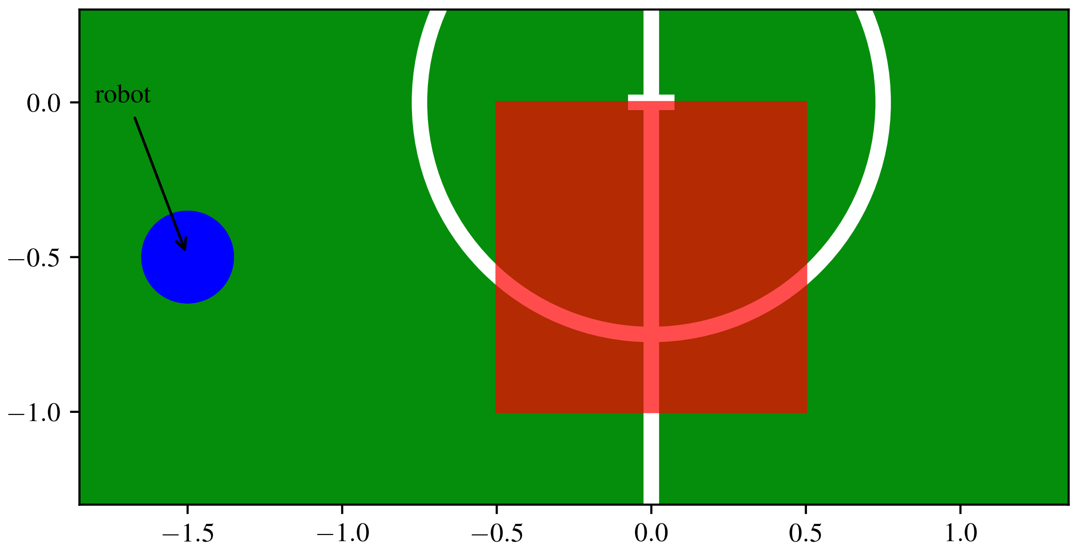


Figure 1.4: *Work space of a circular robot in a open area with a rectangular obstacle.*

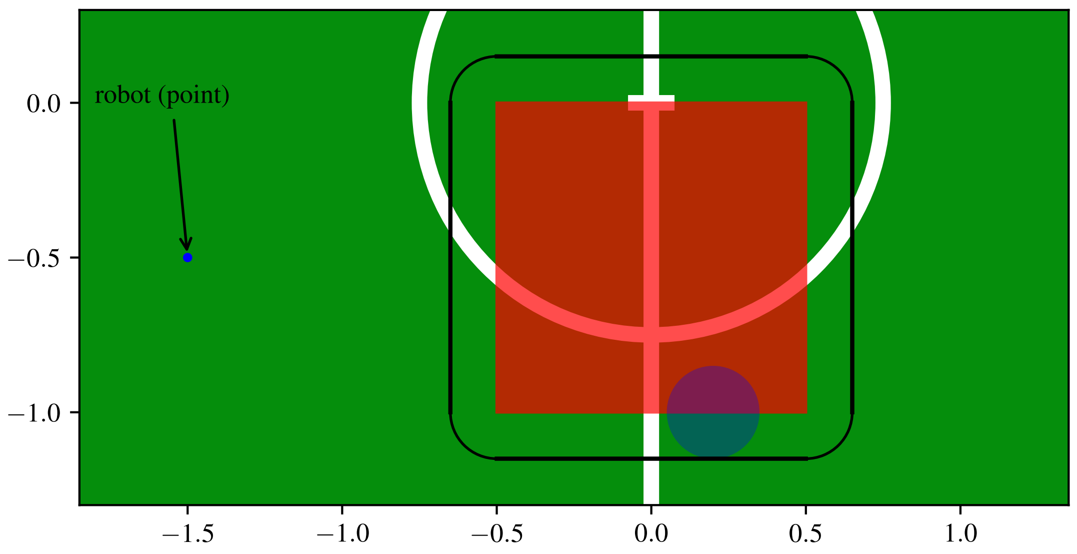


Figure 1.5: *Configuration space of a circular robot in an open area with a rectangular obstacle. Area inside*

ter. Such C-space can be obtained by sliding the shape of the robot along all edges of blocked regions. This can be seen as a simple dilation operation ($\bigoplus$) in mathematical morphology. During a dilation, a so called structuring element is used to expand another input element. Using the circular shaped robot as the structuring element and the environment of the robot as input, the blocking edges get broaden. The resulting free space represents all locations where the center of the robot can move without touching any blocking edge. It is called *free space*. This can heavily depend on the rotation of the robot, if the robot shape is not rotation invariant like the circular robot in the example. This will enhance the complexity of the path planning. Let $W = R^m$ be the work space with m the dimensions, $O \in W$ the set of obstacles, $R(c)$ the robot in configuration c with C being all configurations.

$$C_{free} = \{c \in C \mid R(c) \cap O = \emptyset\} \tag{1.1}$$

$$C_{blocked} = C / C_{free} \tag{1.2}$$

Further let c_{start} and c_{dst} be the start and the destination configurations. Than a mathematical formulation of the a path planning problem will narrow down to the following equation:

$$\begin{aligned} &\tau : [0,1] \rightarrow C_{free} \\ &\text{with } \tau(0) = c_{start} \text{ and } \tau(1) = c_{dst} \end{aligned} \tag{1.3}$$

With the help of the configuration space, one can do path planning with the circular robot just being a point in the configuration space. If the real world is continuous, it needs to be discretized for the path planning. A straight forward approach would be sampling the environment and use some kind of collision detection to test certain spots and incrementally search for a solution in in the configuration space. These methods are called sampling base methods. Popular examples would be the Probabilistic Roadmap (PRM) approach or Rapidly Expanding Random Tree (RRT) method. They characterize C_{free} into a graph in which vertices are configurations and edges are representing collision-free paths through C_{free}. This graph is then searched for a solution. The graph is kind of a discretasation and sampling. The sampling method varies a lot. A totally different approach takes the Potential Field method. It is not based on a graph but on a potential field and a gradient decent method to find the minimum of that field. See section 1.4.4 for detailed information.

1.4 Existing approaches

This section consists of a small selection of existing approaches. Some of the examples shown refer to the follow example situation. Figure 1.6 shows a typical illustration of a soccer field. This type of figure will appear very often in the later chapters and sections. Obviously it deals with a soccer field, showing both penalty areas as well as the center circle and other field marks. The orange circles represent known obstacles, such like other robots or areas that need to be avoided. The yellow-blueish point on the left hand side shows an arbitrary starting position of a robot.

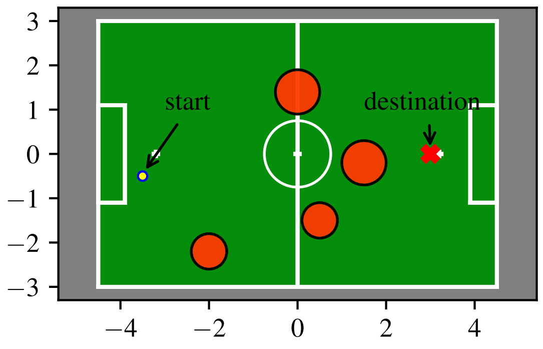


Figure 1.6: *Example situation with 4 obstacles.*

The red cross marks the desired destination. The field shown has the correct dimensions, the axis are marked in Meter, and are given by the SPL rule book.

1.4.1 Probabilistic Road Maps

The Probabilistic Road Map (PRM) method is a typical path planning approach based on sampling the environment. It is widely used and studied [GO04]. The approach samples the configuration space for configurations which are in $\mathbf{C}_{free}$. Each valid configuration is then added as a vertex to a graph, the road map. As the name suggests the sampling process uses a probabilistic approach. It generates possible configurations at random. This can happen without any structural information about the whole configuration space therefore without knowing the environment. The only requirement is a way to check if a vertex is inside of $\mathbf{C}_{blocked}$, to determine a collision, and if not adding it to the graph. The second step of PRM is to connect promising nodes inside of the graph. This is done by a local planner until the graph sufficiently represents the environment.
The description given above is just a rough guide on how the principal works, but it leaves details open. How to sample the configurations space reasonable, how does the local planner connect promising node, what are promising nodes?
Figure 1.7 illustrates a resulting graph of a typical PRM approach. The start and destination are marked as a blue circle with a red dot and a red cross, respective. The smaller red dots are vertices of the graph. The yellow edges show the interconnection between them. The orange larger circles are obstacles which need to be avoided. As shown the graph connects the start and the destination, accomplishing the goal of path planning. But it is obviously not the optimal path and it is not smooth at all.

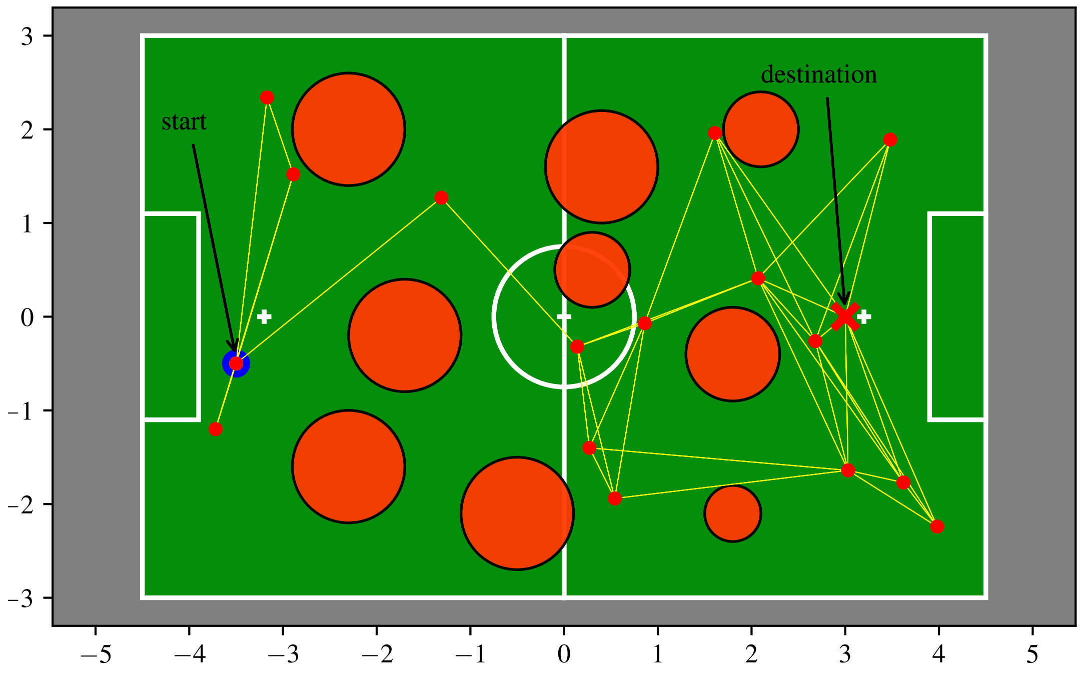


Figure 1.7: *PRM example. Shows the vertices generated by uniform random function and the connecting edges resulting in the graph of a PRM approach.*

Recent publications show that the PRM method is still interesting. [SGS17] shows that PRM combined with the cubic Ferguson's spline technique can generate smooth trajectories. It uses a classic PRM in the first step and simply refines it to make it very smooth. The proposed algorithm is also optimally asymptotic and can satisfy direction limitations of the start and destination location.
In [KUT18] the authors presents a multi-query sampling-based planner resting on the PRM method. It can handle diverse types of planning uncertainties. It uses a sample classification technique to identify uncertain samples. This mechanism seems to be able to react to uncertainties by using a layer of nodes (samples) around the sensitive one. All samples are saved in a matrix-grid structure. Simulation results show the efficient performance of the proposed planner in producing semi-optimal solutions with low computational cost.

This allows this planner to be applied to almost any kind of robot with arbitrary DOF.

1.4.2 Rapidly Exploring Random Trees

Initial approach published in 1999 by Lavalle and Kuffner in [LK99] under the name Randomized Kinodynamic Planning. Nowadays called Rapidly Exploring Random

Trees (RRT) is a search algorithm and belongs to the sampling approaches. The basic idea is to probe and explore the environment in a random manner starting from an initial position. The exploration takes place in the configuration space and will result in an incrementally expending tree. The tree will mark the explored environment with the root being the initial position. This algorithm is often used for motion planning and for efficient movements of robots as in [SS15] and [QMI$^+$14].
The basic procedure of the sampling process is shown in algorithm 1.

Algorithm 1 Rapid Exploring Random Trees

```
1: **graph**.init(initial position)
2: **do**
3:     $\mathbf{C}_{rand} \leftarrow$ Random configurations
4:     $\mathbf{C}_{closest} \leftarrow$ CLOSEST(**graph**, $\mathbf{C}_{rand}$)
5:     **graph**.addEdge($\mathbf{C}_{closest}$, $\mathbf{C}_{rand}$)
6: **while** finished
```

In line 1 the graph is initialized with just the start position as the only vertex and no edge. Line 3 takes a randomized configuration out of C_{free} and calls it C_{rand}. This sample mostly comes form a region around the initial position for example a random translation of the initial position. The CLOSEST function is called in line 4 and returns a vertex inside of G which is closest to the C_{rand}. The CLOSEST function can use any metric defined in the configuration space. The last line, line 5, represents the expansion of the graph by adding an additional edge between the new random configuration and the closest vertex inside of the graph. Depending on the environment an additional collision check needs to be made so that the new edge does not interfere with any obstacle. Therefore, the new edge is not leaving $\mathbf{C}_{free}$ at any point. If an interference occurs then one could simply add an edge in the same direction but ending as close as possible at the obstacle.
Using this approach for path planning problems seems quite costly. Because it will only terminate if the $\mathbf{C}_{rand}$ hits the desired destination. This is quite unlikely. To overcome this, just start the RRT and every 1000th (arbitrary guess) iterations force $\mathbf{C}_{rand}$ to be the desired destination. Subsequently, as soon as the destination is insight in the configuration space the likelihood adding such an edge to the graph is very high.
However some environments require even more specialization. In some cases it is necessary to use a bidirectional search approach. Grow two separate RTTs, RRT_s and RTT_{dst} with their random chosen configurations s_{rand} and dst_{rand}. One starting at the initial position and the other at the desired destination location. After arbitrary steps, try to connect the closest vertex of each with the other. So that $closest(RRT_s, dst_{rand})$ and $closest(RRT_{dest}, s_{rand})$ form the new edge. This will direct the search towards each other.

Overall RTTs are quite popular and many additions and modifications exists. For example [NRH15] presents an algorithm for real-time path-planning in a dynamic environment based on RRTs. Another variant is the informed RRT. It improves the convergence speed of RRT by introducing a heuristic, similar to the way in which A-Star improves the basic Dijkstra algorithm, [GSB14]. RTT approaches provide a good balance between an greedy search and exploration approach. They are fairly easy to implement. Problem is the unknown rate of convergence and their sensitivity when it comes to parameters, like the metric.

1.4.3 Visibility Graph

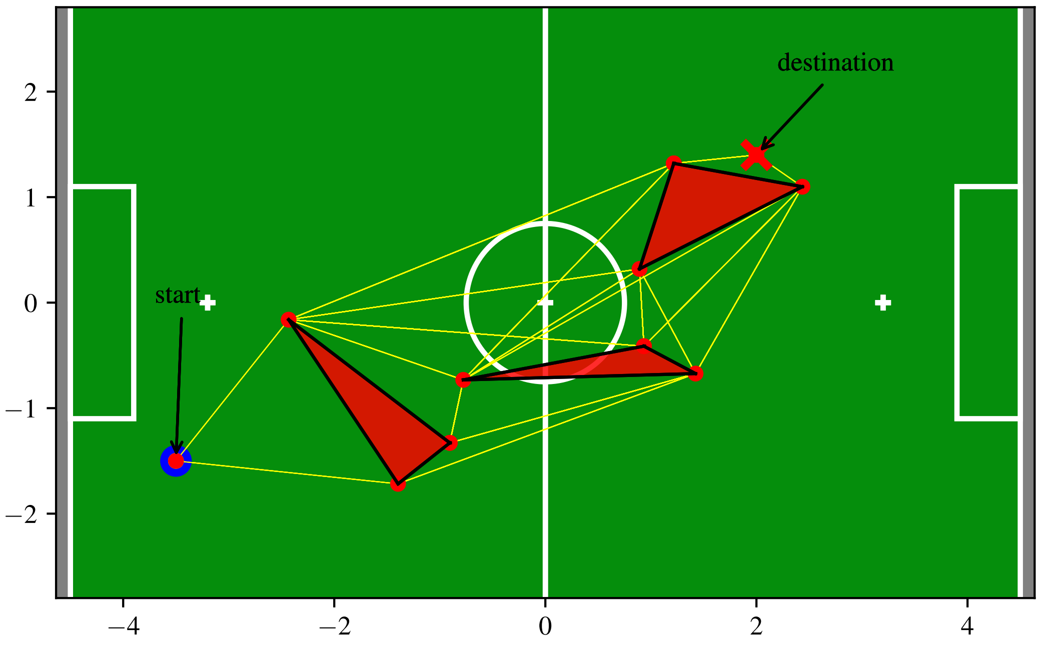


Figure 1.8: *Visibility graph example.*

The Visibility Graph is a popular concept and can be used in motion and path planning, [RId88]. Without any adjustments it works with a point shaped robot and polygon shaped obstacles. Fundamental idea is to represent the collision-free space with a graph. Nodes lie only in the free space and neighboring nodes, connected by an edge, are collision-free and simple to reach. The graph should represent the work space fair enough. As the previous approaches, the graph will reduce the path planning problem to a graph search. The concept of constructing the graph is simple. An example of a visibility graph is shown in figure 1.8 illustrating a partial soccer field with triangular

obstacles. The obstacles are the 3 orange triangles, in general any polygon can be used. The yellow edges show the neighbors of each vertex. As one can see only polygon corners are vertices in the graph, except the start and destination, and all edges are straight line segments. This observation implies the following: the shortest path from start to destination must be shown n the graph.
The edges get constructed using the visibility constrain. The neighborhood of a vertex is always in "sight", the edge is interfered by any obstacle. Therefore, the shortest way between the vertices is represented by their interconnecting edge. This rule seems simple but it is computational expensive. [AAG$^+$85] shows how to take advantage of a special data structure from which it is possible to compute visibility problems nicely. The authors use a method to efficiently calculate line segment intersection in a collection of line segments. This method is known as the sweep line algorithm and finds usage in computational geometry, [BCKO08]. The idea of such an approach is to use an imaginary line which moves over a plane. It stops at some points. All geometric operations, for example the intersection calculations between line segments, are only done at objects that are close or intersect with the so called sweep line at these stopping points. After the sweep line passed over all objects, the complete solution is done. This approach is also used during the initialization of the presented algorithm in chapter 3 in section 3.3.1.
Using this graph with a simple graph search results in an optimal path. The weight is given by the Euclidean distance between vertices of an edge. The computational effort constructing has been pushed down during the recent decades. The naive approach takes $\mathcal{O}(n^2)$. An algorithm presented in 1987 by Subir Kumar Ghosh and David M. Mount introduced the output sensitive property and a complexity of $\mathcal{O}(e + n * log(n))$, where e is the number of edges in the visibility graph and n is the total number of vertices in all the obstacles, [GM87].
However, looking for only one shortest way there is an algorithm with optimal run-time of $\mathcal{O}(n * log(n)$ presented in 1999, [HS99].

1.4.4 Potential Field

[JW10] The current implemented solution of team HULKs is based on the idea of the potential field approach. The other approaches try to navigate through a graph connecting configurations, or in case of rotation invariant robot, positions inside of C_{free}. The potential field method follows a different approach. The robot is represented as a point in C_{free} which acts just as a particle in a potential field. The environment influences the particle in such a way that it floats from start to destination. Inside of that potential field, repulsive forces form all obstacles and an attractive force pulling the particle towards the destination superimpose each other.
Let $\mathbf{U}_{rep}(c)$ be the sum of all repulsive forces of the obstacles at configuration c and

$\mathbf{U}_{att}(c)$ the pulling force of the destination at c. Leading to the potential function:

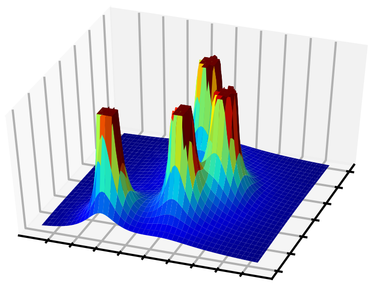
$$\mathbf{U}(c) = \mathbf{U}_{rep}(c) + \mathbf{U}_{att}(c) \tag{1.4}$$

This approach will result in a smooth trajectory around all obstacles. The situation shown in figure 1.6 is used as an example to visualize such a potential field. The repulsive effects are shown in figure 1.9a, each obstacle creates an effect seen as a mountain in the plot. Figure 1.9b illustrates how the destination location is acting like an attracting sink hole. The summation of both is shown in figure 1.9c. A particle, a robot, can calculate at any point in which direction he has to move to reach its desired destination. The computation is a simple gradient decent approach. As seen in the plots above, figures (1.9a, 1.9b, 1.9c), the configuration space is discretized in a grid. Depending on the size of the used grid the computational effort can vary heavily. Another problem is the possibility of dead ends without any chance to get out using gradient decent. Such a local minimal can be seen around $(1,-1)$ between the two obstacle which are close to each other. A solution could be to constrict a local minima free environment by modifying the characteristic of the obstacle, like size or impact on the potential field. Explicit calculation of the free C-space can be very costly.

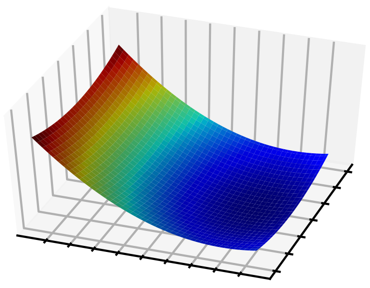
1.5 Requirements

The path planning problem faced in a SPL game can be solved using the mentioned approaches. But these methods introduce drawbacks. For example the Probabilistic Road Map approach is not optimal and can lead to not solving the problem at all. Which is a bad behavior during a match. The currently used approach using potential fields can lead to dead ends for example.
The strictly limited environment encountering during matches can be used to optimize existing approaches and special requirements can be implemented. A solution perfectly tailored to the scenario is desired. A general solution to all ever upcoming path planning problems is not needed. This section sketches the demands and requirements of a path planning solution.

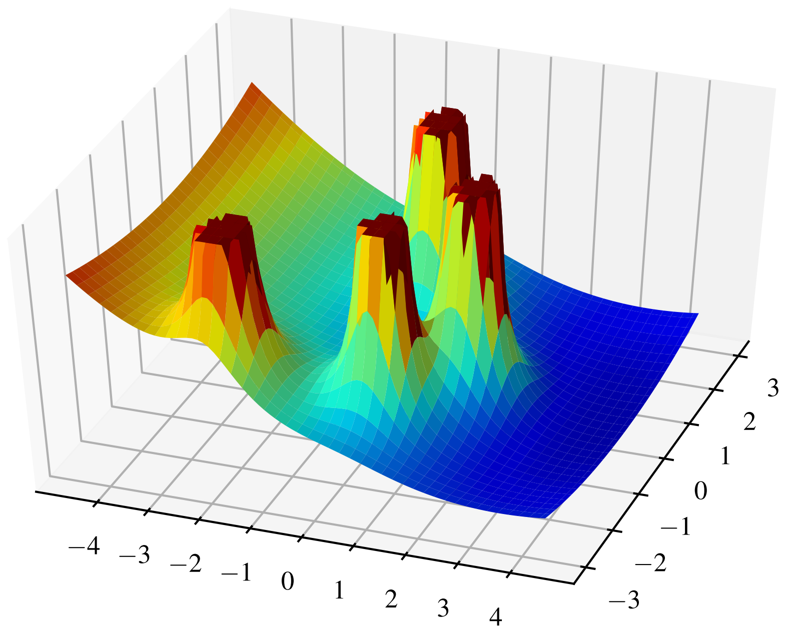
The shown approaches seem promising. But some of them are computational too expensive an other do not provide the wanted features. The essential requirement is to minimize length of the path. The robots are fairly slow and being quick at a wanted position improves the overall game performance. A constrain would be the necessity of a collision free motion, as mentioned in section 1.1. Apart from following rules, obstacle avoidance can also help being quick at a position. If a robot falls, it loses time. The stand up motions are very time consuming. Falling has an additional impact on the robot itself. Breaking arms, legs or even loosing a head occurred in the past. Stressing joints, e.x. during stand up motions, lead to high load in the actuators impacting the



(a) $\boldsymbol{U}_{rep}$



(b) $\boldsymbol{U}_{att}$



(c) $\boldsymbol{U}(c) = \boldsymbol{U}_{rep}(c) + \boldsymbol{U}_{att}(c)$

Figure 1.9: *Summation of attractive and repulsive effects. Resulting in a potential field used for path planning.*

heat. Heated joints tend to loosen up and joint measurements become inaccurate which finally lead to very unstable walking. The humanoid design of the robot comes with some limitations considering the shape of the path. Trajectory should be smooth and continuous without any sharp corners. Turning on spot needs a lot of tiny steps. The hip joint movement is here the limiting factor. Tiny steps have a strong impact on the speed introducing additional delay on reaching the desired position. Observations showed that small diagonal or circular trajectories have less negative impact on the overall traveling

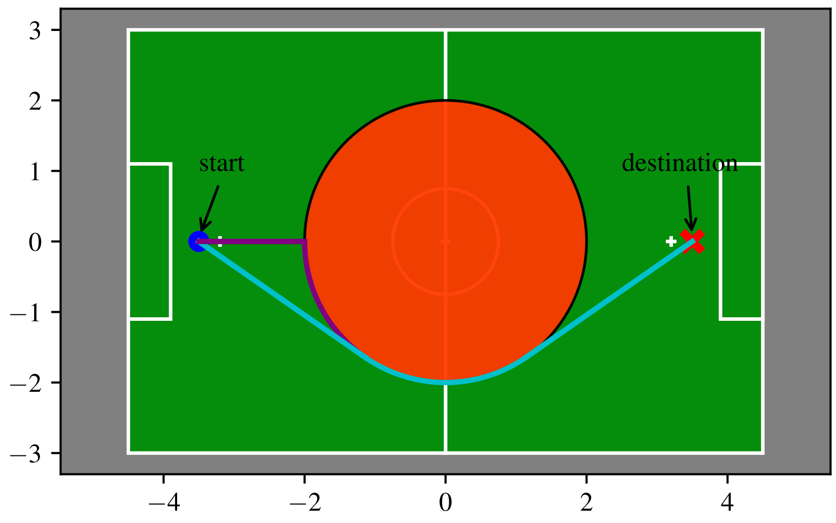


Figure 1.10: *Example showing the detour produced*

speed of the robot, therefore a certain smoothness of the path is wanted.
Reviewing the existing approach considering the above constrains exclude some entirely. The only presented existing approach meeting these requirements seems to be the VG approach, 1.4.3. The Probabilistic Road Map approach, sec. 1.4.1, doesn't guarantee finding a path at all. Also the length of the resulting trajectory depends strictly on the smart probabilistic positioning of the sample points. Each visited way point will result in turning on the spot to align to the next edge. To overcome the corners in the trajectory one could smooth it after. [SGS17] proposes a trajectory planning method for a mobile robot based on Probabilistic Roadmaps combined with the spline technique to generate smooth trajectories. But introducing such a method seems like a complication of the task. The HULKs use currently a potential field approach. Apart from the possibility of not finding a path at all, a major drawback is, it only reacts on close obstacles to overcome this one could increase the radius and the height of obstacle in the potential field. Introducing a large repulsive force impact size, but that will also lead to a much greater detour. The size of an obstacle as direct influence on the avoidance radius. No early reaction at least with small obstacle sizes can be archived. The importance of an obstacle having very early impact on the path planning is illustrated in figure 1.10. The detour cased by not reacting early on obstacle within the potential field approach is marked in yellow. In contrast the blue optimal trajectory is shown.
The optimal path is found using the Visibility Graph method. It seems to be a Swiss army knife, it is optimal, no detours are needed and it is complete, if there is a path it will find it. Problem could be that it is based on a pre-built graph, computing this graph can come with computational cost.

Chapter 2: Prerequisites

This chapter presents prior knowledge and used terminology as well as additional information regarding chapter 3. It is used as a reference in some section of chapter 3 and is not specially needed to understand the presented and developed approach. Reading it before hand is not a necessity.

2.1 Calculation of Tangents

The calculation of tangent lines plays a major part in the underlying graph structure. This section describes how tangent lines and their related points can be calculated. The approach presented in chapter 3 states two types of situation in which a tangent problem needs to be solved. The first would be to find the tangent lines between a point and a circle. Figure 2.1 shows the resulting scene.
To anticipate, the approach introduced in chapter 3 will only lead to this problem with

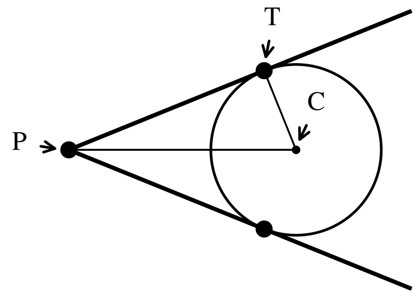


Figure 2.1: *Tangent lines and corresponding tangent points between a point* ***P*** *outside of a circle. C center of the circle and one tangent point T.*

the point located outside of the circle. This fact simplifies the calculation, no other cases need to be handled. This calculation is well documented which is why only a brief insight is given.In the following part $\overline{PT}$ represents the line defined by point P and point T. A line segment starting by point P and terminating at T is shown as $\overrightarrow{PT}$.

$\overrightarrow{PT}$ is only an tangent if $\overline{PT} \perp \overline{CT}$ and $|\overrightarrow{CT}|$ is equivalent to the radius of the circle. That implies that $\overrightarrow{PT}$ is the tangent and T the tangent point of P and the circle. Using the Thales's theorem and an auxiliary circle located half across $\overrightarrow{PC}$ one can easily calculate the tangent point coordinates. The other situation which will occur is the computation of tangent points between two circles. In general there will be 4 tangent points at each circle, 2 inner and 2 outer tangents lines. Figure 2.2 shows that case. If the circles overlap each other, the inner tangents will not be defined, leaving only the outer tangents. If the circles are completely inside of each other no tangent exist at all.

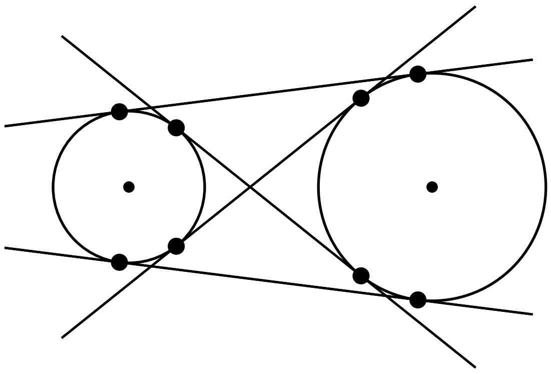


Figure 2.2: *All* 4 *tangent lines and the corresponding* 8 *point between two circles with* ***different*** *radius.*

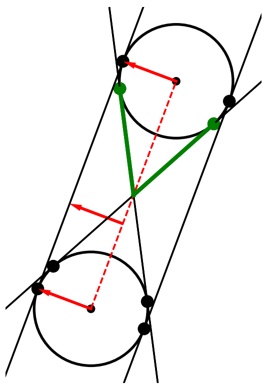


Figure 2.3: *Construction method between 2 circle with same radius.*

The computational effort is depending on the overlap state of the circles. Further, only cases where tangents exist are touched. Edge cases need to be handled elsewhere.
Both circles having the same radius implies that the outer tangents are a translation of the center connecting line segment. The translation takes place perpendicular to the connecting line segment direction. The length of the translation is the radius of the circles. Also the inner tangent calculation is simple. It reduces to 2 point to circle tangent calculations, just as the previous discussed case. The location of point, P, is determined halfway between the two circle centers. Figure 2.3 illustrates the situation and the resulting simplifications. The red dashed line shows the center connecting line and the red arrows one translation direction. To construct both outer tangent points also a translation in the opposite direction is needed. The green line segments are the result of one point to circle tangent calculation

using the corresponding point. Repeating this for the other circle will result in all 8 tangent points.
In case the circles have different radii, the calculation method is changed. Such as the prior method used to compute the inner tangents, the problem gets modified so it can be solved using the introduced point to circle tangent calculation. Figure 2.2 shows the outer tangents are not longer parallel to the center connecting line segment. So a simple shifting will not help. Calculating the outer tangent, an auxiliary circle is used to compensate the radius difference between the two circles. Its center is located at the center of the larger circle and the radius is the radius difference of the two circles. Figure shows the auxiliary circle in red. With that circle the point to circle method in combination with a translation is used to calculate the real outer tangents. The translation is shown with red arrows.

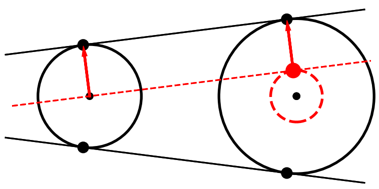


Figure 2.4: *Construction method of the 2 outer tangent line between circles with different radius.*

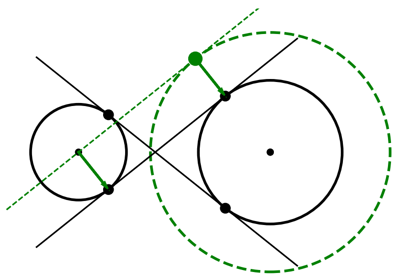


Figure 2.5: *Construction method of the 2 inner tangent line between circles equivalent radius.*

The inner tangents are calculated in a similar way, also using an auxiliary circle. The center is also the larger circle and the radius is the larger radius added with the radius difference. The auxiliary circle is drawn in green in Figure 2.5. Next is to again use the point to circle method to find a tangent between the auxiliary circle and the center of the smaller circle and translate it to the proper position.
Repeating this approach for all cases of outer and inner tangents will reveal all tangent points and their related tangent lines.

2.2 A-Star Graph Search (A*)

This section introduces the popular A-Star Graph Search. It was first published by Peter E. Hart, Nils J. Nilsson and Bertram Raphael in 1972 [HNR72]. The A-Star search algorithm finds the shortest path between 2 locations. It uses a graph with positive edge weights. Each location is represented as a node. The algorithm is considered as a generalization and extension of the Dijkstra algorithm. It extends the Dijkstra search to use a heuristic to focus the search towards the destination. The heuristic has impact on the progress of the search and can be used to control its behavior, see 2.2.1. As long as the heuristic is admissible A-Star guarantees the optimal path.
The algorithm works on a graph consisting of nodes and edges. These nodes can be grouped in 3 different collections. *Unknown* nodes: These are not visited yet and no path lead to them. At the start of the algorithm all nodes, except of the start node, are unknown and belong to this group. *Known* nodes: Nodes which are *seen* and a path does exist. This must not be the optimal path it could be updated in later algorithm steps. All known nodes are noted in the so called *openlist*. As mentioned at the start it only holds the start node. *Visited* nodes: Nodes to which the optimal path is found. These nodes are noted in the so called *closedlist*. This list is used to avoid looping and examine a node multiple times. This list is at the start of the algorithm empty. The two lists are sorted. The order giving value is the *F-Value*, it is calculated by adding up the real costs, $g(x)$ and the heuristic, $h(x)$ where x is the node. Each node inside of the *openlist* and *closelist* has a parent node which is used to backtrack the optimal path. A pseudo code example can be found at algorithm 2.

2.2.1 Heuristic control

At one extreme, if h(n) is 0, then only g(n) plays a role, and A* becomes Dijkstra's algorithm, which guarantees to find the shortest path. If h(n) is always lower than (or equal to) the cost of switching from n to the target, then A* is guaranteed to be the shortest path. The lower h(n) is, the more the node A* expands and the slower it becomes. If h(n) is exactly the same as the cost of switching from n to the target, then A* only follows the best path and never expands anything else that makes it very fast. Although you can't achieve this in all cases, you can specify it in some special cases. It is nice to know that A* behaves perfectly with perfect information. If h(n) is sometimes greater than the cost of switching from n to the target, then A* is not guaranteed to find the shortest route, but it can run faster. At the other end, if h(n) is very high compared to g(n), then only h(n) plays a role, and A* becomes greedy best-first search.

Algorithm 2 Sample A* pseudo code

```
Require: g_score, parent are hash tables
function SEARCH(start, destination, environment)
    initialization
    open.put(0, start)        → The queue is sorted by the first entry, 0
    while not open.empty() do        → main loop start
        current = self.open.get()        → node with smallest f-score
        if current == destination then
            return "Finished"        → destination found
        end if
        for all neighbor ∈ graph.neighbors(current) do
            new_g_score := g_score[current] + cost(current, neighbor)
            if (neighbor not visited) or (new_g_score < g_score[neighbor]) then
                Update or save new g-score and parent
                f_score = new_g + heuristic(neighbor, destination)
                open.put(f_score, neighbor)
            end if
        end for
    end while
end function
```

Chapter 3: Algorithm

The algorithm as devolved and investigated here is based on "A-Star graph, search" but modified in that sense, that the graph is built interactively during the search process. The proposed algorithm simultaneously searches and builds the graph. In contrast to a graph search which takes a graph as input, building the graph before hand is not necessary. The construction of the graph takes place parallel to the search. Therefore in each step of the algorithm the graph gets deeper and expends. Assuming the plain case during a SPL match with 10 robots in total can result in a visibility graph with at least $9 * 2 + 8 * 7 * 4 + 9 * 2 = 260$, nodes not including additional blockages. Combining search and build avoids the computational effort of building not needed parts of the graph. The resulting graph is a modified tangent graph, working with circular obstacles and the situation in an SPL game. The computed path only consist of alternating straight lines and circular arcs, producing a smooth and continuous path.

3.1 Basic Concept

This section provides the fundamental idea behind the algorithm. As mentioned in section 1.3, the problem is maneuvering from point A to point B in a 2-dimensional space. The space is likely not empty, use a straight line in between the points is not always the best choice or possible. The algorithm does avoid obstacles in a straight manner. An obstacle could be a circle, defined by a position and a radius. The first example will take a single obstacle in to account and put the start and destination locations on opposite sides of the obstacle. The resulting trivial path is shown in figure 3.1.

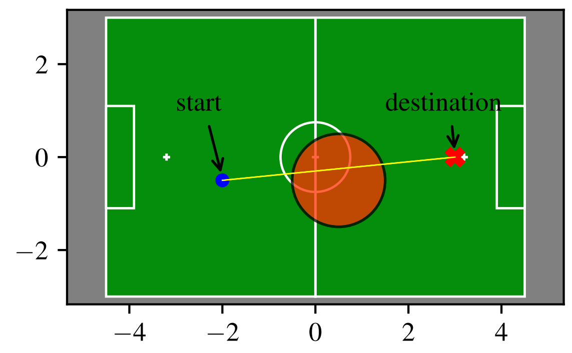


Figure 3.1: *Nontrivial path planning situation. One large obstacle between start and destination. Obstacle is shown by transparent orange circle.*

The robot is marked as a blue dot on the left hand side. Desired location is illustrated as a red cross. Task is to plan a path to the target position avoiding any collision with the obstacle. The obstacle is shown as the orange circle between start and desired destination. The shortest possible path, shown by the yellow line, interferes with the obstacle. The wanted output is the best trajectory avoiding the given obstacle. As shown in section 1.4 different concepts of creating trajectories exist. To find possible paths a visibility based approach is used.
In the first step the visibility approach leads to two possible locations. The state gets represented in a graph $G(V,E)$. V is the set of vertices and E is the set of connecting edges. Each vertex of graph G represents a point location, each edge represents a connection between them. Only vertices at inter-visible locations are connected by an edge and added to the graph G. In the first step in each loop of the algorithm, similar to A-Star, is to pop the first vertex from a priority queue. The priority queue is initialized with one vertex, the start location. In the first step, this vertex gets taken from the queue and is handled as the current vertex. Next step is to find the neighbors of the current vertex. In the example, using the visibility approach to find next possible positions in the free space leads to two valid locations. The resulting situation is illustrated in figures 3.2 and 3.3.

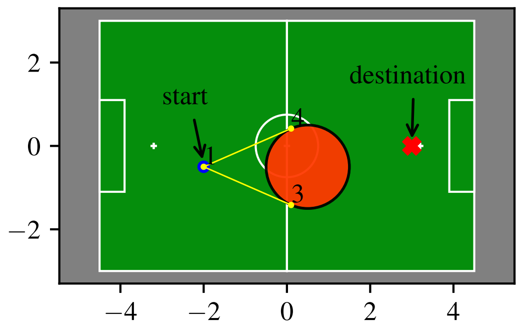


Figure 3.2: *Neighbor vertices of the start vertex based on the position of the tangent points of the obstacle and the start location.*

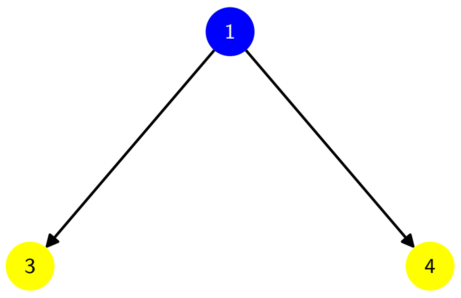


Figure 3.3: *Search tree, showing vertex* 3 *and* 4 *being children of the start vertex* 1.

Figure 3.3 shows the tree structure of the underlying graph. This structure will evolve further each interaction. The topology will always be a tree with the starting vertex as its root. As shown in figure 3.3 the two tangent points between start location and the circle of the obstacle define two possible positions. These two vertices are children and neighbors of the start location, as indicated by the yellow line connecting each. More about finding neighbors is described in section 3.3.3 focusing on the exploration method. In case of more obstacles visible from the current vertex in the environment, there would be more neighbors as well. Figure 3.11 illustrates a situation with 3 visible obstacles.

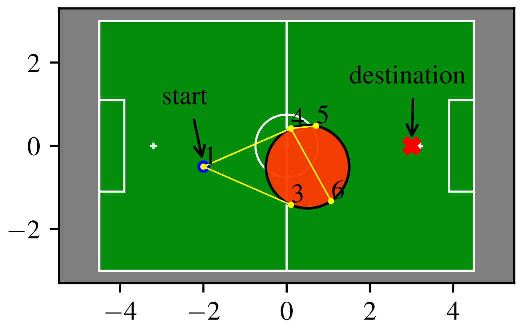


Figure 3.4: *Field with graph of all vertices, after 2nd loop.*

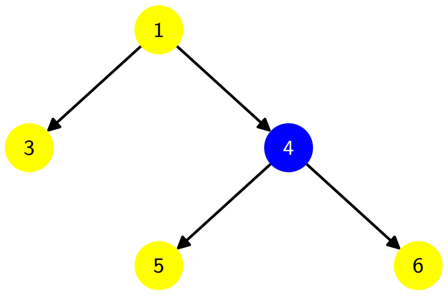


Figure 3.5: *Search tree. Showing vertex 4 as current vertex and its neighbors.*

The neighbors will be put in the priority queue. The order of the queue is determined by the score of each vertex. The score of a vertex A is the sum of the length of all edges on the path between A and the root vertex. The tree topology implies that there exists only one path and due to updated parent relations it is also the shortest. Additionally to the length of the path a heuristic is added to the score as well, Similar to A-Star. In this case the Euclidean Distance between A and the destination vertex is used as heuristic, but one could include additional factors. This is similar to the f-score in a A-Star search, see 2.2.
After adding the neighbors of the current node to the priority queue the 2nd iteration of the algorithm starts with the first step again. Pop the first vertex of the priority queue and consider it as the current vertex. Just as in the first iteration, but obviously a different vertex would be in first place of the queue. In this case the current vertex would be positioned on the arc of the obstacle. Next step would be to find the neighbors. In this situation the neighbors are derived differently as in the first iteration. All neighbors of a vertex lying on the arc of an obstacle are all vertices also located on the same arc. Except its parent, but it is a neighbor too. In this case the parent is the start vertex. The other candidate positions along the arc of the obstacle are calculated. Further in this work that process is referred as *populate*. It's done not more than once per obstacle. See section 3.3.3 for more. The figures, 3.4 and 3.5 show the result of the 2nd iteration including the neighborhood relation. The illustration implies that vertex 4 was first and got popped out of in the priority queue and is used as current vertex, marked in blue. Figure 3.5 shows the spanning tree of the internal graph generated by the search at the end of the interaction cycle. The neighbors in the spanning tree are not necessarily the same as the possible neighbors, shown in 3.3.3. Vertex 4 has two child-vertices, 5 and 6 which got added to the priority queue. As mentioned, the positions of 5 and 6 were calculated during the *populating* process of the obstacle. In this case they are the tangent points of the obstacle and the destination location.
In the following third iteration the same procedure is applied. Shown in figure 3.7 ver-

tex 5 was selected as the most promising according to the priority queue and used as current vertex. In the graph which is illustrated in figure 3.6, it has 3 neighbors in total. In contrast, the minimal spanning tree in figure 3.7 it has two neighbors, vertex 2, which is the destination vertex and its own parent vertex 4. Each edge of the graph has one incoming and one outgoing vertex. The words incoming and outgoing are defined considering the graph and the obstacles. A vertex is incoming if its parent is not located at the same obstacle as itself. An outgoing vertex can only have one child. In this case vertex 5 acts as an outgoing vertex. Its child, vertex 2 is now in the priority queue. Finally

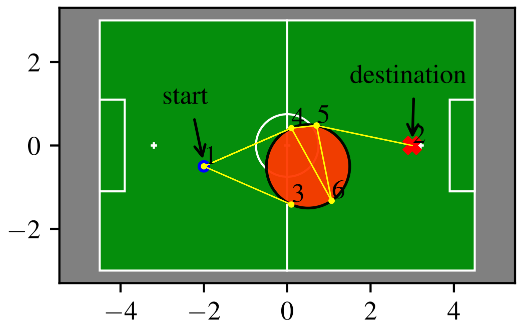


Figure 3.6: *Visualization of evolving algorithm state. End of iteration 3.*

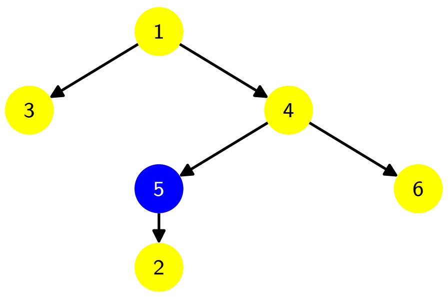


Figure 3.7: *Search tree. Showing vertex 5 as current vertex.*

the last loop uses the destination as its current vertex. As stated in the standard A-Star implementation the algorithm stops and is finished. During the algorithm the minimal spanning tree is updated and stored in the graph structure. Using that information a backtracking approach leads to the final path. Figure 3.8 illustrates the resulting path. The path consists only of straight edges and partial circular arcs. This leads to a smooth and continuously path which a robot can easily follow.

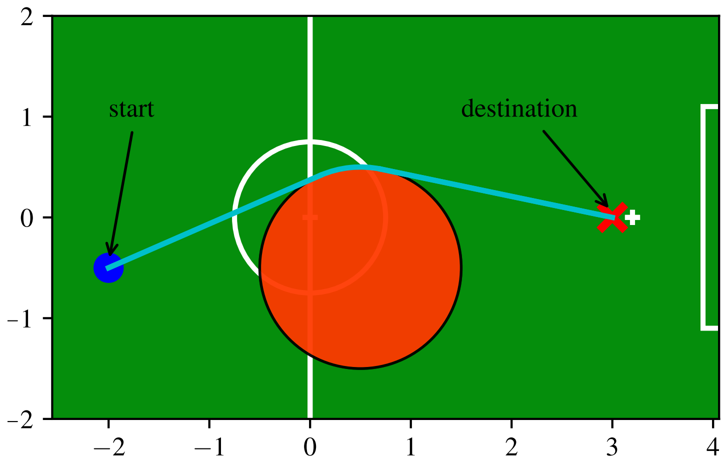


Figure 3.8: *Output trajectory, optimal path from start to destination with no corner only smooth continuous avoidance of the obstacle.*

The described approach is not even close to a productive stage. It cant handle edge cases and isn't robust at all, it needs to be improved. Later in this chapter further enhancements and clever mechanisms are introduced. With these approaches the concept will be able to handle real game scenarios.

For more flexibility and to address possible rule changes by the SPL, a requirement was set. The proposed approach needs also to handle rectangular shaped obstacles. For example the SPL rules, [Com18a], state the Illegal Defender Penalty rule which regulates the maximal number of defending robots inside a teams penalty area. The referee is encouraged not to punish a player if its intention is to leave such a blocked area. With rectangular shaped obstacles the shown approach will result in a rectangular trajectory causing the robot to rotate on spot which again results in slower movement and there increasing the time to reach its destination. Without introducing more complex obstacles a rectangular obstacle can get represented by four obstacles located at each corner of the rectangle with a visual blockage between them. This visual blockage is just a line which interferes with the inter-visibility check mentioned above. These rectangles are called blocked areas. Using an obstacle at each corner of the blacked area will round up the corners of trajectory depending on their size. The substitution of the obstacle in the given example with a blocked area is shown in the following figures. Figure 3.9 illustrates the resulting trajectory. As mentioned, introducing obstacles at each corner ensures a smooth and continuous path. Figure 3.10 displays a magnified part of the top left corner of the blocked area.
This way of handling blocked areas does not stop at rectangular shapes, polygons can easily be represented as well by creating an obstacle at each corner.

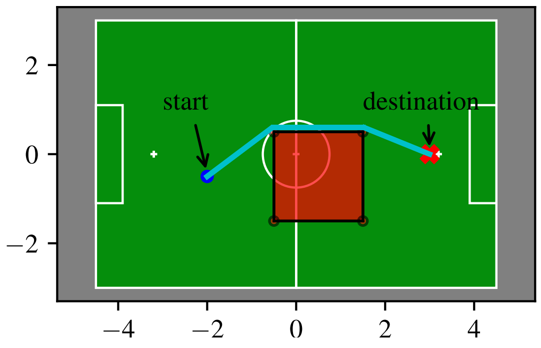


Figure 3.9: *Final trajectory with one rectangular blocked area between start and destination.*

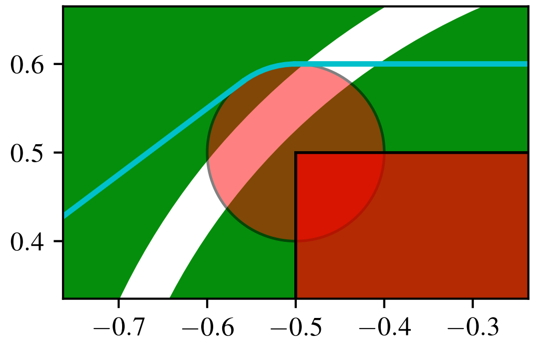


Figure 3.10: *Zoomed in figure 3.9. Showing top left corner of obstacle, trajectory gets rounded, smoothing effect.*

3.2 Data-types

Before going into more detail of the enhancements and additional functionality an overview of the used data structures and types is given.

3.2.1 Input-Vector

The input vector contains the environment of the robot as well as its own position and destination position in absolute coordinates. To keep things simple, all locations are handled as absolute coordinates, also called world coordinates. As mentioned, before the environment of the robot is a known SPL soccer field. The center point is the coordination origin, the X-axis points towards the opponent side. The system is right-handed and the Y-axis lies in the field towards the 'top' of the field. Playing direction would be left to right.
Let $C_{pos} = \{c_1, c_2, \dots c_n\}$ with $c_x = (x, y)$ be a vector containing all positions of the obstacles, $C_r = \{r_1, r_2, \dots r_n\}$ be a vector containing the associated radii, and *start*, *dst* are start and destination positions respectively. Which leads to an input vector of the following:

$$Iv = \{C_{pos}, C_r, start, dst\} \tag{3.1}$$

Since this algorithm works only on a snapshot of the surroundings as input nothing more is needed.

3.2.2 Graph

The "Graph" holds all the information that is generated and required by the search algorithm. It grows and changes during run-time and needs to hold a certain state. This can be represented easily by using object oriented approach. Using a class is a suited method to organize and incorporate related function calls which modify the data. In this subsection the focus lies on the data storage competence of the graph class and its used data types. A typical graph consists of vertices and edges. In case of the used topology, see section 3.3, the graph structure is used in a slightly different way. As shown in section 3.1, about the basic function of the algorithm, the graph is built dynamically during the search. So it will change upon usage. To decouple the search algorithm and the graph structure, the graph does not directly hold any vertices by its own. But it holds information about the edges and the environment. The environment includes obstacles, $start$ and dst locations. Edges are organized in a hash table mapping a vertex as an input to a list of connected vertices.

Given the situation of figure 3.11, looking up the hash of vertex 1 will result in a list l of vertices, $l = \{3, 4, 5, 6, 7\}$. By design the actual starting point is defined as vertex 1, the destination point is vertex 2. Further the graph needs to hold the set of obstacles, the destination and the starting position. These are needed for computing the unknown vertices requirement by the search algorithm. During initialization the graph will be fed with the needed information from the input vector Iv.

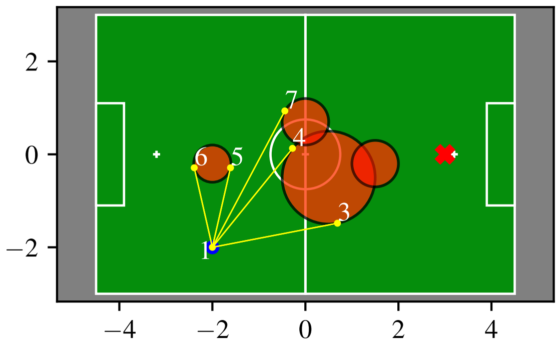


Figure 3.11: *Environment with more then one obstacle.*

3.2.3 Search Algorithm

As mentioned in the introduction of chapter 3, the search algorithm is based on an A-Star search approach and therefore has a lot in common with it. Major part of the search algorithm is the priority queue. It represents the open list of the A-Star and hold all vertices with the most promising at the top. But also additional information will be stored and used. This always depends on how the implementation handles certain parts but as mentioned in the previous section about the graph part, the search tree is not part of the graph structure and lies in context of the search algorithm. The search tree is a minimal spanning tree and can be represented as table of vertices pointing to their parent. This can be done using a hash table mapping a vertex to its parent vertex. Another table holds the corresponding g-value, similar to A-Star. The g-value is the cost to get from the root vertex to another vertex. In combination with this value, the minimal spanning tree is created by using the cost between vertices as weights. In contrast to an normal A-Start approach the necessity of holding a so called closed list is not the case. Holding all g-values in a table able to check if a vertex is already visited makes the closed list redundant. Not only holding the parent of each vertex is important, but the different type of connections. Vertices can be connected via a circular bow or a straight line, this needs to be saved as well. Otherwise the backtracking can't compose the correct path later on.

3.2.4 Output, the trajectory

The final outcome of the whole approach is a trajectory which can be executed by the robot. As described in section 3.1 the trajectory should only contain straight and partial circular arc edges. During the backtracking to find the result they get written in a result vector, p as shown in equation 3.2.

$$p = \{s_0, s_1, s_2, \dots s_n\}, \text{ with } s_x \in \{\text{line}, \text{arc}\} \tag{3.2}$$

In correct order the edges form the final result of the algorithm form the trajectory. The result vector gets pushed in the data flow of the HULKs framework for further application, for example inside a step planer module providing the next physical step of the robot.

3.3 Design

This section covers the design of the algorithm. Therefore it will split the approach in an organized manner. Implementation details are avoided and are found later in chapter 4. Starting point of the approach is the search part, subsection 3.3.2 but before that a initializing phase needs to take place, see subsection 3.3.1. The initialization handles a lot of specific edge cases. From there it will lead to the outer search part followed by the inner logic and all measures which are taken to archive a useful output in all situations. The inner logic is based around the graph, which was introduced in section 3.2.2.

3.3.1 Initialization

The environment determines the dedicated conditions of the input vector. These constrains have a large impact on the process of finding a optimal path. This happens during the initialization of the graph. The graph is responsible to feed the search with meaningful data as well as maintain it. As mentioned the search behaves alike an A-Star search, it needs information of costs between vertices as well as the neighborhood of each vertex. These facts are provided by the graph. To initialize, the graph needs the input vector. Before using the input vector, the information is filtered. The list of obstacles, including their radius can be pretty long specially considering multiple blocked areas and all robots on the field. This list can be narrowed down to relevant obstacles, obstacles which are really used during the algorithm. First one can argue that obstacles inside of others can be neglected. Because the robot will not go inside of any obstacle. Therefore only fully overlapped obstacles should be ignored. To check if circles are inside of each other a sweep line algorithm is used. The remaining obstacles only exist inside the graph. Lets call them extended obstacles.

3.3.1.1 Extended Obstacle

Extended obstacles function as a replacement for real outside obstacles. Compared to the real world obstacles they hold additional information such as, blocked arcs or candidate vertices. The blocked arcs describe the parts of the circular arc which are overlapping other obstacles. In figure 3.12 the blocked arc of the largest obstacle are shown. The two each smaller obstacle overlaps a certain part of the circular arc. This part is not a valid position for any vertices. The later calculated candidate vertices, as well as their edge which lie inside these blocked arcs will be ignored and thrown away. The mentioned candidate vertices hold the tangent point at each obstacle. The tangent

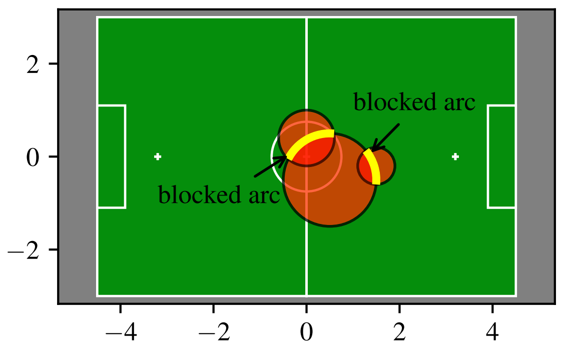


Figure 3.12: *Example of overlapping obstacles. The resulting blocked arcs of the largest obstacle are marked in yellow.*

points belong to all visible inner and outer tangents between all other obstacles. This is done to reduce multiple similar calculations, later more. The candidate vertices as well as the blocked arc field are initialized as empty lists. Extended obstacles should represent the real world obstacles inside the graph but may be modified. The modification is needed to handles several edge cases. Let A be an obstacle with an area which includes the start vertex. This situation will result in an instantaneously termination of the algorithm and no meaningful output. That maybe expected but in case of a soccer game will lead to a standing robot, which is not good. These kind of situations need to

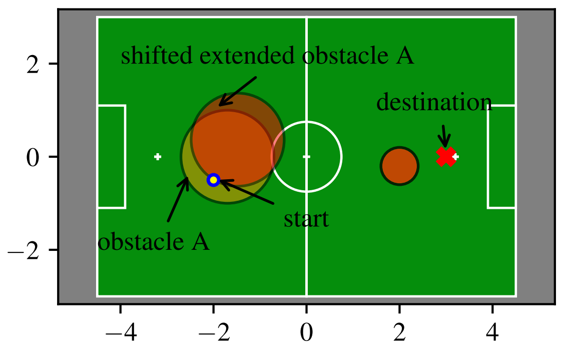


Figure 3.13: *Example of shifted obstacle. Translation of the obstacle in direction of closest circle boundary.*

get handled in a way which do not harm or lead to any game disadvantage. They can be detected before the algorithm starts. One approach could be to ignore the coursing obstacle completely. But this will lead to no obstacle avoidance at all and potential penalties calls by the referee. Another concept could be to move straight away from the center of the obstacle until the outer obstacle edge is reached and then start the path planning. This will lead to a major detour and hurt the game-play. A trade-off approach is to move the obstacle instead. Moving the obstacle straight away from the start vertex seems promising. The shift distance needs to be just long enough to let the start vertex be a little outside of the obstacle. This shifting during initialization, results in a reasonable trajectory as well as not sacrificing the important obstacle avoidance. Figure 3.13 shows such a situation including the original obstacle and its shifted extended version. This approach introduces an additional edge case. If the shift process moves the obstacle in a way that it will include the position of the destination one needs to adapt the direction of the shift, one possible solution can be found in chapter 4. This approach is used similar to handle the destination vertex as well.

3.3.1.2 Blocked Areas

At the end of section 3.1, blocked ares got introduced. During initialization they should not be handled the same way as the extended obstacles. In case a blocked area includes start or destination vertices, different things need to be done. Assuming blocked areas in real game situations, blocked areas cause a lot of different edge cases such as: the start vertex lies inside of a blocked area and the destination not. The destination vertex is inside but the start not. Both, start and destination are inside different blocked areas. And start and destination vertices are inside the same blocked area. These need be handles in a reasonable manner regarding the game. If the start and destination vertices are located in different blocked areas, one can handle each on its on, independent. Therefore the mentioned situations boil down to only three handling approaches: Start inside of blocked area, destination inside of blocked area, both are inside of the same blocked area. The following handling methods include tactic decisions made by and for the Hulks team behavior and may not be suitable for everyone.

Blocked areas are needed to represent certain game rules. For example the illegal defender rule. In case of the start vertex lying inside a blocked area, it makes sense to try to get out as fast as possible. Due to the fact that the referee is encouraged to not call any penalty if the robot's intention is to leave the area straight away. The behavior in such a situation is to plan the path to route through the closest blocking segment. Figure 3.14 shows the final trajectory of such a situation. In that case the left edge of the rectangular blocked area gets removed. This approach leads to a nice trajectory. This deletion of blocked lines is done during the graph initialization.

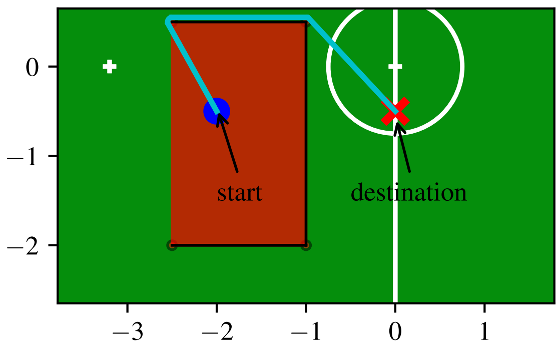


Figure 3.14: *Example of start vertex inside of blocked area.*

In case the destination lies inside of a blocked area, a different approach need to be used. The same approach using deletion of blocked line can result in a case where the final trajectory routes deeper inside of the blocked area. Shown in figure 3.15.

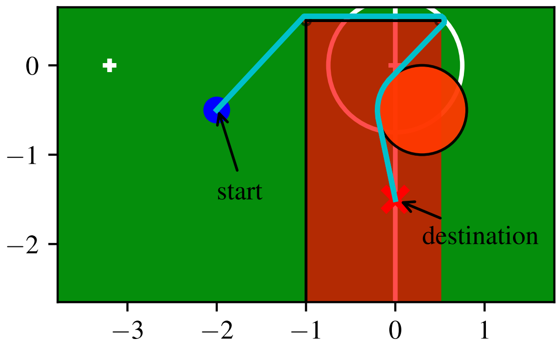


Figure 3.15: *Example of destination vertex inside of a blocked area with a obstacles using the deletion approach.*

The behavior shown in fig. 3.15 is very bad considering the potential penalties calls and the whole goal of this path planning approach. To get around this problem, the deletion approach needs tweaked and several cases need to be caught. That would add a lot of complexity implementation wise. A much simpler way was developed. Shifting the closest blocked line instead of deleting it forces the path to keep from going deeper inside. Keeping the destination as the deepest of path going inside of the blocked area. The result is shown in figure 3.16.

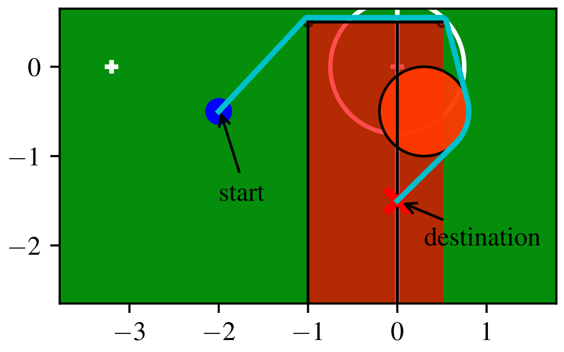


Figure 3.16: *Example of destination vertex inside of a blocked area with a obstacles using the shifting approach.*

All of this needs to take place during the initialization of the graph.

3.3.1.3 Initialize Search

Initializing the search part of the algorithm is quite similar to the initialization process of a A-Star search. It needs several data structures holding the current state of the search algorithm, same as an A-Start search. Additional it needs an initialized instance of the graph introduced before as well as a structure to hold the edge types to reconstruct the correct trajectory. As shown in section 3.2.3. All fields are empty but the graph and the queue. As mentioned above in section 3.3.1, the graph is initialized. The queue just holds the start vertex.

3.3.2 The Search Loop

Please take note of section 2.2, before reading this section. After the initialization the search loop starts. In the end the search has a ordered list, the *open* queue which holds most promising vertices at first position. Every loop cycle the most promising vertex, referred as *current* is checked if it is the destination vertex. If not it will take the *current* vertex neighbors and will add them to *open* queue, if they are not already checked. The whole search is illustrated with pseudo code at algorithm 3 on page 34.

Algorithm 3 Plan the path, modified A-Star

```
1: function FINDPATH($Iv$)          $\rightarrow$ only input is the input vector $Iv$
2:     # initialization
3:     $dest$ := $Iv$.destination
4:     $graph$ := Graph($Iv$)          $\rightarrow$ $graph$ gets initialized, see sec. 3.3.1
5:     $open$ := PriorityQueue()
6:     $open$.put(0, $start$)          $\rightarrow$ put $start$ vertex with priority 0 inside $queue$
7:     while not open.empty() do          $\rightarrow$ main loop start
8:         $current \leftarrow open$.get()          $\rightarrow$ vertex with smallest f-score is $current$
9:         if $current \equiv dest$ then
10:            return "Finished"          $\rightarrow$ destination found
11:        end if
12:        for all $neighbor \in graph$.neighbors($current$) do          $\rightarrow$ see sec. 3.3.3
13:            $new\_g\_score$ := $g\_score[current]$ + cost($current$, $neighbor$)
14:            if ($neighbor$ not visited) or ($new\_g\_score < g\_score[neighbor]$) then
15:                $g\_score[neighbor] = new\_g$
16:                $parent[neighbor] = current$
17:                $f\_score = new\_g$ + heuristic($neighbor$, $dest$)
18:                # Normal A-Star would now add the $neighbor$ to its open list
19:                ~~$open$.put($f\_score$, $neighbor$)~~
20:                # try to optimize.
21:                SkipIfPossible()
22:            end if
23:        end for
24:    end while
25: end function
```

The first few lines show the initialization, described in section 3.3.1. At line 8 the main loop starts which will cycle each vertex in the *open* list. If the destination vertex *dest* is reached it will terminate instantaneously with a return "finished" call in line 10. Otherwise it will grab all neighbors of the *current* vertex using the *graph* instance in row 12 and cycle through all of them. For each of the neighbors a g-value is calculated, line 13. It consists of the g-value of its parent, the *current* vertex, summed with the cost of getting from *current* to itself.
The functionality of the cost function is separately shown in algorithm 4. It needs to consider the type of edge between the vertices u and v. Therefore a simple euclidean distance calculation is not enough. It needs to be distinguished which edge type needs to be considered to calculate the correct cost. For detailed information see section 4.2.1 which some implementation details.

Algorithm 4 The Cost Function

```
1: function COST(u: Vertex, v: Vertex)
2:     if edge(v, u) ≡ "straight" then
3:         # return the euclidean distance
4:     else if edge(v, u) ≡ "CW" then
5:         # return length of partial arc CW
6:     else if edge(v, u) ≡ "CWW" then
7:         # return length of partial arc CCW
8:     end if
9: end function
```

Back to the search, algorithm 3. The search will only add vertices to the *open* list if they are unknown, or more specifically not visited, and update them if new calculated cost to each of them is lower than before. The lines 14 $\rightarrow$ 17 show that procedure including updating all maps. Up to this line its similar to a normal A-Star graph search except of creating the graph during run time. If one would use a normal A-Star approach, line 19 would the vertex the *open* list and then handle the next neighbor. Analyzing the structure of the resulting graph and the way how neighbors are calculated, optimize the cycle number of the search algorithm is possible.
Algorithm 5 shows the optimization. Instead inserting the neighbor vertex to the *open* list, in some cases the neighbors pair vertex will get put inside of it. This rapidly decreases the number of search cycles.
First the neighbors vertex gets assigned to a local reference, *pair*, to create a better

Algorithm 5 SkipIfPossible, Can the pair vertex be added.

```
Require: Namespace of FindPath()
 1: function SKIPIFPOSSIBLE
 2:     pair := pair of neighbor
 3:     if pair ≡ current then          → todowas ist der pair vertex vom neighbor?
 4:         open.put(f_score, neighbor)
 5:     elsepair is a legit candidate
 6:         new_g_score := g_score[neighbor] + cost(neighbor, pair)
 7:         if (pair not visited) or (new_g_score < g_score[pair]) then
 8:             g_score[pair] = new_g
 9:             parent[pair] = neighbor
10:             f_score = new_g + heuristic(pair, dest)
11:             open.put(f_score, pair)
12:         end if
13:     end if
14: end function
```

overview. Pair vertex is the same as current vertex implies that the neighbor vertex is an endpoint of an edge which results in a normal A-Star behavior. In line 3 and 4 check and represent that. But if the neighbor vertex is an starting point of an edge it will only introduce one new possible unknown vertex, the corresponding endpoint of the edge. If the pair vertex is not known or can be updated, line 7, then the graph gets updated. Finally the pair vertex is put inside of the *open* list. This optimization results in a much lower search cycle number, later shown in section 3.5.

3.3.3 Graph - Exploration of neighbors

This section shows how the underlying graph feeds the search algorithm with the needed data. The graph structure has on obvious appearance in the search loop, see algorithm 3. The method *neighbors* is called in line 12 and returns a list of all neighbors of a given vertex. Also in the cost function the graph needs to be used because only the graph has the knowledge and therefore the capability of computing the correct cost. In this way the graph structure is totally independent from the search algorithm. This implies that the *cost* function used in line 6 of algorithm 5 needs to be handled by the graph structure too. If the search algorithm needs the neighbors of a given vertex the graph will calculate the needed vertices and return them inside of its *neighbors* method. The functionality of the *neighbors* method depends on the type of the given vertices.

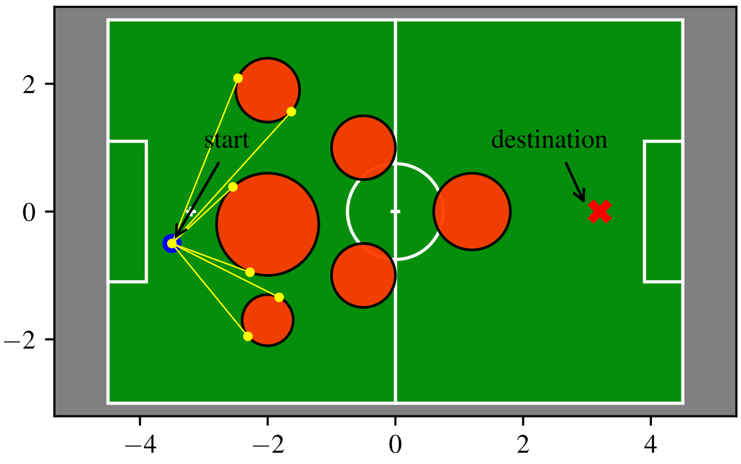


Figure 3.17: *Complex environment.*

The neighbors of the given vertex can already be known, in case the search calls *neighbors* with the same given vertex multiple times. The graph holds all prior calculated edges. Therefore it can just return them without any calculations. In case the required neighbors are not known, the graph needs to calculate their position. The calculation differs as mentioned above. It depends on the given vertex and its position. The simple case is when the given vertex is not at an obstacle, therefore lies not on an arc of an obstacle. This case handles the first cycle used of the shown approach. The preprocessing shown on figure 3.13 and in the corresponding section 3.3.1 makes sure that the start vertex is always not on an arc. Creating a list of neighbors of the start vertex is pretty easy and is already touched in section 3.1 and 3.2.2. First of all the direct path to the destination vertex should be checked. If start and destination are connected right away then no search needs to take place and the situation can be solved with a trivial solution. Connected means if there is a line of sight between start and destination, which is not interfering with any other obstacle. The list of neighbors would then only contain the destination vertex, all other possible neighbor vertices get neglected for optimization. Ignoring does not lead to an unwanted and incorrect behavior. In case the direct path is blocked all neighbors need to be computed. In case of handling the start vertex, that would be all visible tangent points at all obstacles of a line starting from the start vertex. Figure 3.17 shows a good example of a scene with a complex environmental conditions. The other situation is if the given vertex is an indeterminate location on the path. The visibility approach implies that all intermediate vertices are located at an obstacle. In that case all neighbors are only found at the sane obstacles as the given vertex as well as one single remote point. The given vertex was calculated using a tangent. The remote point is the vertex at the opposite side of the tangent edge. The other vertices, at the same obstacle, are found by calculating all tangents to all other obstacles. For more consult section 2.1. Only visible tangents are valid and the corresponding points are added to the list of neighbors. Figure 3.18 illustrates such a intermediate vertex and its

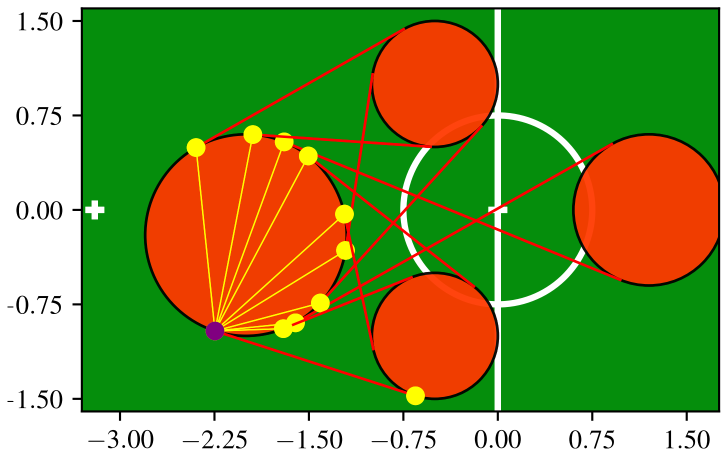


Figure 3.18: *Four obstacles. Intermediate vertex shown in purple and it's neighbors in yellow as well as the corresponding tangent edges in red.*

neighbors. The example has a slightly different setup as in figure 3.17. The top most as well as the lowest obstacle were deleted, but the rest is similar. The four remaining obstacles are shown. The purple dot represents the given vertex. The yellow dots are the valid neighbors and the red lines show their corresponding tangent edge. The other end of each tangent edge is the pair point of the neighbor. As mentioned all neighbors

except of one are located at the same obstacle as the given vertex. The highlighted red tangent edge connects the given vertex with its pair at the other obstacle. In this scenario the *neighbors* call of the graph, given the purple vertex, will return all yellow marked vertices with with their true costs.
Combining these cases into one single function could lead to a structure similar to algorithm 6. The functionality is toughed in the following paragraph.

Algorithm 6 *neighbors*, returning neighbors of a given vertex.

```
Require: Initialized graph
function NEIGHBORS($v$: Vertex)
    neighbors:= empty list
    if $v$ not an intermediate vertex then          → not at an obstacle
        if destination vertex is visible then
            append dest to neighbors
            return neighbors
        end if
        for all $obs \in obstacles$ do
            calculate tangents to obs from $v$
            if tangent points are valid then
                append tangent point to neighbors
            end if
        end for
    else          → $v$ is an intermediate vertex, located at an obstacle
        $obs_v$ := obstacle at which $v$ lies
        if $obs_v$ is was not populated yet then
            populate $obs_v$          → calculate all candidate vertices, see algo. 7
        end if
        for all $candid \in candididates$ do
            if candid is reachable then
                append candid to neighbors
            end if
        end for
        append pair vertex of $v$ to neighbors
    end if
    return neighbors
end function
```

Algorithm 7 *populate* obstacle, create the candidate vertices at a specific obstacle.

```
1: function POPULATE($o$: Extended Obstacle)
2:     find blocked arcs of $o$
3:     compute tangent points between destination and $o$
4:     if tangent points are visible then
5:         add to candidate list of $o$
6:     end if
7:     for all $obs \in obstacles \setminus o$ do          $\rightarrow$ all obstacles except of $o$
8:         compute tangent points between $o$ and $obs$
9:         if tangent points are visible then
10:            add to candidate list of $o$
11:        end if
12:    end for
13: end function
```

This paragraph will clarify the basic functionality of Algorithm 6 and explain the very important if clauses in line 10 and 18. These two lines determine if a point gets added as a vertex to the list of neighbors. As mentioned above the procedure differs if the given vertex v is at an intermediate location or not.
The first part from 4 $\rightarrow$ 13 handles the case when v is not located at an obstacle. The first if condition in line 4 handles a direct line of sight optimization. If the destination is reachable, it will return early without any computation wasted for calculating the other neighbors. This is only a valid optimization if the strict separation between graph structure and the superior search algorithm is neglected. The for-loop in line 8 $\rightarrow$ 12 handles the other neighbors. The statement in line 9 provides the 2 tangent edges and their 2 corresponding tangent points between v and the current obs. For each point the if-clause in line 10 checks if it is valid. The term valid includes the visibility constrain as well as the check if the tangent point is not inside of a blocked arc. Checking these constrains is simpler than it seems. The visibility constrain includes the blocked arc constrain. The visibility check is computational expensive but there is no way around it, see section 3.3.4.
The second part from line 15 $\rightarrow$ 22 handle intermediate locations of vertex v. The obstacle at v is referred as obs_v. If obstacle obs_v is not populated yet it gets populated. To *populate* an obstacle means to compute all candidate vertices at that obstacle. This is shown in algorithm 7 and done in a straight forward way. All candidate vertices are determined by tangent edges between obs_v and all other obstacles obs. Additionally the destination vertex is used to create two tangents to the obstacle obs. All tangent edges are put through a visibility check and then added to the list of candidate at obstacle obs. In line 20 the list is used to determine all reachable candidates. Reachable in such a way that no other obstacle is blocking the connection on the arc.

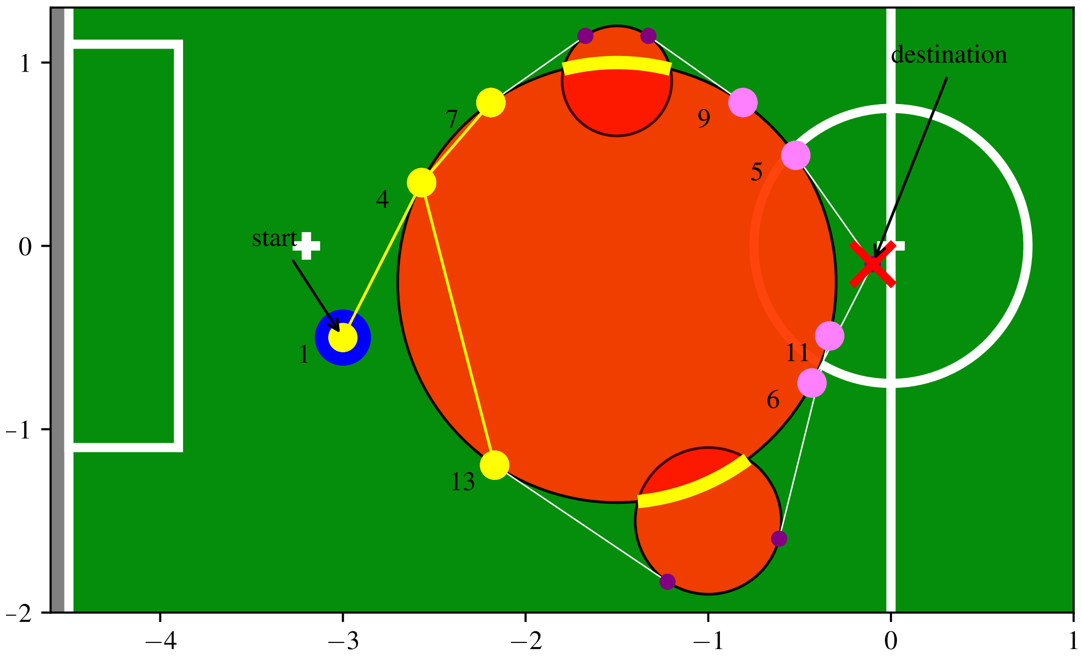


Figure 3.19: *Neighborhood of vertex* 4 *represented in yellow as well as the blocking arcs of the big obstacle. Candidate vertices shown in magenta with their corresponding tangent edges and pair vertices illustrated with white lines and purple dots*

Figure 3.19 illustrates an example showing the neighborhood of vertex 4 in yellow and the candidate vertices of the large obstacle in magenta. The blocked parts of the arc are highlighted in yellow as well. Line 24 of algorithm 6 then add the pair vertex also to the returns neighborhood. That all should handle all cases of neighbor exploration and the algorithm 6 is explained.
Now the graph structure is capable of feeding the search algorithm with reasonable data.

3.3.4 Visibility check

This sub section deals with the mentioned visibility check. Two vertices are intervisible on if the connecting line segments does not intersect with any obstacle of edge of an blocked area. There exists very efficient methods to find intersection point of line segments by sorting them in a smart way. These methods are based on the sweep technique. This procedure is often used in the algorithmic geometry. Such an algorithm is also called a sweep algorithm. The core of a sweep is the sweep line. It is moved through the entire space until all objects of the problem have been processed. A data structure is used to store the objects touched by the sweep line. Such a data structure is then called a sweep status structure. In general, a sweep converts an n-dimensional

static problem into an $(n-1)$-dimensional dynamic problem.
Unfortunately the introduced approach, creating the graph on the fly can not be tackled with that kind of method. Due to limiting the search space by not computing the whole graph before hand.
The drawback is the necessity of checking every edge after computing it on visibility. This is done in a straight forward manner. Thinking of a line segment in a space full of obstacles. The line segment connects two points and represents the line of sight from one to the other. If this line intersects with an obstacle the line of sight is blocked and therefore the two points are not visible to each other. The environment consist of obstacle as well as blocking lines. This brings the problem down to a line segment and circle intersection problem.
Let v and w be two points in a two dimensional space and A the resulting line segment between them. Goal is to find out if these two points are visible to one another. The obstacles are circles $C = \{c_1, c_2, \dots c_n\}$ containing position and radius as well as $Ls = \{Ls_1, Ls_2, \dots Ls_n\}$ containing the other blocking line segments.

Theorem 1. *Let I_c be the set of intersection points of circle c and line segment A. I_C can contain none,* 1 *or at most* 2 *points of intersection. Only if I_c contains* 2 *points an intersection occurred.*

The theorem 1 is based on the special use case while checking the visibility of two points. In case the line of sight only touches the check circle I_c will contain 1 point of intersection but the visibility still holds.
Next will be the line segments. This is done in a more simple manner. Due to the fact that all end point of the blocking line segments are inside of other obstacles no special cases of touching end points needs be handled. But the intersection detection only needs to trigger if the line of sight crosses the blocking line segment. Touching is fine and helps to avoid unnecessary fails during path planning.

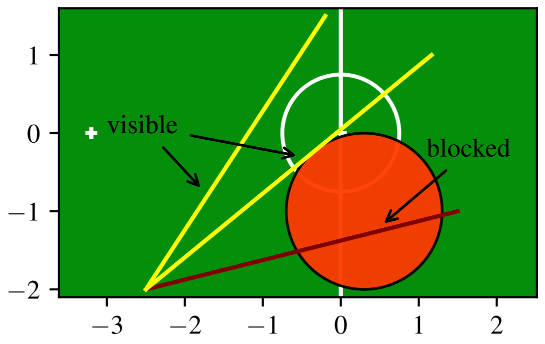


Figure 3.20: *Toughing edges are considered as visible.*

3.4 Correctness

The shown approach needs to perform correctly, it can not be used in the stated scenario if it is not robust. As stated in [DB82] functional correctness can be checked in a straight forward manner. For the introduced algorithm all possible input vectors need to result in a reasonable output and therefore must produce usable trajectories. To distinguish between correct and incorrect or useful or not, we need to state pre-conditions and the corresponding post-condition. To simplify this process, the approach will be divided into sequential parts. Each part will then be checked separately. If all parts fit in to each other, the whole algorithm will produce the wanted output. If all slices are functional correct the state transitions between them are robust. The following subsections will examine logical parts of the approach. Beginning with the initialization phase, the search algorithm as well as the underlining graph logic needs to be checked.

3.4.1 Initialization

First of all the initialization is going to be inspected. The initialization phase prepares the input vector and modifies the given environmental state to be suitable for the path planning approach. The preconditions are simple. The input vector needs to provide valid data. As mentioned in section 3.2.1, the input vector roughly consists of a lists of circles, coordinated of locations and line segments. This data fields get filled outside of the approach. In case of the HULKs framework different perception modules produce these information and the path planning state a dependency.
The post condition should be a modified data set on which the rest of the approach works. As shown in section 3.3.1 all cases are handled in a reasonable way and the intended output gets generated.

3.4.2 The Search, modified A-Star

As shown, the solid A-Star search got modified in a way to add more vertices to the open list as in the traditional implementation shown in section 2.2. The introduced modification just adds functionality on top of the traditional approach. It just adds additional vertices to the open list and therefore does not touch any properties of the A-Star, regarding correctness or completeness. Resulting in an optimal solution is also still guaranteed. The correctness of the A-Star search algorithm depends on the used heuristic. It needs to be at least admissible, which implies the property of underestimating or predict exactly the real costs, as mentioned in [BCKO08]. The consistent property can be neglected. So that the heuristic needs to fulfill the following equition, eq. 3.3. Let $g(x,y)$ be the real cost and $h(x,y)$ the heuristic to location y starting from location x.

Then the following must hold:

$$g(x) <= h(x), \text{ with } x, y \in \{\text{all locations}\} \tag{3.3}$$

To accomplish a directed search, the euclidean distance is used as heuristic in the introduced approach. This is widely adapted and used in many navigation problem solved with A-Star. Alternatives like the Manhattan distance or Diagonal distance make only sense while using a grid type sampled environment. The used representation of the world is accurate and does not use any sample technique which will lead to grid-like world coordinates. Therefore, the euclidean distance is a promising heuristic. Another way would be to design a special adapted heuristic. This could be further work to introduce even more robot related information like a rotation penalty. As mentioned, the properties of the heuristic control the A-Star search behavior. As the Euclidean distance holds the required specifications the search part is correct.

3.4.3 The Graph

The graph part, explores and creates the graph on which search works on. The introduced method creating the graph in section 7 and 6 grantees the correct graph. Worst case would be that the whole graph needs to be generated and the maximal run-time occurs. Using the whole graph for the search to work on, will result in a complete and correct outcome. Therefore only the step by step creation needs to be checked more precisely. For each step all pre- and post-conditions are presented in the related sections. All possible inputs result in the expected output and can be used to accomplish the goal of finding the optimal path. As mentioned above the heuristic controls the behavior of the whole approach. The heuristic and the cost calculation can be both handled by the graph. Therefore, the graph has impact on the ordering inside of the search open list and needs to work correctly. This is shown in the related sections as well.

3.5 Complexity

The complexity analysis is based on the implemented prototype code. Let n be the number of obstacles including the added ones of the blocked areas. The initialization has a linear proportional complexity of $\mathcal{O}(n)$. During run-time, the tangent calculation adds $\mathcal{O}(n!)$ and the expensive visibility checks additional $\mathcal{O}(n^2)$. Due to the directed search, the complexity of each run can be significant lower, only the worst situations will lead to a large number of iterations. A statistically analysis shows promising results. 2500 different situations were generated and investigated. To restrict the random generated situations to be more realistic following constrains were used. The start and the desired destination location are more than 2.5m apart of each other. A total number of 9 obstacle with a radius of 0.3m are place inside of the field dimensions with at least 0.3m distance.

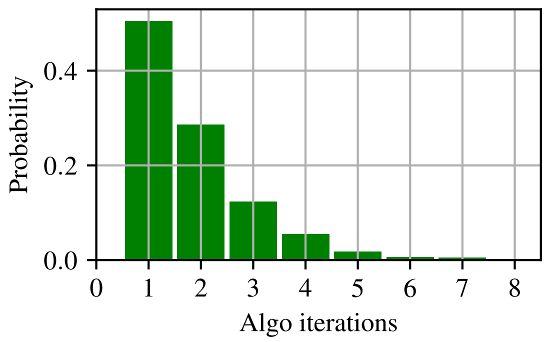


Figure 3.21: *Histogram, showing distribution of algorithm iterations of all* 2500 *simulated situations.*

The observed statistics are shown in table 5.1 and 3.1. This part will focus on the number of function calls. A list of often called function is shown in table 3.1. The search algorithm needs only 1.9 iterations to find the path to the destination. That number seems pretty small. Looking at figure 3.21 shows a histogram of the iterations regarding their occurrence probability. Nearly the half of the situations result in a trivial problem, with no obstacle blocking the straight connection.
The table 3.1 shows mean of how often the specific function was called in each situation.

Table 3.1: *Observed number of function calls, mean over all 2500 runs.*

Function Calls/Per situation

1.8919	Search Algo. Iterations
18.2502	connected() calls.
0.7774	populate() calls.
34.4595	visible() calls.

Chapter 4: Implementation

This chapter describes parts of the code which can be found at `https://github.com/fwarmuth/tangentgraph`. It is not research related in any mean, only for documentation purpose. The list below gives a overview how the prototype implementation of the approach introduced in chapter 3 took place. It is not a row by row explanation! It is a selection of problems encountered during development. The following list gives an overview of which parts and functions are explained in detail.

List of Code

1 Cost Function 47
2 Connected Function 48

4.1 Development Framework and Visualization

Before starting the development of the proposed path planning approach a development framework was built. Due to the fact that debugging software running on a remote machine especially with very limited or close to no visual output is difficult and leads to slow progress. The normal development framework of the HULKs include different self-made debug options but introduce to much overhead for a proof of concept implementation. All development of the HULKs for the NAO is done in CPP with as less as libaries as possible. CPP introduce a over head regarding graphical output. Therefore a quick look beyond the end of the nose revealed Python as a fast and easy way to bring some graphics to the screen. With Python it is fairly easy to create graphical output in a just a view lines of code.
Initially Pygame was used to visualize the progress of the development. Later Matplotlib was used to create the nice figures in this work.

4.1.1 Pygame and Matplotlib

Pygame is an easy to use Python library for making graphical applications. Built on top of the SDL, Simple DirectMedia Layer is a popular cross-platform development library. It was designed to provide easy access to input devices and to create graphics with OpenGL. With help of a simple collection of wrapper functions the development could begin.
Later On, a second interface was created. This time Mathplotlib was used, the MATLAB-like plotting library can produces quality figures. But it introduced some problems as specially the different widths of lines were problematic. Matplotlib sets the width of lines in absolute pixels and not relative to the coordinate system. Therefore a 2 pixel width line zoomed in is still 2 pixel wide. To overcome this problem a simple redraw function is called depending on the zoom/size of the coordinate system. All figures in this work were done using Matplotlib.

4.1.2 Networkx as a graph visualizer

During development, a graph visualizer was needed to illustrate the internal graph. Implementing such thing from scratch is not needed. Networkx is a free Python library regarding networks and graph-theory. It is very efficient and scales well. Only one features was used, in combination with Matplotlib Networkx can plot networks. Figure 3.3, 3.5 and 3.7 are done using it. Also heavy debugging took place with this method to visualize different graphs.

4.2 Python Prototype Code Snippets

This list shows selected implementation details and special function calls. They are taken out of context and is not runable by copy and paste. The following code is shortened, error handling as wells some comments are removed compared to the production code. This is necessary still be describable. The goal of this chapter is to show how the edge case handling is done. Handling edge cases turned out to be very time consuming and at least 30% of the total time was wasted here.

4.2.1 cost(u: Vertex, v: Vertex, edge_type: EdgeType)

The cost function resides inside the graph structure so it can work with the environment properly. The cost between two vertices is the based on the euclidean distance between them. The function expects two vertices, *u* and *v* as well an edge type. If both vertices are located at the same obstacle, the direct connection is blocked. Therefore the edge type between them is not straight. This switch case is handled in row number 2. Returning the simple case in row 3. The challenge is computing the real cost between two points located on the arc of the obstacle. Its necessary to check if the two vertices are not located at the same coordinates. For example 3 obstacles have tangent points in a straight line. To avoid multiple paths with the same costs, the intermediate stops get a small penalty shown in row 10. The direction of the circular edge is determined by the connected function, see section 4.2.2. The rest is solved using basic geometry.

Code Snippet 1 Cost Function

```
1  def cost(self, u: Vertex, v: Vertex, edge_type: EdgeType):
2      if edge_type == EdgeType.STRAIGHT:
3          return norm(u.location - v.location)
4
5      # Give values better names
6      obstacle = self.obstacles[v.obstacle_id]
7      u_angle = m.angle_at_circle(u.location, obstacle.position)
8      v_angle = m.angle_at_circle(v.location, obstacle.position)
9      if u_angle == v_angle:
10         return np.finfo(float).eps
11
12     if edge_type == EdgeType.CCW:
13         if u_angle < v_angle:
14             return obstacle.radius * (v_angle - u_angle)
15         else:
16             return 2 * np.pi * obstacle.radius - obstacle.radius * (u_angle - v_angle)
17     elif edge_type == EdgeType.CW:
18         if u_angle < v_angle:
19             return 2 * np.pi * obstacle.radius - obstacle.radius * (v_angle - u_angle)
20         else:
21             return obstacle.radius * (u_angle - v_angle)
22
23     raise RuntimeError("Unknown edge type!")
```

4.2.2 connected(u: Vertex, v: Vertex)

The connected function tackles the reachability constrain described and shown in figure 3.19 on page 40. The goal is to find an arc on the perimeter of the obstacle which is not blocked by anything. It expects two vertices located at the same obstacle and returns a tuple consisting of a connected flag and a direction indicator.
An obstacle can have multiple blocked arcs and therefore an iteration over the whole set of arcs is necessary. The function is designed in a typical manner, starting with the assumption that both arcs between the vertices are free and usable, than checking each blocked arc. First the angle of each vertex regarding the X-Axis and the center of the obstacle is calculated, as shown in row 7 and 8. First step of the loop is to try to return

Figure 4.1: *List of possible scenarios. Showing the value range and their relative value of the angles and the reasoning of each case.*

```
   -PI                 0                   PI
1 - |----V----U----S::::::::E----(V---U)----|, U -> V is blocked (CCW)
2 - |----U----V----S::::::::E----(U---V)----|, V -> U is blocked (CW)
3 - |-------V------S::::::::E-------U-------|, V -> U is blocked (CW)
4 - |-------U------S::::::::E-------V-------|, U -> V is blocked (CCW)
5 - |::::E---------U--------V----------S::::|, V -> U is blocked (CW)
6 - |::::E---------V--------U----------S::::|, V -> U is blocked (CCW)
```

early, without iterating over all blocked arc. In case any of the two vertices lie inside a blocked arc or both directions are blocked the function returns, see row 13 to 15. Than a large switch case covers all situations. The vertex angles as well as the arc start and end lie in an interval of $[-pi, pi)$ by definition. Let `V` and `U` be the angles of the two vertices. `S::::E` represents the range of the blocked arc, from `S` to `E`. As mentioned all values can only be in $[-pi, pi)$, carrying these values on a scale using a simple code style results in figure 4.1. Illustrating all possible scenarios. This cases get handled in row 17 to 30. The last part of the code, starting at row 32, constructs the desired output form and in case of no blockage selects the shortest way around the obstacle.

Code Snippet 2 Connected Function

```python
def connected(self, u: Vertex, v: Vertex):
    # math_negative and math_positive hold the state of
    # the both possible ways along the circlular egde (arc)
    math_negative = True
    math_positive = True

    obstacle = self.obstacles[u.obstacle_id]
    angle_v1 = m.angle_at_circle(u.location, obstacle.position)
    angle_v2 = m.angle_at_circle(v.location, obstacle.position)

    if obstacle.blocked_arcs:
        for arc in obstacle.blocked_arcs:
            if m.point_inside_arc(u.location, arc) or m.point_inside_arc(v.location, arc)\
                    or (not math_negative and not math_positive):
                return False, None

            if arc.start_angle < arc.end_angle:
                if angle_u < angle_v < arc.start_angle or arc.end_angle < angle_u < angle_v:
                    math_negative = False
                elif angle_v < angle_u < arc.start_angle or arc.end_angle < angle_v < angle_u:
                    math_positive = False
                elif angle_u < arc.start_angle and arc.end_angle < angle_v:
                    math_positive = False
                elif angle_v < arc.start_angle and arc.end_angle < angle_u:
                    math_negative = False
            else:
                if angle_u < angle_v:
                    math_negative = False
                else:
                    math_positive = False

    if math_positive or math_negative:
        delta = np.arctan2(np.sin(angle_v - angle_u), np.cos(angle_v - angle_u))

        if delta >= 0:
            if math_positive:
                return math_positive, EdgeType.CCW
            else:
                return math_negative, EdgeType.CW
        else:
            if math_negative:
                return math_negative, EdgeType.CW
            else:
                return math_positive, EdgeType.CCW
    else:
        return False, None
```

Chapter 5: Evaluation

Evaluating path planning approaches is challenging. As shown in the introduction section 1.4, different strategies have been proposed in the last decades. Each uses a different approaches to face the path planning problem. Comparing the resulting path can be an approach to evaluate different strategies, by taking a closer look, it turns out not to be. The performance term can only be concise regarding a specific application. Some methods are very quick run-time wise but will never find the optimal path. Some need a lot of computational iterations and are hard to implement. But most important are the requirements specific to the application in which the approach is used. In case of RoboCup competition, running against time and avoid penalty rules.
The testing section will compare selected approaches with the approach proposed in chapter 3 regarding the challenging requirements in a SPL match.

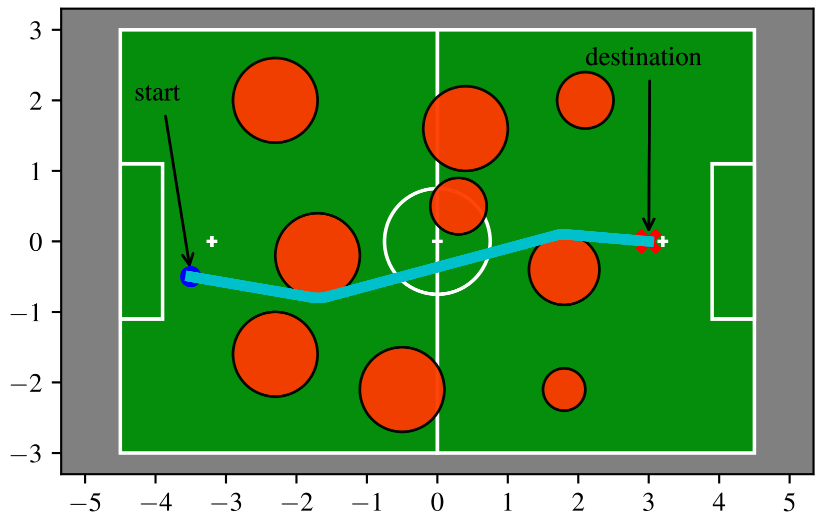


Figure 5.1: *Showing the only resulting path using the proposed approach applied on the case study.*

5.1 Comparison

The here proposed path planning approach is discussed in the light of the given RoboCup SPL competition application. During the match, a robot needs to move quick and avoid collision with obstacles. The specific requirements are proposed and listed in section 1.5. And can be summarized in the following major aspects:

- Short Path - no unnecessary detours, optimal path in distance and time.
- Collision Avoidance - minimize chance of collisions/penalties.
- Smooth Trajectory - resulting trajectories should be easily to follow, considering the robot capabilities.

In this section the introduced existing approaches, see section 1.3, are compared based on a case study. Many different situations can occur during a soccer match. The case study is reduced to a single situation. Obviously one single case can not prove much and cannot represent all challenges in path planning, but it can help to highlight and clarify drawbacks and benefits of each method. A well defined standard situation is always necessary when comparing behavior and should be constant. Figure 5.1 shows a random static game situation as well as the reference path computed with the proposed approach. The number of obstacles is 9 and is selected to represent the maximum number of robots on the field during a game. Including the advanced blocked areas has no additional effect on the approaches and is therefore neglected.
The probabilistic road map (PRM) approach described in section 1.4.1 violates two of

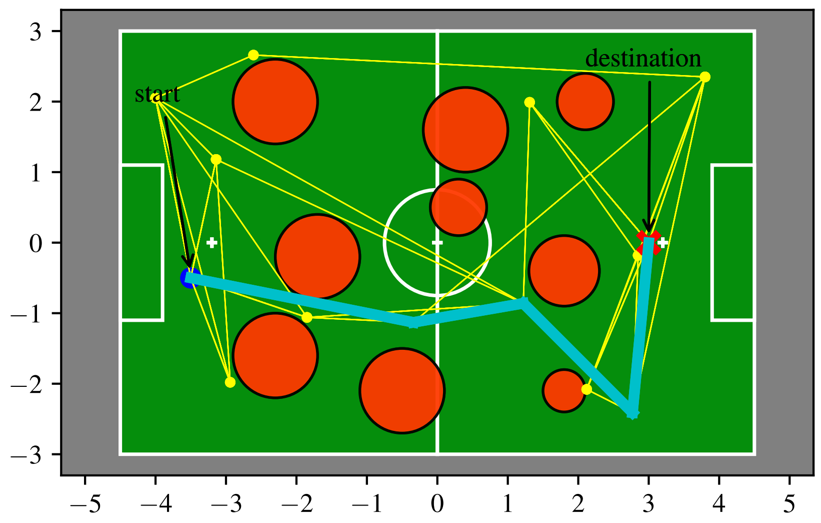


Figure 5.2: *Showing a resulting path of PRM applied on the case study.*

the required aspects. PRM is not able to find the optimal path in a deterministic fashion. Due to its probabilistic method to sample the environment ts unlikely to find the optimal path. Also the found paths could include major detours. Figure 5.2 shows an example solution found by the PRM approach. The first part of the path seems legit, the 2nd part also looks not to much off of the reference path. But the last two straights are introducing such an expensive detour. Not only regarding the length also the sharp corners are bad for the robots motion. The robot will need to stop completely and turn on spot to follow that trajectory, which will consume a long time to reach the destination. Another drawback compared to the proposed approach is the non-deterministic behavior of PRM. Every time PRM will sample the free space differently. As mentioned before, the PRM approach can be modified to a deterministic sample technique to achieve consistent path planning results but regarding the application it's results are out performed beaten by the proposed approach. Looking into detail about PRM can be reduced to three challenges. Connecting neighboring points, collision checking and sampling method. PRM is probabilistic complete. When sampling very often the graph will contain a solution. Drawback in the details would be that the graph is not constructed with the path in mind. Leading to worthless nodes in the graph bringing no benefit.

The next presented approach is based on Rapidly Explore Random Trees (RRT). As mentioned in the overview section 1.4.2, the key aspect is to construct a graph by generating next possible states in the free space using random controls. The graph delivers a tree topology. This method was not implemented and cannot be compared face to face. Also this method can be tweaked by optimizing selected parameter and therefore is not meant to be compared in numbers. But still statements regarding the performance can be made. The execution of random controls in RTT lead to no specific coverage of the free space. It has the advantage to directly include the movement constrains of the robot. RTT is often used in automobile path planning because it can represent the movement limitation very well. Including this in the proposed approach will increase the complexity but still be feasible.

The current method used by the HULKs is based on a potential field. The attractive force pulls the robot towards its destination whereas obstacles introduce a repulsive force distracting the robot. The reference situation using some reasonable intensity of the attractive and repulsive forces is illustrated in figure 5.3. Due to the many obstacles, the field is crowded, in part, the influence radius of the obstacle overlap and add up. This behavior is

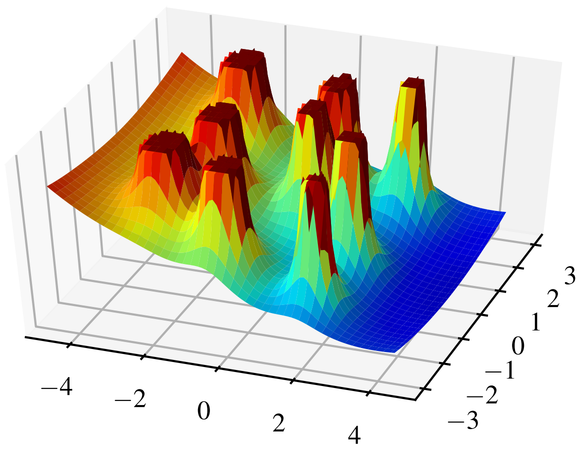


Figure 5.3: *Potential Field illustrating the reference situation.*

not wanted at all and can lead to blocking a passage which is actually wide enough. The HULKs implementation uses the potential field to modify its desired movement vector.The movement vector initially points towards the desired destination and is used to calculate the next foot step. Depending on the location of the robot, the movement vector gets rotated away from obstacles introducing a collision avoidance. The attractive force is represented by setting the initial direction of the movement vector towards the desired destination. The strength of the repulsive forces is tweaked in such a way so that it performs well in most situations. Figure 5.4 shows the top view of the potentials on the reference situation.
The contour lines are drawn in black and the gradient in color goes from red to black

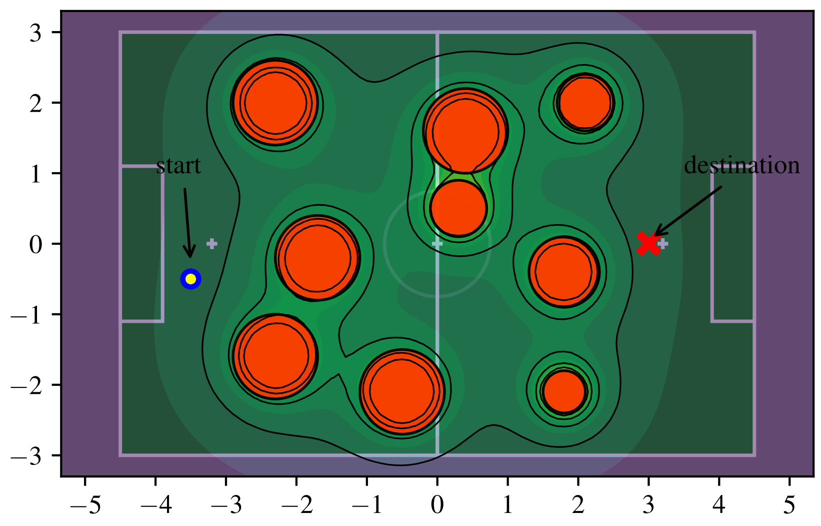


Figure 5.4: *Top view of the reference situation with repulsive potential field and it's contour lines-*

but due to the underlying green of the soccer field the colors are inaccurate. There the problematic passage blocking behavior can be seen in two examples. As soon as two obstacles are close enough to each other, the gap between them gets filled up, so that the field's gradient gets squeezed out. Another major faulty behavior is seen at the destination. The destination is not at the lowest point of the field, due to the obstacle next to it. This can imply that the robot will not find any path exactly terminated at the destination. This needs to be handled separately but still will introduce unwanted detours. Last but not least, in this case the optimal path is not found.
Neglecting the visibility graph approach in the evaluation was a conscious decision and is justified by the problematic implementation of circular obstacle and vice versa. Without any further work-around it cannot be adapted to run in the used framework. The same argument applies to the RRTs implementation.

5.2 Testing

Beeline vs planned path, means.

4.0793 m	Euclidean distance between start and destination.
4.1293 m	Distance of planned of trajectory.

Table 5.1: *Distance between start and destination compared with the final planned path.*

In table 5.1 the beeline length, without obstacle avoidance, is compared to the length of the planned path. Due to the situation design the mean is around 4 m but most important, the planned path is not having a massive impact on the distance. At least over the mean. Obviously this is not true for all cases.

Chapter 6: Conclusion and Outlook

The final chapter will summarize the findings of this work. A short outlook on the future work and further improvements regarding the presented approach can be found in section 6.1.
A entire rework of the a path planning method for the NAO robot focusing on challenges faced in the RoboCup SPL was done. The recent world championship results and statistics were analyzed and used to find problems and weaknesses during participated matches of the team HULKs. Obvious problem of team HULKs considering the other Top4 world wide teams were the faced time penalties caused by insufficient obstacle avoidance. To tackle that weakness, other existing approaches with collision avoidance were investigated. The outlined challenges and requirements were used to evaluate and compare these against each other. Three main requirements were pointed out: 1. Minimized length of path. Reducing the travel time as well avoid the wear and tear of joints. 2. Collision avoidance to reduce time penalties. 3. Smooth trajectory avoiding corners and slow turns on point with no progress in distance. After detailed information about A-Star and tangent computation, the visibility graph based concept of a specialized path planning method was developed.
Main feature of the proposed approach is the way it constructs the graph. Other approaches build up the graph first and then use a search algorithm to find the shortest path between two nodes. In contrast, the developed approach is combining these two steps by using an informed graph search to only construct promising parts of the graph. The search is based on a slightly modified A-Star algorithm. Obstacles are represented in circle form. Introducing other shapes like lines and rectangular can still be used to implement forbidden areas. Some edge cases like starting in an obstacle or enclosed obstacle are handled correctly. The construction of the graph is based on the model of the visibility graph and therefore grantees a good obstacle avoidance. The minimal path length is also implied by the used of visibility properties considering the triangle inequality. The resulting path consists only of straight line segments and circular partial arcs leading around the obstacles.
The 2nd part of this work covers the implementation. Firstly a visualization and debug framework was implemented. Due to good and easy to use visualization libraries, such as PyGame and the later used Matplotlib Python was used for framework devel-

opment. The prototype implementation was also done in Python. To compare methods, the Probabilistic Road Map approach as well as parts of the currently used potential field approach were implemented. The results show that the proposed approach fulfills the stated requirements and can be used on the robot. Unfortunately it is only a proof-of-concept prototype, and no port to the robot is done yet. The final implementation is still pending.

6.1 Further Work

The proposed and exercised approach of a guided path searching delivered promising results. But there is still room for further investigation and chances for improvements. The results promise a good way of avoiding obstacles and finding the shortest path. As an extension the most recent history of the environment could be taken in account as a guess if and how the obstacle may move on, for the next iteration. The proposed method has its strengths in avoiding stationary obstacle, reacting on changes in the environment is handled by computing a path as often as possible. By using state information, movement of obstacles could be used to estimate position changes of obstacles. To include such movement information in the approach can be done by deforming the obstacle's shape regarding the assumed movement. Deformation could depend on the distance from the navigating robot to the obstacle considering the time until the robot will get closer. Therefore the overall approach doesn't need to change critically.
Another point of refinement could be the cost function used during search. So far the cost factor takes only the distance of the path into account, considering additional factors could improve the resulting path. Analyzing the real walking speed of the robot following different sized partial arcs and straights can give insight when it makes sense to take a longer path instead of a lot of narrow arcs.

Bibliography

[AAG$^+$85] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility-polygon search and euclidean shortest paths. In *Proc. 26th Annual Symp. Foundations of Computer Science (sfcs 1985)*, pages 155–164, October 1985.

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[Com18a] RoboCup Technical Committee. Robocup standard platform league (nao) rule book, 2018.

[Com18b] Standard Platform League Technical Committee. Game controller statistics, 2018.

[DB82] Douglas D. Dunlop and Victor R. Basili. A comparative analysis of functional correctness. *A Comparative Analysis of Functional Correctness*, 14:229–244, 1982.

[GM87] S. K. Ghosh and D. M. Mount. An output sensitive algorithm for computing visibility graphs. In *Proc. 28th Annual Symp. Foundations of Computer Science (sfcs 1987)*, pages 11–19, October 1987.

[GO04] Roland Geraerts and Mark H Overmars. A comparative study of probabilistic roadmap planners. In *Algorithmic Foundations of Robotics V*, pages 43–57. Springer, 2004.

[GSB14] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 2997–3004, September 2014.

[HNR72] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. pages 28–29, 1972.

[HS99] John Hershberger and Subhash Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.

[JW10] Qian Jia and Xingsong Wang. An improved potential field method for path planning. In *Proc. Chinese Control and Decision Conf*, pages 2265–2270, May 2010.

[KUT18] W. Khaksar, M. Z. Uddin, and J. Torresen. Incremental adaptive probabilistic roadmaps for mobile robot navigation under uncertain condition. In *Proc. Computing Science and Automatic Control (CCE) 2018 15th Int. Conf. Electrical Engineering*, pages 1–6, September 2018.

[LK99] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int. Conf. Robotics and Automation (Cat. No.99CH36288C)*, volume 1, pages 473–479 vol.1, May 1999.

[NRH15] Kourosh Naderi, Joose Rajamäki, and Perttu Hämäläinen. Rt-rrt*: a real-time path planning algorithm based on rrt. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, pages 113–118. ACM, 2015.

[QMI$^+$14] A. H. Qureshi, S. Mumtaz, K. F. Iqbal, Y. Ayaz, M. S. Muhammad, O. Hasan, W. Y. Kim, and M. Ra. Triangular geometry based optimal motion planning using rrt*-motion planner. In *Proc. IEEE 13th Int. Workshop Advanced Motion Control (AMC)*, pages 380–385, March 2014.

[RId88] N. S. V. Rao, S. S. Iyengar, and G. deSaussure. The visit problem: visibility graph-based solution. In *Proc. IEEE Int. Conf. Robotics and Automation*, pages 1650–1655 vol.3, April 1988.

[SGS17] P. Sudhakara, V. Ganapathy, and K. Sundaran. Probabilistic roadmaps-spline based trajectory planning for wheeled mobile robot. In *Proc. Data Analytics and Soft Computing (ICECDS) 2017 Int. Conf. Energy, Communication*, pages 3579–3583, August 2017.

[SS15] S. Spanogianopoulos and K. Sirlantzis. Non-holonomic path planning of car-like robot using rrt*fn. In *Proc. 12th Int. Conf. Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 53–57, October 2015.