University of Southern Denmark

Group 1

Robot Systems 5th semester - Fall 2018

RB-RCA5 Project

Emil Vincent Ancker
0906-1997
emanc16@student.sdu.dk

Mathias Emil Slettemark-Nielsen
2302-1995
masle16@student.sdu.dk

Mikkel Larsen
0808-1996
milar16@student.sdu.dk

Søren Emil Hansen
0502-1997
soeha16@student.sdu.dk

Supervisors: Anders Lyhne Christensen, Jakob Wilm & Lars-Peter Ellekilde

Project period: 27.09.2018 - 16.12.2018

Abstract

This report explores some of the problems involved in the research fields artificial intelligence, robotics path planning and computer vision when developing a mobile robot capable of collecting marbles.

In this report multiple approaches are considered and compared to select the deemed optimal approach.

The research fields were all used to approach different problems. Artificial intelligence made it possible to control the mobile robot using fuzzy logic and perform local obstacle avoidance. Furthermore artificial intelligence was used to optimize the navigation strategies, by performing reinforcement learning to maximize the amount of collected marbles within a fixed time frame. Through robotic path planning, it was possible to construct a roadmap of a given environment, that made it possible for the robot to move between two points in the environment. When detecting and mapping a marble, computer vision was used.

Contents

1 Project Description

The purpose of this project is to explore the problems involved with making a robot able to find blue marbles in a simulated environment. This will be implemented as algorithms based on the fields artificial intelligence, computer vision and robotic path planning.

The simulation is handled in the simulator Gazebo using Gazebo's robotic simulation framework. At the start of the project, an environment was handed with the robot, marbles and obstacles. The robot is a *Pioneer 2DX* and is equipped with a color camera and a LIDAR scanner.

The project's Github repository can be found here: `https://github.com/ancker1/RCA5-PRO`.

1.1 Project Analysis

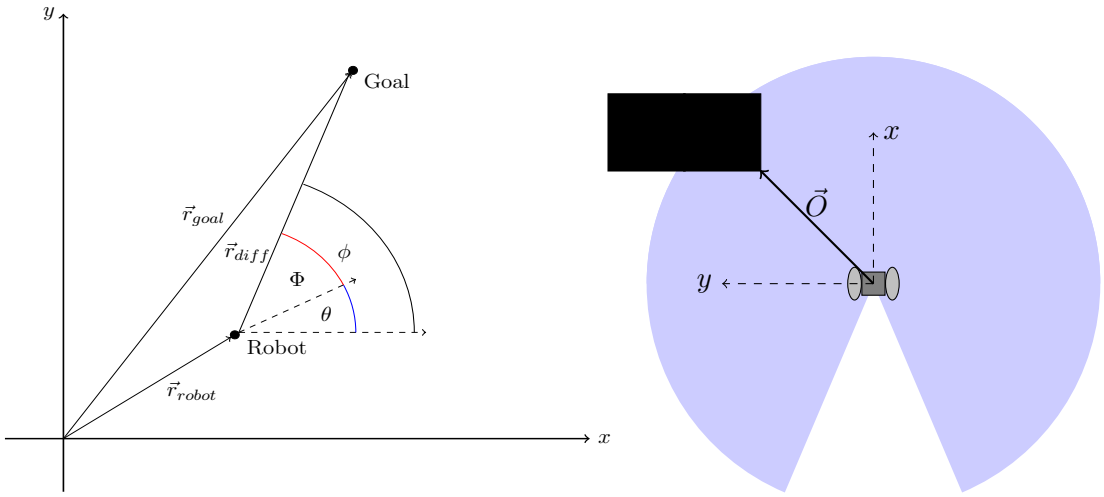
The following subjects will be investigated

- Fuzzy control for local obstacle avoidance and reaching a goal.
- A path planning algorithm making it possible to visit all rooms in the environment.
- An algorithm to travel in a roadmap
- Usage of Q-learning to optimize navigation strategies.
- Computer vision algorithms to detect and locate the marbles.
- Critical points in the environment must be found in order to detect all marbles.

2 Design of Fuzzy Logic For Robot Control

The purpose of the fuzzy controller is to navigate locally between two points in a plane while performing simple obstacle avoidance. For the implementation of fuzzy logic the library FuzzyLite [1] is used.

First step in designing the fuzzy controller is to design and define the inputs.



(a) Inputs relevant to reaching destination. θ is the current orientation of the robot.

(b) Inputs relevant to obstacle avoidance. The blue disk is LIDAR sensors domain.

Figure 1: Inputs for the Fuzzy Controller illustrated

To solve the first part of the task: "Navigate between two points in a plane", it is necessary to know the distance to the goal and the relative angle between the robot's orientation and the goal, as illustrated in figure 1a.

To solve the second part of the task: "Perform simple local obstacle avoidance", it was chosen to give inputs to the controller in terms of the distance and angle to the closest point on the nearest obstacle, which is illustrated in figure 1b as $\vec{O}$.

The input to the Fuzzy Controller to solve the first part of the task is:

$$\Delta\vec{r} = |\vec{r}_{diff}| \angle\Phi \tag{1}$$

Furthermore an input called Path is given to the fuzzy controller making it able to know which way to go around obstacles.

The outputs of the fuzzy controller is **Speed** which sets a common speed for both of the wheels, and **Direction**, which sets an offset in the speed of the wheels resulting in a

rotation. The designed Fuzzy Controller is now as follows:

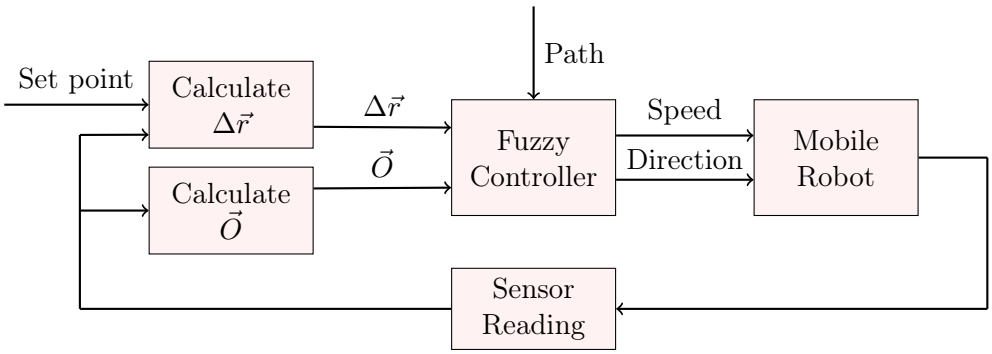
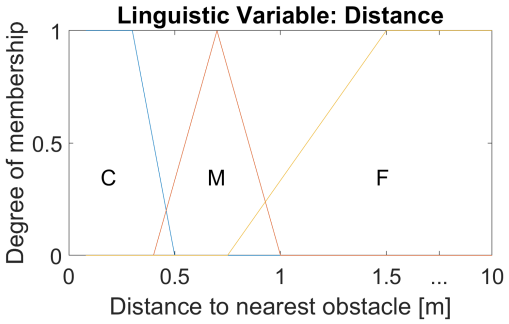


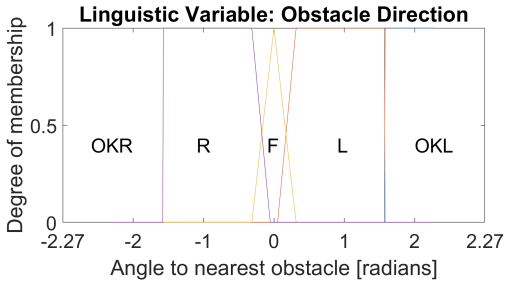
Figure 2: Fuzzy Controller block diagram

2.1 Linguistic Variables

The definition of membership functions is subjective. The designed membership functions for each fuzzy set is plotted against their respective linguistic variables' universal discourse below.

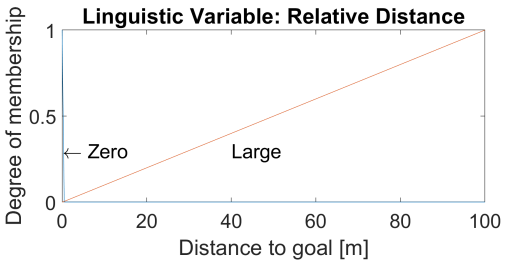


(a) Distance to the closest point on the nearest obstacle

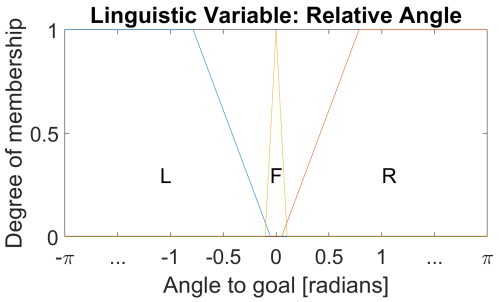


(b) Angle to the closest point on the nearest obstacle

Figure 3: Membership functions for linguistic input variables related to obstacles

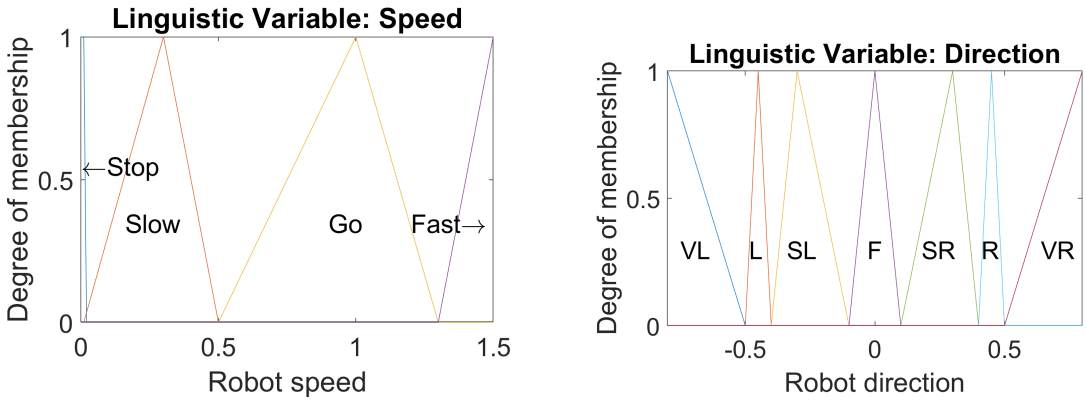


(a) Distance to the goal destination



(b) Angle between the orientation and the goal

Figure 4: Membership functions for linguistic input variables related to the goal point



(a) Speed of the robot

(b) Direction of the robot

Figure 5: Membership functions for the linguistic output variables

The linguistic input variable Path is used to define the mode of the fuzzy controller and is defined by a crisp set:

$$\text{Path} = \mu(x) = \begin{cases} R & \text{if } 0 < x < 1 \\ L & \text{if } 1 < x < 2 \\ S & \text{if } 2 < x < 3 \\ C & \text{if } 3 < x < 4 \\ NON & \text{if } 4 < x < 5 \end{cases} \tag{2}$$

Rulebase

The amount of needed rules to cover all combinations of inputs in each mode is 90. The amount of rules written in each mode is 15, but some rules cover more than one combination, for example:

If Relative Distance is Zero **Then** Direction is F **and** Speed is Stop

The above linguistic rule covers 45 combinations of inputs.

The designed rulebase can be seen in the project Github repository[1].

2.2 Testing the Fuzzy Controller

To test the fuzzy controller multiple hypotheses are tested to indicate the general performance of the fuzzy controller's ability to avoid obstacles when moving towards a set goal position.

[1] `https://github.com/ancker1/RCA5-PRO/blob/fuzzy/robot_control/fuzzycontroller.cpp`

When testing the hypotheses in this section, the robot will be spawned inside a spawn area with a random orientation and the goal is placed randomly inside a goal area. This is done 100 times to test each hypothesis.

GIFs of interesting cases can be found at the project Github reposity[2].

Hypothesis: *The robot is able to navigate left and right around a single obstacle.*
The hypothesis can be split in to smaller hypotheses: *The robot is able to navigate left around a single obstacle* and *The robot is able to navigate right around a single obstacle.*

Hypothesis	# Succesful
Navigate right	100
Navigate left	100

(a) Results

(b) Environment

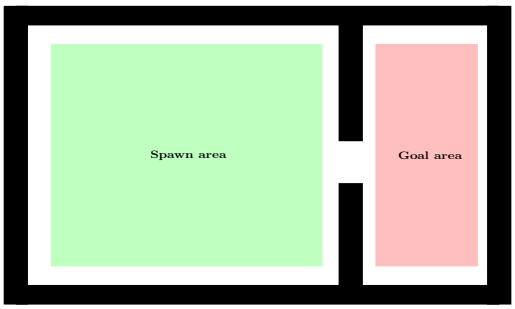
Figure 6: Environment and results from test of hypothesis

The environment, spawn and goal areas are shown in figure 6b for both hypotheses. From the results shown in table 6a the original hypothesis could not be rejected.

Hypothesis: *The robot is able to navigate through a room entrance.*

Hypothesis	# Succesful
Navigate through entrance	79

(a) Results



(b) Environment

Figure 7: Environment and results from test of hypothesis

From the results shown in table 7a it is seen that the robot will fail 21 out of 100 times. The cases where the robot failed were all because of the same phenomenon, this is because

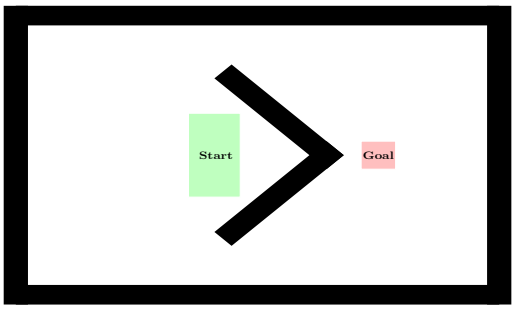
[2] `https://github.com/ancker1/RCA5-PRO/tree/fuzzy/Test/Fuzzy`

the robot will come near to both of the sides of the entrance, which can be seen in the GIF[3] for the entrance obstacle.

Hypothesis: *The robot is able to navigate around nonsimple obstacles.*

Obstacle type	# Succesful
Corner I	100
Corner II	100
Star	100
Curve	0

(a) Results



(b) Environment: *Corner I*

Figure 8: An example of the used environments and results from test of hypothesis

The environments used to test *Corner II*, *Star* and *Curve* can be seen in appendix A.1. From the results in table 8a the hypothesis is rejected, since the robot only was able to navigate around in some of the tested environments.

In the testing of the hypothesis, it was discovered that the robot will not be able to navigate around obstacles, where an obstacle and the goal are on each side of the robot, without the robot being able to go directly to the goal. This phenomenon can be seen in the GIF[4] for the curve obstacle.

[3] https://github.com/ancker1/RCA5-PRO/blob/fuzzy/Test/Fuzzy/door_entrance.gif
[4] https://github.com/ancker1/RCA5-PRO/blob/fuzzy/Test/Fuzzy/curve.gif

3 Recognition of the blue marbles

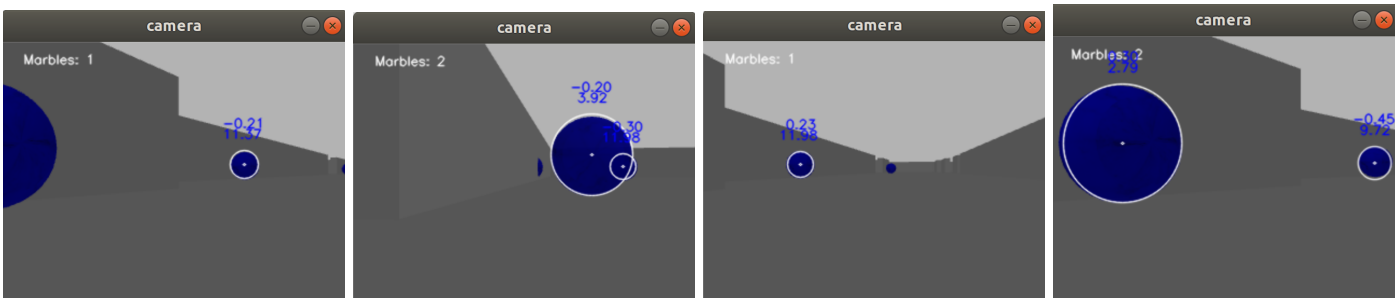
In order for the robot to recognize the blue marbles, initially the blue color must be separated from the rest of the image by thresholding the image. Afterwards, the robot must be able to recognize the marble object using various approaches: *Hough Circle Transform*, *Square Fit* and *Template Matching*.

3.1 Hough Circle Transform

Hough Circle Transform is a feature extracting algorithm, based on edges in an image. In Hough space a circle with a constant radius is drawn for every edge points in the image space. Each circle in the Hough space votes for an origin of the detected circle in the image. The point where most circles intersect is the center of the detected circle.

Implementation of Hough Circle Transform

To apply the Hough Circle Transform the incorporated function in OpenCV is used. The minimum distance between detected centers is set to $\frac{image\ width}{15}$ which is a trade-off between false positives and false negatives. The minimum and maximum radius of the detected circles are set to zero. Thereby, Hough Circle Transform will detect circles with any radius. The upper threshold for the canny edge detector is set to 5 and lower threshold to half of this. Finally, the accumulator threshold is set to 25. The Hough transform with the parameters described is illustrate on figure 9 below.



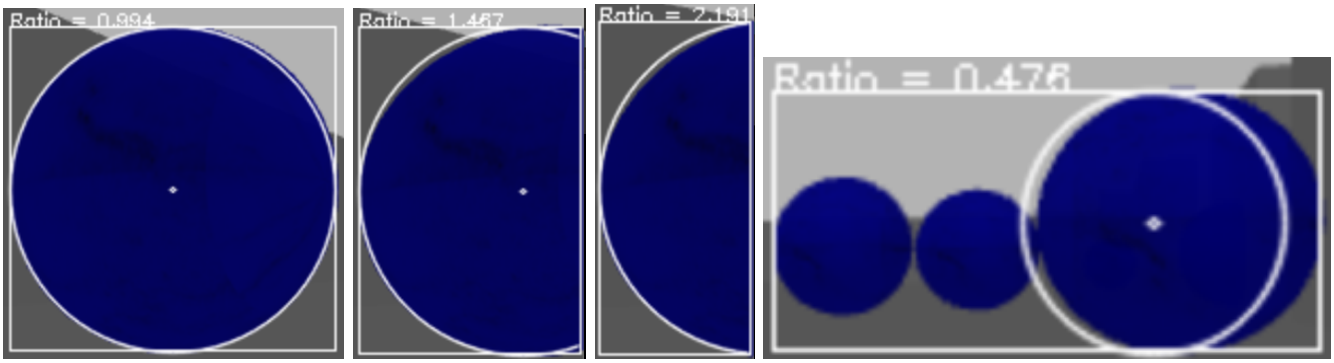
(a) Less than half showing. (b) Stacked circles (c) Example of false negative. (d) Complete circle

Figure 9: Hough functionality examples

The implemented Hough Circle Transform with the parameters described above does not detect circles where under half of the circles are showing, as illustrated in figure 9a. The overlapping circles can be detected as seen in figure 9b. The accumulator threshold is also rarely surpassed by distant circles, as fewer edge pixels are present, which can be seen in 9c.

3.2 Square Fit

A primitive method compared to Hough Circle Transform is implemented to minimize computational complexity. This method only works in an environment where the object to detect have a distinct color difference from the background. The first function performs color thresholding to the marble from the rest of the image. Afterwards it detects the contours in the filtered image to calculate the center image coordinates, (x_0, y_0), and radius, r, for each circle. The radius of each circle is calculated by fitting a square to each circle. This is illustrated on figure 10 below.



(a) $h : w \approx 1$ (b) $1 < h : w < 2$ (c) $2 < h : w$ (d) $h : w < 1$

Figure 10: Square Fit categorization at different $h : w$ ratios

The following methods is used to calculate the center coordinates and radius for each circle. Where x_r is the upper-left corner x-value of the rectangle in the image. h and w are the height and width of the rectangle, respectively.

- If a full circle is showing: ratio $h : w \approx 1$.
  - (x_0, y_0) will be equal to the center of the rectangle.
  - The radius will be given by: $r = \frac{w}{2}$.
- More than half of a circle or more than one circle is showing: $1 < h : w < 2$ or $h : w < 1$.
  - y_0 is set equal to the horizontal center of the rectangle.
  - The radius is set to: $r = \frac{h}{2}$
  - x_0 is found to be the point, where the contour is highest.
- Less than half of circle is showing: ratio $h : w > 2$.
  - y_0 is equal to the horizontal center of the rectangle.
  - The radius is set to: $r = \frac{w}{2} + \frac{h^2}{8w}$, which is the radius of an arc.
  - x_0 is either $x_r + r$ or $x_r + w - r$ depending on the side of rectangle showing.

The above will result in only the largest marble being detected, when multiple marbles are overlapping as illustrated on figure 10d.

3.3 Template matching

When performing template matching a template is moved over the whole image while evaluating a matching metric and storing it in the matching space. The normalized cross correlation is used as matching metric, since the detection will be invariant to overall lumination. The score ranges between 0 and 1.0, where 1.0 is a perfect match. A threshold of 0.8 is chosen, which yields the detection in figure 11 using a screenshot of a marble as target image in 10 different sizes.

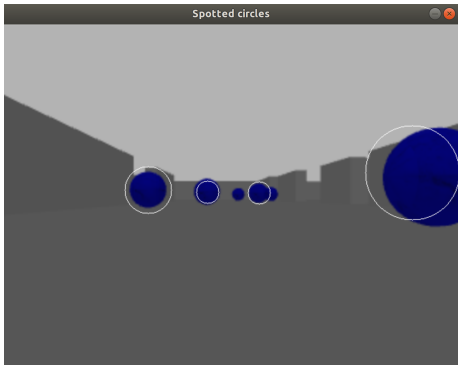


Figure 11: Template matching detection.

This method is only able to identify full circles correctly, since it is searching for a complete match of the template. A larger amount of target images can be used, but then the computational complexity would increase.

3.4 Computational time

A test for each of the three circle detection methods are made on figure 12 with 1000 iterations. The tests are run on an Intel Core i7-6650U processor with 2.20 GHz, yielding the following average running times:

- Hough Circle Transform: 4.74 [ms]
- Square Fit: 1.32 [ms]
- Template matching: 150.92 [ms]

This test is not very accurate, since a lot of factors are involved including the resources available at execution time. But it can still give an indication of the computation time.

Figure 12: computation time test image.

3.5 Mapping the marbles

It would be favorable to map the location of the blue marbles. This requires that the distance and angle to the marbles are calculated from the detected circles.

Distance to marble

Mapping from a point in the 3D world (x, y, z) to a point on the image plane (i, j) for the simple pinhole camera, can be modelled as the following matrix:

$$\begin{bmatrix} i \cdot w \\ j \cdot w \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & c_i \\ 0 & f & c_j \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \frac{\text{width}}{2\tan\left(\frac{\text{FOV}}{2}\right)} & 0 & \frac{\text{width - 1}}{2} \\ 0 & \frac{\text{width}}{2\tan\left(\frac{\text{FOV}}{2}\right)} & \frac{\text{height - 1}}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3}$$

The parameters for the camera matrix are declared in `model.sdf`.

The distance to the marble can be calculated with the camera matrix. Figure 13 shows a sketch of the marble in the real world and the marble on the image, where the center of the marble is at the optical center. From figure 13 and equation 3, equation 4 is deduced.

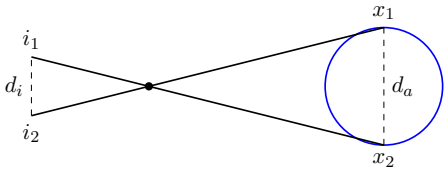


Figure 13: Sketch of distance to marble

$$\begin{aligned} i_1 \cdot w &= f \cdot x_1 + c_i \cdot z \\ i_2 \cdot w &= f \cdot x_2 + c_i \cdot z \end{aligned} \Rightarrow (i_2 - i_1) \cdot w = f \cdot (x_2 - x_1) \Rightarrow \frac{f}{(i_2 - i_1)} = \frac{w}{(x_2 - x_1)} \tag{4}$$

Figure 13 and equation 4 yields the relationship $\frac{f}{d_i} = \frac{w}{d_a}$, since d_i, d_a and f is known, the distance w can be found.

Angle to marble

The angle ϕ to the circle can be found using trigonometry:

$$\phi = \text{FOV}\left(0.5 - \frac{x_0}{\text{Width}}\right)$$

Combining the distance and the angle to the marble with the current location and orientation of the robot, the position of the marbles in the real world can be calculated.

Deciding if detected marble should be mapped

To determine if the marble has been discovered before the detected circle is checked whether it is overlapping a previous circle. If so the circles are very likely the same circle. Furthermore, the coordinates of the circle are checked whether they are within a certain interval of discovered marbles. The interval is proportional to the distance to the marble, thereby making up for the uncertainty of distance.

The coordinates of a marble is updated to be a weighted mean of all circles' coordinates which has been determined to be the same marble. The weight depends inversely on the distance.

Error as function of distance

The different algorithms are tested by moving the robot so it is facing the marble with a distance of 2 meters. The correct position of the marble in the map is known and the distance between this and the calculated distance of the observed marble is noted for each algorithm. The marble is then moved in 1 meter intervals, always noting the error of each algorithm, until it is 35 meters away. This is meaned over 100 iterations yielding the plots in figure 14.

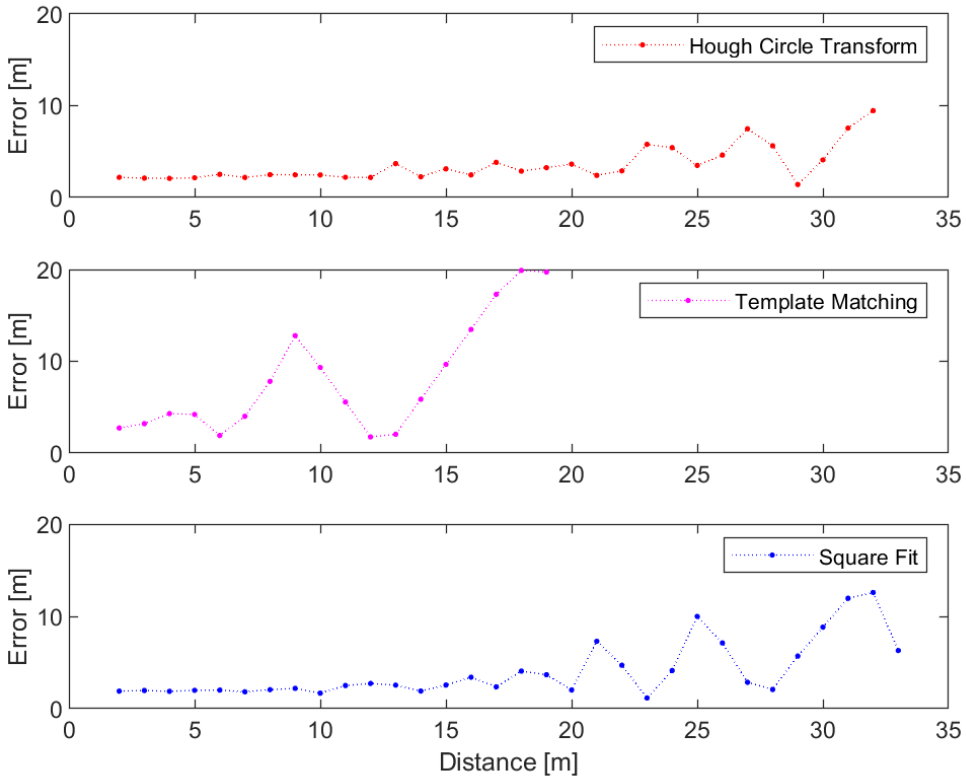


Figure 14: Error of circle detection algorithms at different distances.

In figure 14 it can be seen that Hough Circle Transform and Square Fit perform about equally well until 20 meters, where the error in increases to above 5 meters. Additionally, the Template Matching algorithm is unable to spot circles farther than 20 meters away.

Mapping test

The mapping is tested by manually exploring all the rooms. In figure 15, the location of

the marbles can be seen using all three detection algorithms running simultaneously.

Figure 15: Marbles mapped with Hough are colored red, template matching are colored green, square fit are colored pink. The 20 spawned marbles are colored blue.

Figure 15 shows that none of the algorithms detects all the marbles perfectly. From the test performed the average distance from the detected marble to the actual marbles are shown in table 1.

Algorithm	**Average distance [m]**	**# Marbles detected**
Hough Circle Transform	1.84	22
Square Fit	3.89	26
Template Matching	6.94	35

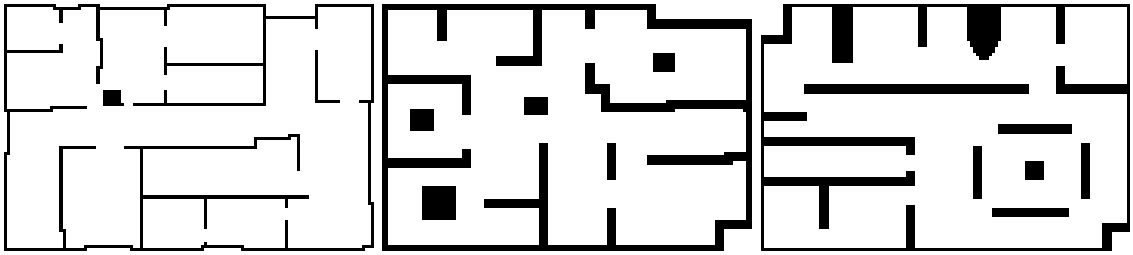
Table 1: Results from the mapping test

4 Finding critical points

In order for the robot to know which points should be visited in the environment the floor plan must be analyzed. To analyze the floor plan two approaches are considered.

1. Detect local maxima in a brushfire grid
2. Detect rooms as rectangles and find the centers

The maps, shown in figure 16, are used to test the approaches stated above.



(a) Map 1 (b) Map 2 (c) Map 3

Figure 16: Different floor plans for experfiments

4.1 Local maxima of Brushfire

The Brushfire algorithm computes a brushfire grid where each pixel holds the minimum distance to an obstacle measured in pixels. Thereby, making a resolution complete map. The local maxima of the brushfire grid are found by dilation. Afterwards, the local maxima are found by comparison of a dilated version of the image and the brushfire grid. Figure 17 shows the critical points derived from the local maxima.

Figure 17: Critical Points found with brushfire

4.2 Detect rectangles

For each nonobstacle pixels a rectangle is made. The boundaries of the rectangle is found by expanding from the pixel in both width and height until an obstacle is met. The rectangle is then classified as an obstacle to avoid double detection. Then the next nonobstacle pixel is found from the altered image. The rectangles are thresholded by their area to decrease the amount of critical points, which is the centers of the rectangles. Figure 18 shows the critical points found with rectangles in map 1.

Figure 18: Critical Points found with rectangles

4.3 Brushfire versus Rectangle

The two methods are compared by the amount of critical points and their visibility of the maps. The number of critical points should be as low as possible but still give full visibility of the maps. Table 2 shows the two methods critical points and their visibility on different maps.

	Map 1		Map 2		Map 3	
	Critical Points	Visibility	Critical Points	Visibility	Critical Points	Visibility
Brushfire	17	100 %	16	99.97 %	17	100 %
Rectangles	20	99.96 %	19	99.95 %	16	99.88 %

Table 2: Critical points and their visibility on different maps

From table 2 it can be concluded that the brushfire method performs better than the rectangle method on Map 1, 2 and 3.

Images showing the visibility of the different maps can be found in the project Github repository.[5]

[5] https://github.com/ancker1/RCA5-PRO/tree/master/Test/Critical%20Points

5 Route planning

In this section two methods to develop a roadmap will be described and then compared to each other, to make the robot able to navigate in an environment. During the design and test in this section the maps shown in figure 16 have been used.

5.1 Voronoi Diagram

The Voronoi diagram creates a road map with accessibility, departability and connectivity. The road map is based on the Voronoi vertices. The Voronoi vertices are where boundaries between Voronoi regions meet. Hence, making a road map with maximum distance to the obstacles. The implemented Voronoi diagram is offline and based on a resolution complete model.

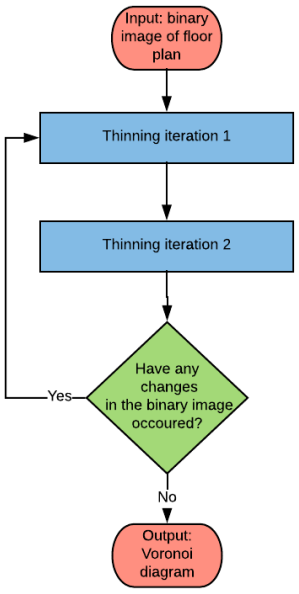
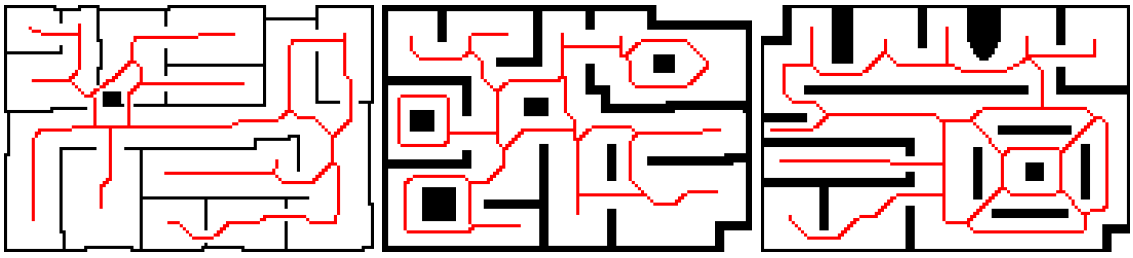


Figure 19: Flowchart of Zhang-Suen algorithm

Implementation of Voronoi Diagram

The Voronoi diagram is implemented by using the Zhang-Suen algorithm [2]. The flowchart of the implemented Zhang-Suen is shown on figure 19. The Zhang-Suen algorithm operates on binary images, evaluating individual pixels and comparing it to its neighbors, performing thinning of the white nonobstacle area, resulting in the Voronoi diagram if performed until no change have occured.

Figure 20 shows the Voronoi diagram for different floor plans.



(a) Map 1 (b) Map 2 (c) Map 3

Figure 20: Voronoi diagrams of different floor plans

A GIF showing the accessibility, departability and connectivity for the implemented Voronoi diagram can be found in the project Github repository.[6]

[6] https://github.com/ancker1/RCA5-PRO/tree/master/Test/Roadmap/VoronoiDiagram

5.2 Boustrophedon Decomposition

From the Boustrophedon decomposition a road map with accessibility, departability and connectivity can be created. Since the environment is known in advance, it is analyzed before given to the robot. Therefore, the algorithm is implemented as offline and model based. Furthermore it is implemented as resolution complete since the map of the environment is discrete. The Boustrophedon decomposition was chosen instead of trapezoidal to get less vertices, but still good road map coverage.

Implementation of Boustrophedon Decomposition

The implemented Boustrophedon decomposition is split up into two functions. In the first part outer corners are found, then in the second part the corners are used to find a cell point between the corner and an obstacle or an adjacent corner point to establish the cells.

Corner detection

A simple method for finding both outer and inner corners was chosen. Thus, the method only works on a simple binary pixel map. An appropriate kernel, shown in figure 21 is convoluted with the binary pixel map. If one of the eight unique sums is calculated a corner is detected.

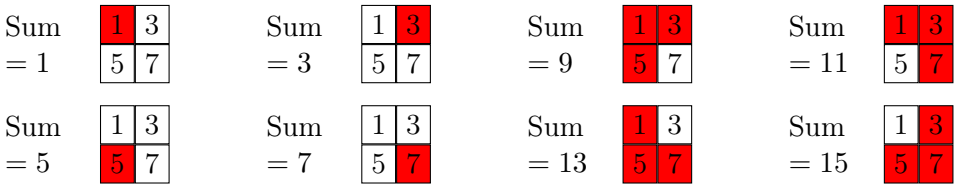
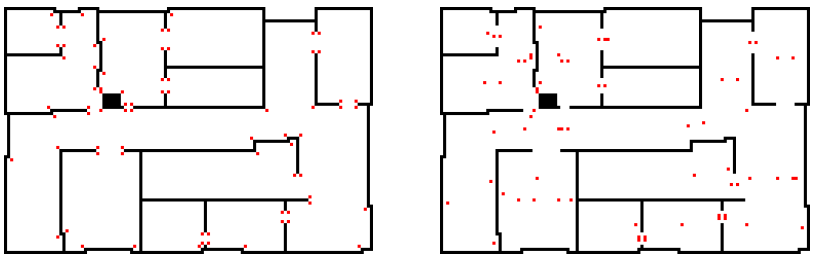


Figure 21: Outer corner detection 4 most left, Inner corner detection 4 most right

As the Boustrophedon decomposition only uses outer corners as vertex detection, the inner corners are not used. The corner detection used on Map 1 can be seen in figure 22a.



(a) Outer corner detection performed on Map 1

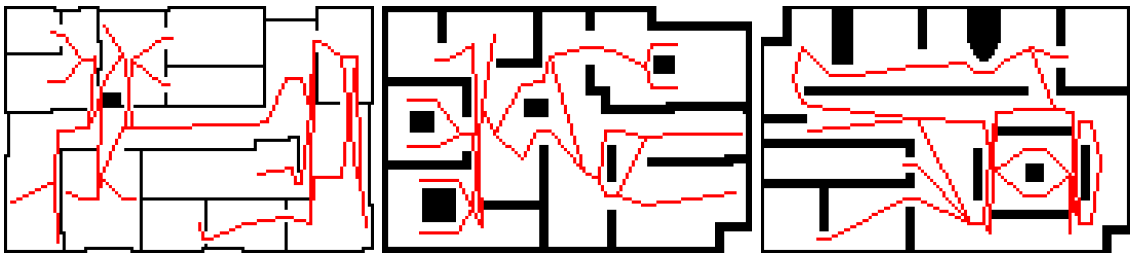
(b) Cell points calculated on Map 1

Figure 22: Method for Boustrophedon decomposition

Cell point and Cell detection

The cell points are placed in the middle between the currently evaluated corner and an adjacent corner or obstacle. This is illustrate on figure 22b.

The cells are defined as the space between the cell points. By looking at each individual cell and connecting each cell point within that cell to the opposite cell points the roadmap will be generated as illustrated in 23.



(a) Map 1 (b) Map 2 (c) Map 3

Figure 23: Roadmap generated using Boustrophedon decomposition

A GIF showing the accessibility, departability and connectivity for the implemented Boustrophedon decomposition can be found in the project Github repository.[7]

5.3 Graph Search Algorithm

To plan the route on the roadmap the problem can be reduced to finding the best path between two points within a graph. Therefor a graph search algorithm is implemented. Two algorithm were considered, A* and Dijkstra. A* uses a heuristic, h, which represents the distance from a node to the end goal. Furthermore a cost, g, is defined as the sum of distances between the nodes from the starting node to the current node. The estimated shortest path is calculated as $f = g + h$, which A* uses in its best-fit search to find the shortest path. In contrast to this, Dijkstra only uses the cost g in its exhaustive search

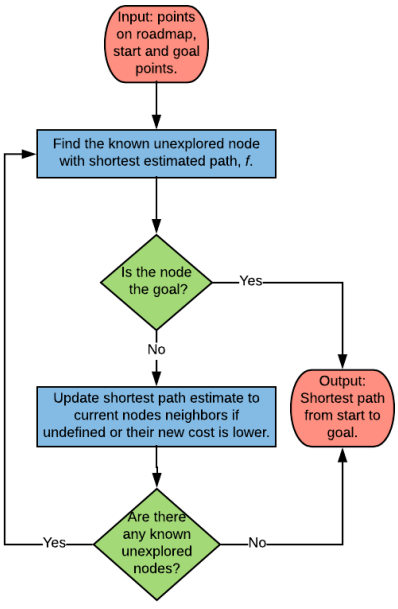


Figure 24: Flow chart of implemented A*

[7] https://github.com/ancker1/RCA5-PRO/tree/master/Test/Roadmap/BoustrophedonDecomposition

to find the shortest path. A* is chosen since it only explores a subset of what Dijkstra explores.

Implementation of A*

The map is initialized with the roadmap, the start point and the goal point. The rest of the map is defined as obstacles. After the initialization of the map the A* algorithm follows the flowchart on figure 24. The A* code implemented has drawn inspiration from [3].

A GIF showing an example of the implemented A* algorithm can be found in the project Github repository.[8]

5.4 Comparing road maps

To compare the roadmaps, Voronoi and Boustrophedon, an experiment is performed. 10,000 start and end points are generated using a uniform distribution. Not all 10,000 start and end pairs are used since not all can be connected to the roadmap or they are within an obstacle. The generated start and end points are connected to the their respective closest point on the roadmap.

The start and end points can be connected in two ways:

1. A direct path is drawn if no obstacles are inbetween.
2. Start and end points are connected to the roadmap and the shortest path on the roadmap is taken.

The results from Voronoi is sorted by distance and Boustrophedon is sorted with respect to Voronoi. The start and end points are noted as sample numbers. The results for Map 1 is illustrated on figure 25.

[8] https://github.com/ancker1/RCA5-PRO/blob/master/Test/A%20star%20-%20Voronoi%20diagram.gif

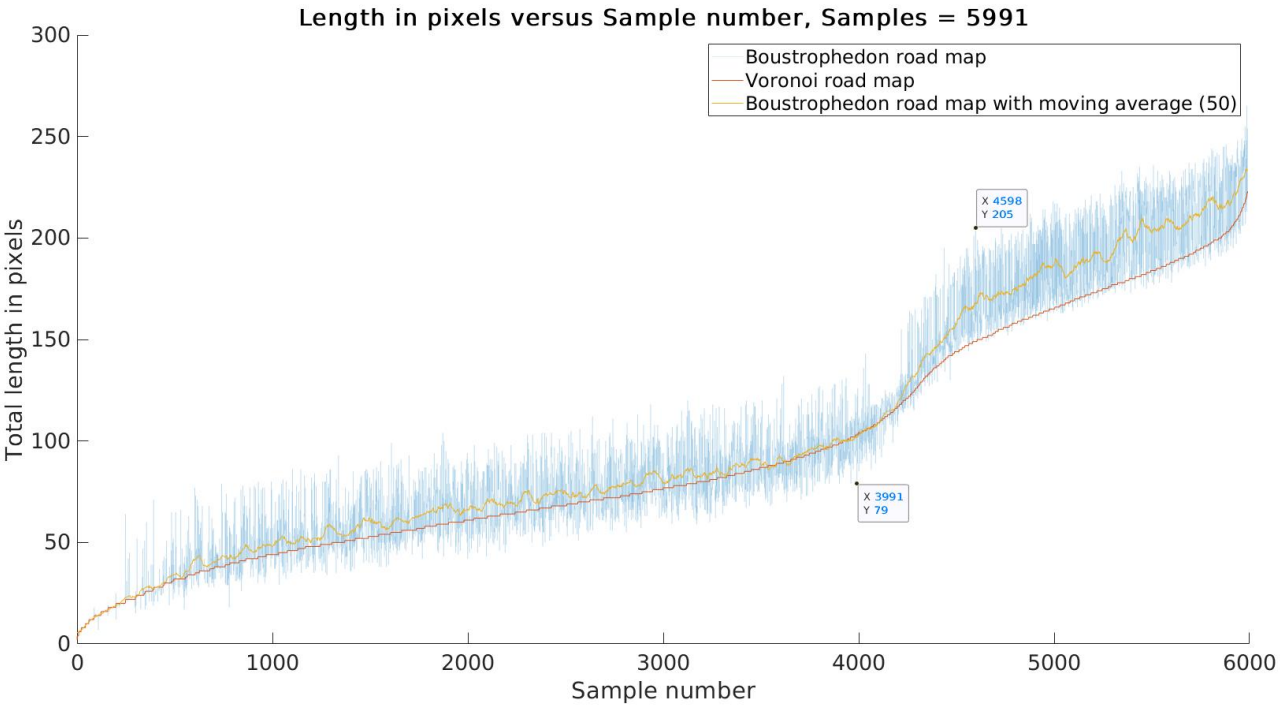
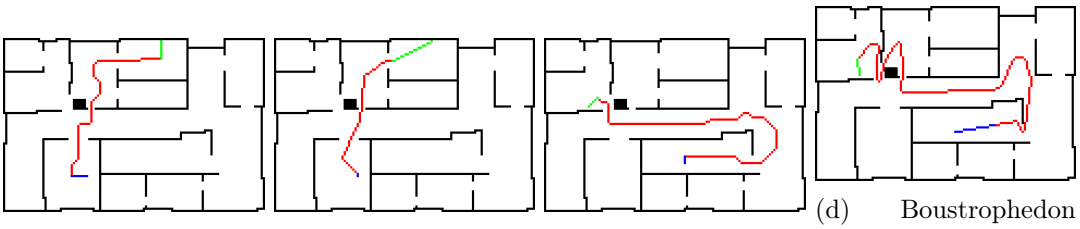


Figure 25: Comparing distances on roadmaps at Map 1

The two data boxes shown in figure 25 shows the two cases with the largest negative and positive difference in comparison of Voronoi and Boustrophedon. These two routes are illustrated in figure 26. By looking at the figure 25 it is seen that Voronoi is the best roadmap with respect to distance in Map 1.



(a) Voronoi worst case, index 3991.

(b) Boustrophedon best case, index 3991.

(c) Voronoi best case, index 4598.

(d) Boustrophedon worst case, index 4598.

Figure 26: Worst and best cases for Voronoi and Boustrophedon with respect to distance travelled.

	Map 1		Map 2		Map 3	
	Voronoi	Boustrophedon	Voronoi	Boustrophedon	Voronoi	Boustrophedon
# Usable samples	5991	5991	5280	5280	5220	5220
Total length [Pixels]	560691	606856	353632	356894	430254	447313
Best choice [%]	57.1	24.8	32.9	38.9	48.4	27.6

Table 3: Results for length experiment on Map 1, Map 2 and Map 3

The experiment is performed on the maps in figure 16. The results are shown in table 3. The results in table 3 indicates that in general Voronoi is the best roadmap with respect to distance.

The length experiments for the maps can be found at the project Github repository.[9]

[9] https://github.com/ancker1/RCA5-PRO/tree/master/Test/Roadmap/Length%20test

6 Reinforcement Learning For Navigation Optimization

The purpose of the navigation optimization is to optimize the sequence of visited rooms in a fixed time frame. The off-policy temporal difference algorithm Q-learning is being used for this optimization.

The definition of Q-learning:

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{5}$$

By the above definition the action-value function, Q, will approximate the optimal action-value function, q_*, but this will only be done if all state-action pairs are visited and updated. Therefore, a policy must be chosen that fulfills this.

Choosing a policy

It was chosen to use an ϵ-greedy policy, since this will fulfill the requirement of exploration. The policy is defined by:

$$\pi(s) : A \longleftarrow \begin{cases} \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{6}$$

The policy stated in equation 6 describes a policy which will choose the greedy action most often, but at a probability of ϵ a random action will be chosen.

Design of states

Since the optimization problem is in what sequence to visit the rooms given a finite time horizon, it was chosen to implement the environment as a graph structure shown in figure 27.

The states are dependent on the current node that the agent is located in, and to make sure that the reinforcement learning task has the Markov Property, the state will also be dependent on the nodes that have been visited prior to the current node. This will lead to the following amount of states:

$$\text{amount states} = N \cdot 2^N \tag{7}$$

where N is the amount of nodes.

Design of actions

The actions are simply defined as a change of node, meaning if the action is *1*, the agent

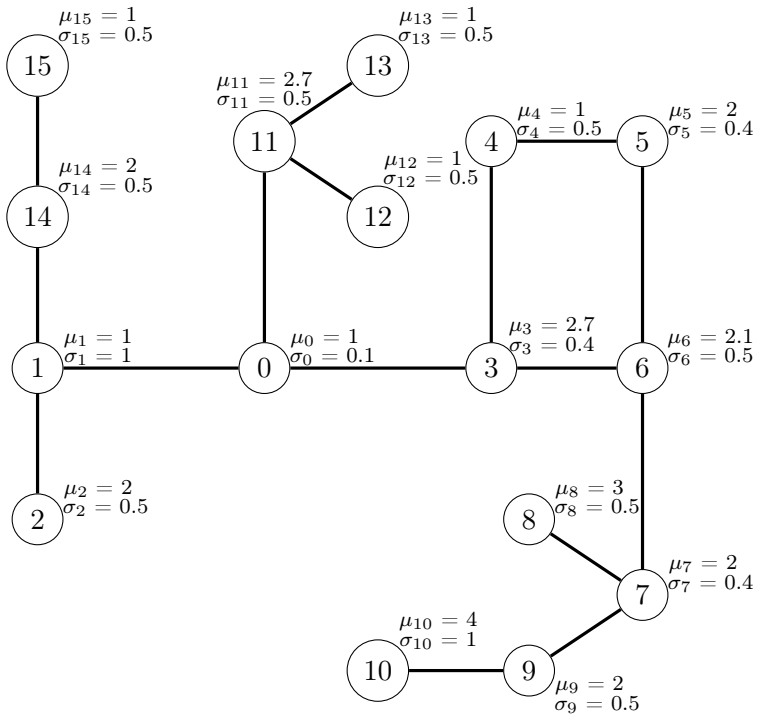


Figure 27: Graph environment used for Q-Learning

will go to node 1. In figure 27, it is seen that not all nodes are connected, meaning that for each state it needs to be checked which nodes the agent can travel to.

Design of reward

The reward in each node is given based on a normal distribution. Each node has a mean and a standard deviation which will generate a random amount of marbles, that each gives a reward of 10, based on the normal distribution. The reward will only be granted the first time the agent visits the node. A negative reward will be given when the agent visits a node already visited, which is equal to negative half a marble.

6.1 Implementation of Q-learning

With the design of states and actions stated above, the naive implementation of the action-value function Q will be a 3-dimensional matrix defined as:

$$\text{float Q[N][}2^N\text{][N]}$$

The memory usage in bytes is given by:

$$\text{Usage in bytes} = N^2 \cdot 2^N \cdot 4 \tag{8}$$

Where N is the amount of nodes. From equation 8 it is seen that the memory complexity

is $\mathcal{O}(2^N)$. This raises a problem, for example a graph containing 25 nodes will require allocation of a 3-dimensional matrix that will take up 20 [GB] memory. Therefore, it is chosen to implement the linked list data structure, where only the entries needed in the matrix implementation are created as a link node. The drawback of implementing the linked list data structure, is the time complexity of executing a search in the linked list which in worst case is $\mathcal{O}(N)$. The graph below shows the memory usage with the implementation of linked list.

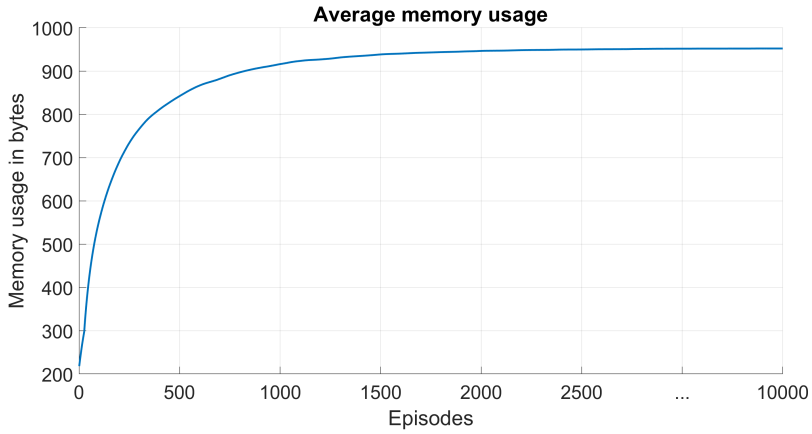


Figure 28: Memory usage as a function of processed episodes meaned over 500 iterations on a graph containing 16 nodes

It is seen from graph 28 that the memory usage is drastically reduced.

6.1.1 Implementing Q-learning on a graph with 16 nodes

The graph represented in figure 27 is used in the implementation. The amount of steps was chosen to be 7, meaning that the agent is able to shift room 7 times in each episode.

Choosing values of relevant variables

The relevant variables are: ϵ, α, γ and β, where β is the rate of epsilon decay. Furthermore the initialization of the action-value function Q might appear to be relevant.

The Q-Learning algorithm is tested at different interesting values of the variables:

- $\epsilon = \{\ 0.01,\ 0.10,\ 0.20\ \}$
- $\alpha = \{\ 0.10,\ 0.20,\ 0.30,\ 0.50\ \}$
- $\gamma = \{\ 0.10,\ 0.20,\ 0.30\ \}$
- $\beta = \{\ 1.0,\ 0.999\ \}$

All the possible combinations of the above values are tested with realistic initial values: $Q(s,a) = 0\ \forall\ s \in S^+,\ a \in A(s)$. Each test is meaned over 500 iterations each containing

10,000 episodes. To show the data meaningful a moving average of 25 is performed. The interesting results are shown in the graphs shown in figure 29.

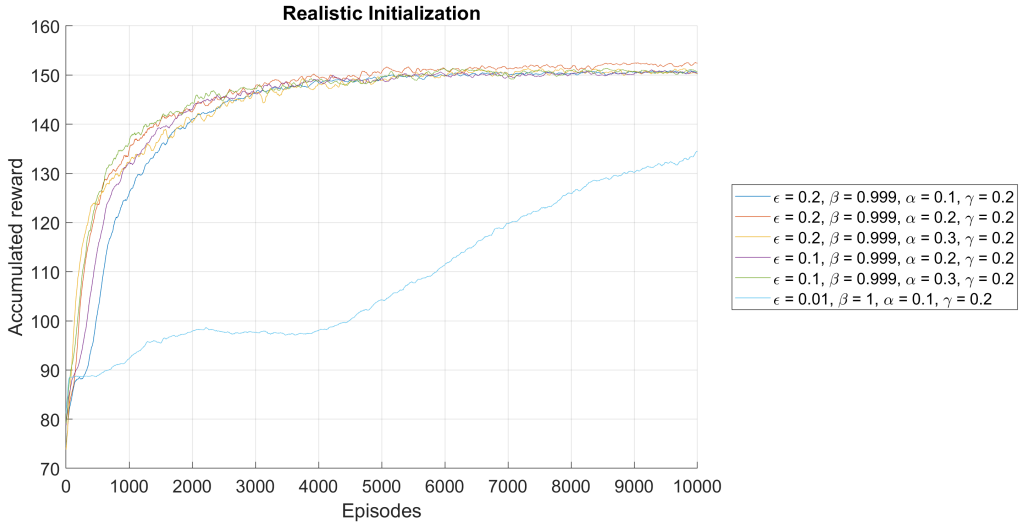


Figure 29: Sum of rewards as a function of processed episodes with realistic initialization

The combinations of variables found to be interesting are now tested with optimistic initialization of the action-value function: $Q(s,a) = 15 \ \forall \ s \in S^+, \ a \in A(s)$

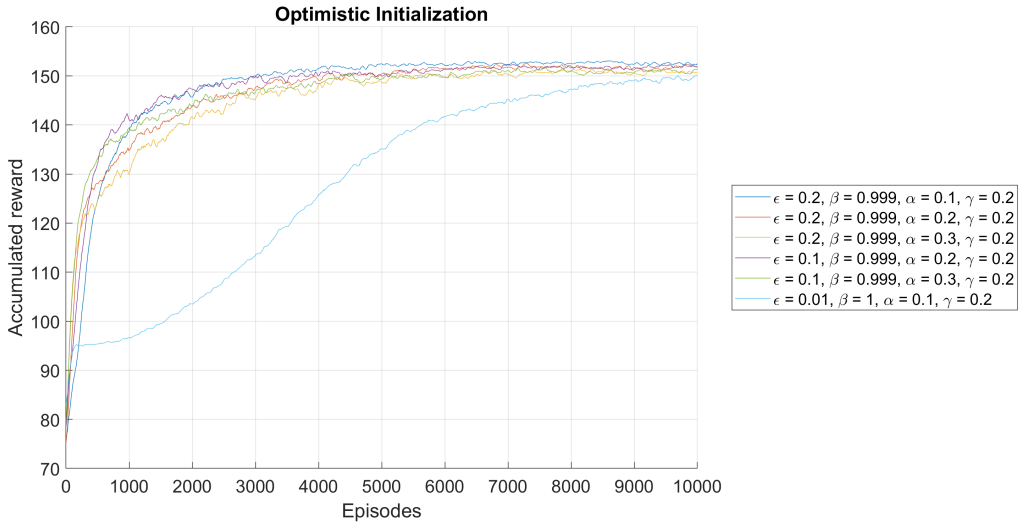


Figure 30: Sum of rewards as a function of processed episodes with optimistic initialization

It is seen in graph 30 that it is beneficial to initialize the action-value function as optimistic, since it will lower the amount of episodes processed for the combinations to reach an accumulated reward of approximately 153, which is the maximum average reward in 7 room changes, which can be seen from the graph in figure 27.

A good combination of inputs is chosen to be: $\epsilon = 0.2, \beta = 0.999, \alpha = 0.1, \gamma = 0.2$ with optimistic initialization of the action-value function.

7 Discussion & Optimization

In subsection 6.1.1 it was described how the relevant variables were chosen and tested. It might show that some other combinations are better than the one chosen. This could be tested by increasing the amount of values for each variable or using techniques for maximizing functions with several variables.

It might be relevant to investigate maximization bias, which occurs because Q-learning uses the maximum action-value estimate to select the action and as an estimate of the true value of the action. To avoid this, double Q-learning could be implemented. Double Q-learning will use an action-value function to choose the action and another to estimate the true value of the action. This will increase the memory usage, but the computational complexity will remain the same.

In subsection 3.5 the distance to the marble was found by using the camera matrix. The calculation of the distance to the marble was based on the marble being in the optical center of the image. This is not always the case. If a marble is placed outside the optical center a distortion will occur, which will have an impact on the shape of the marbles seen in the camera.

The localization of the robot is based on an ideal GPS extracted from Gazebo. If the robot was to be developed to operate in the real world, there might be noise on the GPS. To get a more accurate location of the robot, Kalman filter could be implemented to predict the location. To implement a Kalman filter, a system model and observation model of the robot need to be made.

The planning of the environment in section 4 and 5 is based on a known map. If the environment instead was unknown, a sensor based approach that could map the environment would need to be implemented. In the sensor based approach the robot would need to construct a map based on feature extraction, which could be done with line feature extraction using the LIDAR sensor.

8 Conclusion

During this project a fuzzy controller was developed that made the mobile robot able to perform local obstacle avoidance while travelling to a set destination, it was shown that the fuzzy controller had limitations when it faced more complicated obstacles.

In coverage of the floor plan, it was shown that generating critical points from the brushfire algorithm would be favorable in terms of visibility of the map and minimizing the amount of critical points, compared to expanding rectangles in the floor plan to generate critical points.

A roadmap generated from Boustrophedon decomposition was compared with a roadmap from a Voronoi diagram, showing that the Voronoi roadmap would most often supply the shortest path. With implementation of the graph search algorithm A*, it was possible to find the shortest way in the roadmaps. A* was then used in a random sampling test of the roadmaps showing that the accumulated distance travelled was lowest for the Voronoi-based roadmap.

The implementation and test of Q-learning within a fixed time frame, using an ϵ-greedy policy in a graph with 16 nodes representing the robot's environment, showed that the following combination of variables made the robot able to reach the maximum average reward when initializing the action-value function optimistically: $\epsilon = 0.2, \beta = 0.999, \alpha = 0.1, \gamma = 0.2$.

It was shown by comparison of the Square Fit algorithm, Template Matching and Hough Circle Transform, that the Hough Circle Transform was favorable with respect to false positives and error as a function of distance, when detecting the marbles.

9 Bibliography

[1] Juan Rada-Vilela. fuzzylite: a fuzzy logic control library, 2017. http://www.fuzzylite.com/.

[2] Made Sudarma & Ni Putu Sutramiani. The thinning zhang-suen application method in image of balinese scripts on the papyrus, 2014. Volume 91 - No 1.

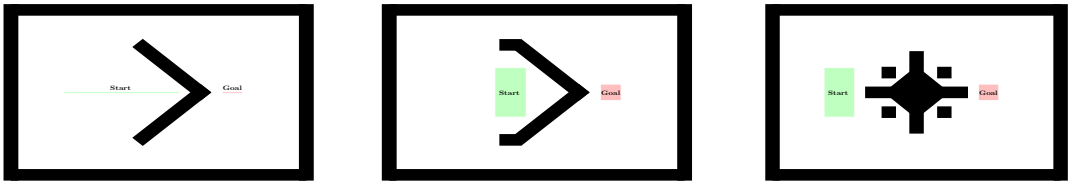
[3] Sumi Makito. A*(a-star) pathfinding algorithm in c++, 2018. https://github.com/SumiMakito/A-star.

Appendices

A Appendices

A.1 Test of Fuzzy Logic

The testing of the hypothesis *The robot is able to navigate around nonsimple obstacles* is tested in three maps, that is not shown in section 2. These are shown here:



(a) Environment: *Corner II* (b) Environment: *Curve* (c) Environment: *Star*

Figure 31: Environments used to test hypothesis for nonsimple obstacles

The implementation used to test the fuzzy logic is located in the project Github repository in the branch *fuzzy*:

`https://github.com/ancker1/RCA5-PRO/tree/fuzzy/robot_control`

The script **gazebo_spawn.sh** was developed to make it possible to spawn the robot in a given position and with a given orientation. This made it possible to automate the process of testing the fuzzy logic. Some of the interesting paths of the mobile robot using fuzzy logic can be seen here:

`https://github.com/ancker1/RCA5-PRO/tree/fuzzy/Test/Fuzzy`

A.2 Test of roadmaps

The implementation of the roadmaps and A* used in the test in the report is located in the project Github repository in branch *Roadmap_length_test_driver*:

`https://github.com/ancker1/RCA5-PRO/tree/Roadmap_length_test_driver/map_control`

The results from the tests can be seen here:

`https://github.com/ancker1/RCA5-PRO/tree/master/Test/Roadmap`

A.3 Test of Q-learning

The implementation of Q-learning used in the test of Q-learning is located in the project Github repository in branch *QLearning*:

`https://github.com/ancker1/RCA5-PRO/tree/QLearning/Q-Learning`

A.4 Test of computer vision

The implementation of the detection algorithms is located in the project Github repository in branch *master*:

`https://github.com/ancker1/RCA5-PRO/tree/master/robot_control`