Group 2 Final Project Report
ME-GY 6923 - SIMULATION TOOLS FOR MECHATRONICS AND ROBOTICS

MATLAB Simulation of Mobile Robot Navigation through a Maze

Almas Purisic Jagennath Hari Archit Sharma

Outline

Title: *MATLAB Simulation of Mobile Robot Navigation through a Maze*

Abstract

A major field in the study of robotics is mobile robots and their ability to complete localization and navigation objectives: reach point A from point B. To further develop this study, we utilized the simulation tools of MATLAB to demonstrate an autonomous, mobile robot can still reach its goal point in a randomly, unknown environment: a randomly auto-generated maze. To affirm this assumption, we used six different commonly known navigation methods which were Ultrasonic and Tactile Sensing, Bug 2 Algorithm, Probabilistic Roadmap Method, Rapidly-Exploring Random Tree, A*, and D* Search Navigation. From our procedure, we found that while all were sufficient in completing this experimentation, each algorithm excels in different aspects to complete navigation in a random environment.

Introduction

As civilization advances into the era of robotics, the field of study has begun to evolve into a multi-disciplinary area of research with many possibilities. From the perspective of the Institute of Electrical and Electronics Engineers (IEEE), a robot is widely accepted as an "autonomous machine capable of sensing its environment, carrying out computations to make decisions, and performing actions in the real world" [1]. Furthermore, a robot can be defined deeper by the capabilities and actions it can perform, such as mobility, medicine, manufacturing, and more. In our study, we wished to explore the potential mobile robots have in their ability to make navigational decisions. Robotic Localization and Navigation is a widely practiced area of research by professional and aspiring roboticists alike. However, most demonstrations and procedures are conducted repeatedly in an environment that the robot has been exposed to countless times in the trial and research phase. As the purpose of our methodology, we aspired to demonstrate that even in a new and/or randomly created environment, a mobile robot still has the attributes and control capabilities to be able to complete its actions in the real world. To affirm this, we utilized the tools provided by Mathworks in its widely-acclaimed software known as MATLAB. Within the simulation software, we created and executed six different areas of approach to prove our theory: the Bug 2 Algorithm, the Probabilistic Roadmap Method (PRM), Rapidly-Exploring Random Tree (RRT), A*, and D* Search Navigation, and an Ultrasonic and Tactile Sensing Algorithm that our group derived as well. To ensure that the environments that we created for this project were truly random, we called for the randomize maze function in MATLAB with multiple parameters and maze resolutions to demonstrate different degrees of difficulty a mobile robot could face in its navigation.

Procedure:

Theory and Mathematical Background

Before we began our procedure, we made assumptions about the environment generated. First, we assumed that out of all the mazes that we generated, there was a guaranteed method of reaching the goal point. Second, we expected the maze to be a closed environment, meaning that while the mobile robot navigates through the maze, no external variables such as inclement weather or outside hazards would affect the robot's ability to continue on its journey. And lastly, we assumed that with every maze randomly generated for the demonstration, each pathway has a fair value for traveling (no traps, trap doors, secret passageways, etc.). In regards to the motion capabilities of the robot, we expected our robot to follow the dynamics equations of a differential drive robot [2]. However, for our graphs and models shown, we used a marker in free space with a diameter of 1 meter for ease of comprehension. Alongside the actuators, we presumed that the sensor hardware, both tactile and ultrasonic, on the robot will be fully functional to detect obstacles one meter and five meters away respectively [3][4]. To conclude our analysis of the necessary mathematical concepts to begin our main procedure, we also correlated the relationship between matrix mapping on a grid for MATLAB (hence the software name's origin) to the navigation of a mobile robot to its endpoint.

Random Environment Generation

Before the methods of navigation can be executed, the proper environments must be generated. To accomplish this, we called the mapMaze() function with variable characteristics to create an algorithm in MATLAB to randomly generate all of the mazes we used for the procedure. Figures 1 and 2 are both examples of the mazes that were automatically generated for the navigation methods. As a supplement to our theory, we generated two versions of mazes: 'simple' and 'complex'. Figure 1 on the left represents a simple maze meanwhile Figure 2 represents a complex maze. The difference between the two was the Map Resolution attribute we called to zoom on a sector of the complex maze to create the simple maze as seen in Code Block M1 below (RRT method as the example, 10 for maze size).

```
generated_maze_for_RRT = mapMaze('MapSize',[10 10],'MapResolution',3)
```

Code Block M1

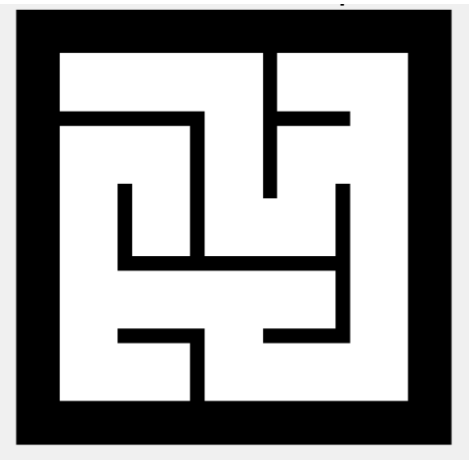

Figure 1: A 'simple' randomly generated maze

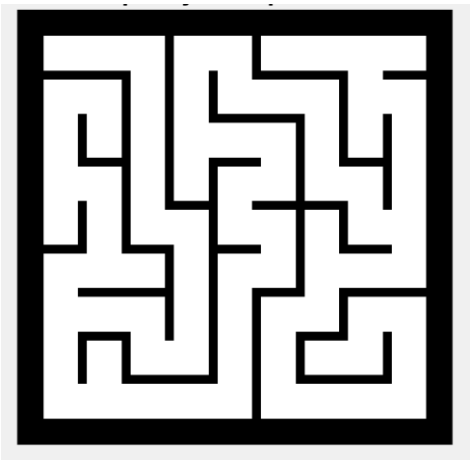

Figure 2: A 'complex' randomly generated maze

Ultrasonic and Tactile Sensing

On top of all the algorithms that MATLAB offers in its support libraries and documentation, my group self-derived our own algorithm assimilating both tactile and ultrasonic sensors to conduct maze navigation. Our purpose for this was to ensure robots built and operated at fundamental levels of navigation are still capable of control and approaching the goal point using only onboard sensors.

```
currentpos=round(ginput(1));
goal=round(ginput(1));
```

Code Block S1

To begin our first approach, we allowed for the user to arbitrarily choose where the starting point, A, and endpoint, B, will be within the maze by clicking any point in the maze's free space. 'currentpos' is the starting point and 'goal' is the endpoint with 'ginput' being where the user selects.

```
while(1>0) %creating an infinite loop
    num=randi([1 4],1,1); %calling a random number from 1-4
    if num==1 %if statement
        [currentpos,x,y]=leftUS(maploutinmatrix,currentpos,xtemp,ytemp);
    elseif num==2 %if statement
        [currentpos,x,y]=rightUS(maploutinmatrix,currentpos,xtemp,ytemp);
    elseif num==3 %if statement
        [currentpos,x,y]=upUS(maploutinmatrix,currentpos,xtemp,ytemp);
    elseif num==4 %if statement
        [currentpos,x,y]=downUS(maploutinmatrix,currentpos,xtemp,ytemp);
    end %end statement
    xtemp=[xtemp;x(end,:)]; %updates xtemp
    ytemp=[ytemp;y(end,:)]; %updates ytemp
```

Code Block S2

The premise of our algorithm was to then utilize a random motion loop from the 'num=randi' variable in line two. Each motion is recorded on the map by the if statements and updated by the 'temp' variables for x and y.

```
function [currentpos,xtemp,ytemp]=leftUS(maploutinmatrix,currentpos,xtemp,ytemp)
    for i=1:5
        if maploutinmatrix(currentpos(1,1)-1,30-currentpos(1,2)+1)==0
        lastpsotion=[currentpos(1,1) currentpos(1,2)];
        xtemp=[xtemp;currentpos(1,1)-1];
        ytemp=[ytemp;currentpos(1,2)];
        currentpos=[currentpos(1,1)-1 currentpos(1,2)];
        end
    end
end
```

Code Block S3

As seen in the Code Block S3 above, we later created a function for the ultrasonic sensor to move in the compass directions of North, South, East, and West to begin the robot's navigation using the variables we created earlier from Code Block S2. For the tactile sensors, we created Code Block S4 below with the same structure as ultrasonic by changing the variable names so it ran the tactile sensors' parameters solely.

```
function [currentpos,xtemp,ytemp]=left(maploutinmatrix,currentpos,xtemp,ytemp)
    if maploutinmatrix(currentpos(1,1)-1,30-currentpos(1,2)+1)==0
        lastpsotion=[currentpos(1,1) currentpos(1,2)];
        xtemp=[xtemp;currentpos(1,1)-1];
        ytemp=[ytemp;currentpos(1,2)];
        currentpos=[currentpos(1,1)-1 currentpos(1,2)];
    end
end
```

Code Block S4

As Code Blocks S1-S4 illustrates (the full algorithm shown in *MATLAB Code, Annotated* section below), we created an algorithm that uses what was random motion code to have a premise of navigation that allows the robot to record its path as it approaches the endpoint without colliding into obstacles. The parameters for both sensors ran lateral compass directions with only one difference: the ultrasonic sensor moves the robot five meters if no wall is detected while the tactile sensor moves only one meter. If a wall is detected by the ultrasonic sensor, the robot will continue to move until one meter from contact. If the tactile sensor reads one meter from contact or less, the loop passes and returns to the ultrasonic sensor to find a new direction to find the goal. As seen in Figure 3 below, the robot represented by the * icon has found and stopped at the goal point in the top left corner of the maze, which the starting point of the maze was the bottom right corner represented by the ⊕ icon.

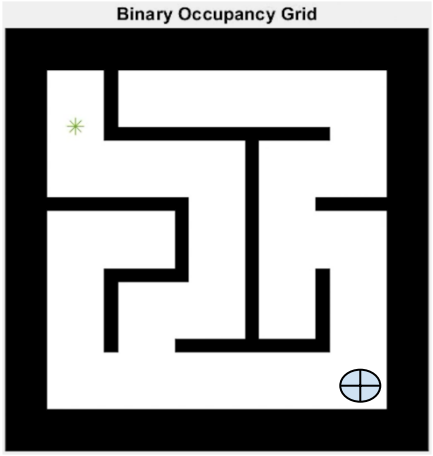

Figure 3: Simple maze solved with Ultrasonic and Tactile Sensing

Bug 2 Algorithm

The second approach we executed is the first of many common methods studied used for navigation: The Bug 2 Algorithm. Unlike the Bug 0 and Bug 1 Algorithm, Bug 2 has both memory and localization capabilities in its algorithm [5]. This algorithm produces an 'm-line' which is a direct path from the starting point to the goal point for the robot to follow. Obviously, in a maze, there will be plenty of obstacles in the way so the robot will then conduct wall following on the robot until it reaches the m-line again. This differs from Bug 0 and Bug 1 since Bug 0 does not work on every environment for the way it follows some obstacles repeatedly and Bug 1 will loop around a whole obstacle until its end to calculate the best path [5]. As seen in Code Block B1, we used the Bug 2 function in MATLAB to call the algorithm for the robot to solve in the randomly generated maze.

```
Tactilesensor=Bug2(mazeforfirstthree.occupancyMatrix);
```

Code Block B1

In Figure 3 below, the robot was successfully able to go from the starting position to the goal point, with some sidetracking as the wall follower leads the robot away from the goal as it attempts to find the m-line again (the full algorithm shown in *MATLAB Code, Annotated* section below).

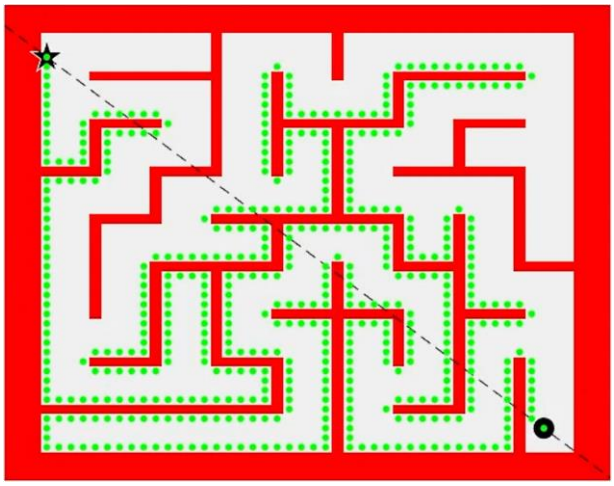

Figure 4: A complex maze solved with Bug 2

Probabilistic Roadmap Method (PRM)

The next algorithm we used for our methodology was the Probabilistic Roadmap Method, also known as the PRM algorithm. This function behaves in sparsely sampling the environment's free space and placing various nodes across the map [5]. When calling the function, as seen in Code Block P1, the user can determine the node amount they wish to calculate the path the robot takes while navigating the environment. Although, while the higher number of nodes creates a more optimal path for the robot to follow, the higher number of nodes creates more of a computational and runtime burden on the simulation. When the user selects or inputs the start and endpoints for the map, the points turn into nodes that are also used for the path.

```
prmComplex = mobileRobotPRM(mazefortactile,500);
```

Code Block P1

Figure 4 below shows the robot successfully navigating through the maze while being controlled and operated by the PRM algorithm. The legend to the figure goes as follows: The white regions of the maze show areas of the maze that is unreachable by the nodes; the blue regions represent potential paths the robot could have taken towards the goal; the blue dots represent nodes scattered across the maze for path building, and the red line is the optimal path or the 'road map' the PRM chose as the most optimal path to reach the endpoint. The PRM algorithm is effective in creating a path for a mobile robot to navigate through a maze, however as more nodes are used for better optimization, the simulation faces longer processing times and more computational power to run the algorithm.

Figure 5: A simple maze solved with PRM

Rapidly-Exploring Random Tree (RRT)

The fifth commonly used algorithm we examined for our theory was the Rapidly-Exploring Random Tree, or the RRT Algorithm. This method is also probabilistic in nature as it is an updated version from 2001 of the Random Tree algorithm [6]. Unlike Random Tree, RRT creates nodes in 'tree-like' structures from the farthest node from the starting point in contrast to any node deviating from the origin [6]. This updated version is more efficient since not nearly as many nodes are required to find a path. Instead of declaring the number of nodes like PRM, the user can control the variable distance between nodes to find the most optimal path. However, like PRM, the closer the distance between nodes is for optimization, the simulation must run more iterations and generations of nodes, which can become extensive on the runtime and simulation. For our experiment, we set the distance between nodes to be 0.5 meters as shown in Code Block R1 with the 'MaxConnectionDistance' variable. The 'ss' and 'sv' variables stand for both the state space and the map for the robot respectively (the full algorithm shown in *MATLAB Code, Annotated* section below).

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance =0.5;
```

Code Block R1

```
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-');
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2)
```

Code Block R2

We executed the RRT algorithm with the set parameters of Code Block R1 and the plot characteristics of Code Block R2 to generate the successful trial of Figure 6 below. The red line is the optimal path that the RRT algorithm found from the starting point in the bottom right corner to the goal point in the top left corner of the maze. As it can be seen, the algorithm took hundreds of node iterations, especially in the beginning regions, to be able to find the goal point in a simple maze.

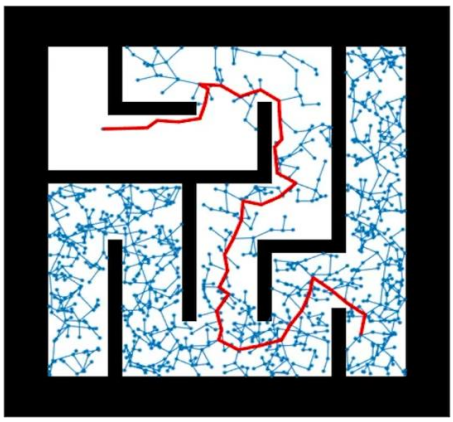

Figure 6: A simple maze solved with RRT

A* Search Navigation

Our procedure became more advanced as we analyzed more optimal methods of environment mapping and navigation such as the A* search algorithm. Like the Dijkstra Algorithm, A* Navigation operates with nodes being scattered across the environment's free space; however, instead of considering every node, each node is given a 'cost' in relation to its distance from the origin[5].

Cost = distance + heuristic value

The cost is determined by the physical distance the next node is from the current node the robot is stationed at or 'visiting'. The heuristic value is an A* function of the algorithm that calculates the cost of the next node's closest distance to the goal point. From the starting point, A* will have the robot examine all neighboring nodes and select the one with the shortest distance, or overall cost. As it navigates the maze, if it finds that a neighboring node is cheaper and can be reached by an already visited node, the algorithm will reevaluate its node path selection to maintain the lowest cost.

```
rng('default');
map = mapClutter;
planner = plannerAStarGrid(mazeforfirstthree);
```

Code Block A1

Examining Code Block A1, 'rng' creates a random number generator which mapClutter() then populates a randomly generated complex maze with number values at pseudo-nodes. Then, the third line of Code Block A1 calls the A* function to have the algorithm evaluate and optimize a path for the robot to follow to the goal point at the lowest cost. In Figure 7 below, the robot is initialized at the starting point in the bottom right corner of the maze with its endpoint being in the top left corner. The algorithm was successful as the brown line represents the path with the cheapest distance and cost for the robot to navigate with, while the tan regions are considered pathways and the white regions represent unfavorable node pathways (full algorithm shown in *MATLAB Code, Annotated* section).

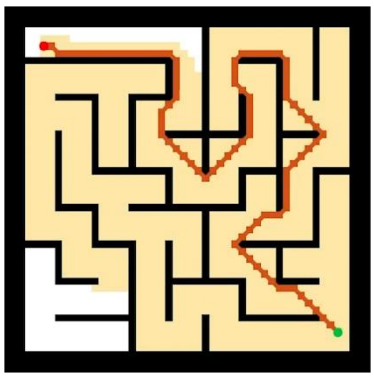

Figure 7: A complex solved with A* Search Algorithm

D* Search Navigation

Our final algorithm that we used to cross-examine our theory (alongside the other methods) was the D* Search Algorithm. Its name is derived from being a 'Dynamic A* Search'; meaning that although similar to A*, this algorithm plans the path's cost in reverse order: from goal to starting point. Also, while A* does reevaluate path costs as it visits neighboring nodes, D* is dynamic in reevaluating the robot's path relative to local changes in a dynamic environment as it processes its navigation as well. This allows D* to be more efficient than most other methods as it can reevaluate its incremental wavefront planner in environments, or mazes, that are more complex and unknown [5].

```
costoptimisation=Dstar(mazeforfirstthree.occupancyMatrix);
c=costoptimisation.costmap(goal);
costoptimisation.plan(goal);
costoptimisation.query(round(ginput(1)),'animate')
```

Code Block D1

Code Block D1 demonstrates the main components of the D* Algorithm (full algorithm shown in *MATLAB Code, Annotated* section below): Line 1 of D1 initializes the D* function on a randomly generated complex maze; Line 2 creates a cost map and optimizes a path cost from the goal point; Line 3 plans the optimized path the robot will take at the starting point, and Line 4 animates the path from wherever the user clicks on the maze to be the robot's starting point. Figure 8 illustrates Code Block D1 and Line 4 as the robot was successfully able to navigate through the randomly generated, complex maze. The green line represents the robot's optimized path with the algorithm processing the lowest cost from the darkest region to the robot's starting point, which is coincidentally the farthest and lightest region of the maze. Although not demonstrated in our procedure, the D* Algorithm can make adjustments to the path in its navigation if the maze is changing or dynamic. Another note to be aware of is unlike most of the other methods, D* Search is not dependent on node distances or node count so the runtime is exceptionally faster, especially with the complex maze model.

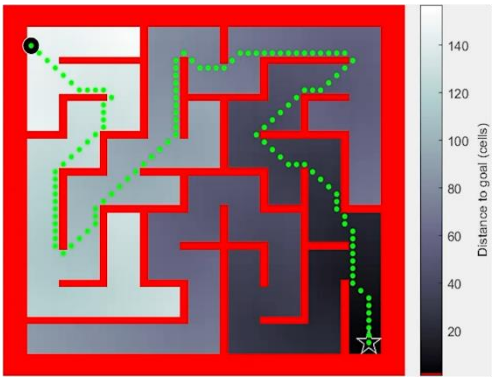

Figure 8: A complex maze solved with D* Search Algorithm

Results and Model Analysis

Analyzing all methods that we used for this project, the Ultrasonic and Tactile Sensing Algorithm my team derived ourselves was primitive and time-consuming, as it relied heavily on random motion to develop a degree of navigation to have the robot reach the goal point. While it was operational and successful, more development can be done to smooth out and optimize the robot's navigating performance. The Bug 2 algorithm guarantees the robot's arrival to the goal point in a complex maze with the collaboration of the algorithm's m-line and wall follower. Although, the wall follower will occasionally lead the robot to unnecessary obstacles and regions of the maze that other algorithms would not acknowledge in its navigation.

The Probabilistic Roadmap Method addresses this redundancy with its nodes system, allowing for a more optimal path. However, to efficiently navigate the robot through a complex maze, many nodes need to be declared in the initialization process and a long runtime for the simulation to consider each iteration. The constraint also slightly affected the Rapidly-Expanding Random Tree Algorithm but instead of node count, a variable distance between nodes has to be chosen carefully to ensure the path generated for the robot is still considered optimal, especially in complex mazes. For our experiments, we used simple mazes for both of these methods to display the concepts with easier comprehension. If the complex maze function were to be called, more iterations would be necessary and the simulation process would run for much longer.

To avoid these doubts, we also evaluated the advantages of using A* and D* Search Navigation as control methods for our robot to solve the random mazes. The A* Search Algorithm does find an optimal path from the starting point of the robot to the goal by its cost calculation function. The runtime for this algorithm relies on the complexity of the randomly generated maze as the method has to analyze the cost value of the physical distance and the heuristic value of each node to the goal. On the other hand, The D* Search Algorithm operates in reverse order from the goal point to the robot's starting point. This provides an advantage as D* Navigation increments cost value from the goal to the nodes neighboring by all the way to the starting point. Furthermore, the runtime of this method is exceptionally faster than most and adjusts to local changes in an environment; although, the assumption was made in the beginning that all randomly generated mazes will be a closed environment.

[For cross-comparison of all six models, plots, and their pertinent pathways, please refer to the *Appendix* section at the end as well.]

Conclusion

As seen through our methodology, areas of research, procedure, and analysis, a mobile robot is capable of navigating through a randomly-generated maze environment, simple and complex alike. All six of the methods we used to reach this conclusion supplemented this theory-especially the primitive algorithm we built using only Ultrasonic, Tactile, and Kinematic logic. The Ultrasonic and Tactile Sensing and Bug 2 Methods were both fundamental in nature and allowed for many redundancies in its path as it navigated simple mazes to reach the goal point. The Probabilistic Roadmap Method and Rapidly-Expanding Random Tree method did eliminate a plethora of these redundancies in their trials and executions; however, both methods' optimization abilities were constrained by either node count or distance between nodes, which can incapacitate the simulation's runtime and processing ability. To avoid these dependencies, A* and D* were both experimented with within complex mazes to demonstrate advanced methods to optimize a robot's navigation through complex environments. Nevertheless, all six methods were successful in their study, and all were sufficient in demonstrating that they can be utilized to navigate a robot through an unknown environment. As the field of mobile robotics continues to advance, more physical support and digital toolboxes will be provided to supplement all six of these algorithms' current capabilities of guiding and controlling a mobile robot further on.

References

[1] Spectrum, IEEE. “What Is a Robot?” *ROBOTS*, IEEE Spectrum, 28 May 2020, https://robots.ieee.org/learn/what-is-a-robot/.

[2] MathWorks. “Plan Path for a Differential Drive Robot in Simulink.” *Plan Path for a Differential Drive Robot in Simulink - MATLAB & Simulink*, 2019, https://www.mathworks.com/help/robotics/ug/plan-path-for-a-differential-drive-robot-in-simulink.html.

[3] Mathworks. “Ultrasonic.” *Connection to Ultrasonic Sensor on Arduino Hardware - MATLAB*, 2019, https://www.mathworks.com/help/supportpkg/arduinoio/ref/arduinoio.ultrasonic.html.

[4] Mathworks. “Mytouchsensor.” *Read Touch Sensor Value - MATLAB*, 2019, https://www.mathworks.com/help/supportpkg/legomindstormsev3io/ref/readtouch.html.

[5] Kopman, V. “ ME-GY 6923 Simulation Tools for Robotics - Lecture 12.” Professor V. Kopman, Sept. 2021.

[6] Becker, Aaron. *RRT, RRT* & Random Trees. YouTube*, YouTube, 21 Nov. 2018, https://www.youtube.com/watch?v=Ob3BIJkQJEw&t=30s. Accessed 22 Dec. 2021.

MATLAB Code, Annotated

```
%Generating a complex maze by calling function

mazeforfirstthree=randommaze_for_firstthree();  %calling the complex maze function
show(mazeforfirstthree) %displaying the maze in a figure
title("Hopefully, a complex maze","FontSize",18); %title and font size
axis off %removing axis

%%
%Generating a simple maze by calling function

mazefortactile=randommaze_for_RRT(); %calling the simple maze function
show(mazefortactile) %displaying the maze in a figure
title("Haha!!! This is less complex","FontSize",18); %title and font size
axis off %removing axis

%%
% Simulation of self designed tactile and Ultrasonic sensor

mapmatrix=mazefortactile.occupancyMatrix; %gets the binary occupancy matrix
maploutinmatrix=transpose(mapmatrix); %transposes the matrix and matrix gets flipped while converting from map to matrix
maps=binaryOccupancyMap(mapmatrix); %recreates the map using the obtained matrix wihout transpose to infalte the size so that it can be
implented in cartesian space
show(maps) %opens the map(maze)
currentpos=round(ginput(1)); %user input of start positon by clicking the point on the maze using the left mouse button
goal=round(ginput(1));        %user input of goal positon by clicking the point on the maze using the left mouse button
xtemp=[currentpos(1,1)]; %assigns xtemp
ytemp=[currentpos(1,2)]; %assigns ytemp
lastpsotion=currentpos; %inital value setting
while(1>0) %creating an infinite loop
   num=randi([1 4],1,1); %calling a random number from 1-4
   if num==1 %if statement
      [currentpos,x,y]=leftUS(maploutinmatrix,currentpos,xtemp,ytemp);    %assinged to 1 and calls the leftUS function
   elseif num==2 %if statement
      [currentpos,x,y]=rightUS(maploutinmatrix,currentpos,xtemp,ytemp); %assinged to 2 and calls the rightUS function
   elseif num==3 %if statement
      [currentpos,x,y]=upUS(maploutinmatrix,currentpos,xtemp,ytemp);     %assinged to 3 and calls the uptUS function
   elseif num==4 %if statement
      [currentpos,x,y]=downUS(maploutinmatrix,currentpos,xtemp,ytemp); %assinged to 4 and calls the downUS function
   end %end statement
   xtemp=[xtemp;x(end,:)]; %updates xtemp
   ytemp=[ytemp;y(end,:)]; %updates ytemp
   show(maps) %opens the maps(maze)
   axis off %turns off axis
   hold on %hold on
   plot(xtemp(end),ytemp(end),Marker="*",MarkerSize=10) %plots the current position of robot as a *
   plot(xtemp,ytemp,Marker=".") %plots the past location of robot as .
   delete(findobj(gca, 'Marker', '.')) %destroys all . markers so that only current positon along with line is destroyed
   pause(0.1) %adds a delay of 0.1 secs
   num=randi([1 4],1,1); %calling a random number from 1-4
   if num==1 %if statement
      [currentpos,x,y]=left(maploutinmatrix,currentpos,xtemp,ytemp); %assinged to 1 and calls the left function
   elseif num==2 %if statement
      [currentpos,x,y]=right(maploutinmatrix,currentpos,xtemp,ytemp); %assinged to 2 and calls the right function
   elseif num==3 %if statement
      [currentpos,x,y]=up(maploutinmatrix,currentpos,xtemp,ytemp); %assinged to 3 and calls the up function
   elseif num==4 %if statement
      [currentpos,x,y]=down(maploutinmatrix,currentpos,xtemp,ytemp); %assinged to 4 and calls the down function
```

```
    end %end statement
    xtemp=[xtemp;x(end,:)]; %updates xtemp
    ytemp=[ytemp;y(end,:)]; %updates ytemp
    show(maps) %opens the maps(maze)
    axis off %turns off axis
    hold on %hold on
    plot(xtemp(end),ytemp(end),Marker="*",MarkerSize=10) %plots the current position of robot as a *
    plot(xtemp,ytemp,Marker=".") %plots the past location of robot as .
    delete(findobj(gca, 'Marker', '.')) %destroys all . markers so that only current positon along with line is destroyed
    pause(0.1) %adds a delay of 0.1 secs
    if xtemp(end)==goal(1,1) && ytemp(end)==goal(1,2) %checks if goal is reached
        break; %break statement
    end %end statement
end %end statement
hold off %turning off hold

%%
%Finding start and goal points for complex maze if user doesnt want to
%click points on the maze by interative point clicker
binarymatrix=mazeforfirstthree.occupancyMatrix; %gets the binary matrix for the complex maze
[m, n]=size(binarymatrix); %gets the size of the matrix using m and n as outputs
for ii=m:-1:1 %reverse for loop for m to 1 for start location
    for jj=n:-1:1 %reverse nested for loop for n to 1
        if binarymatrix(ii,jj)==0 %if statement to check whether a certain point it is 0
            start=[ii jj]; %assigns start location
            break %break statement for innner loop
        end %end statement
    end %end statement
    if binarymatrix(ii,jj)==0 %if statement to check whether a certain point it is 0
        break %break statement for outer loop
    end %end statement
end %end statement

for ii=1:m % for loop for m to 1 for end location
    for jj=1:n % for loop for m to 1 for end location
        if binarymatrix(ii,jj)==0 %if statement to check whether a certain point it is 0
            goal=[ii jj]; %assigns goal location
            break %break statement for innner loop
        end %end statement
    end %end statement
    if binarymatrix(ii,jj)==0 %if statement to check whether a certain point it is 0
        break %break statement for outer loop
    end %end statement
end %end statement

%%
%Bug 2 algorithm implementation in a complex maze

show(binaryOccupancyMap(mazeforfirstthree.getOccupancy)) %opens the maze for the bug2 algorithim
Tactilesensor=Bug2(mazeforfirstthree.occupancyMatrix); %adds algorithm to the maze
Tactilesensor.query(round(ginput(1)),round(ginput(1)),'animate') %given start and goal locations by user input, animates the maze
title("Bug 2 Algorithm Path (Not Fun))","FontSize",18); %adds title
axis off %turns off axis

%%
% D star implementation for complex maze

show(binaryOccupancyMap(mazeforfirstthree.getOccupancy)) %opens the maze for D*
goal=round(ginput(1)) %takes user input for goal by left mouse click
costoptimisation=Dstar(mazeforfirstthree.occupancyMatrix); %creates a costmap by adding dstar function to the maze
c=costoptimisation.costmap(goal); %optimises the cost by giving the goal
```

```
costoptimisation.plan(goal); %plans the cost by giving goal
costoptimisation.niter; %displays the interation number
costoptimisation.query(round(ginput(1)),'animate') %displays path given input for start location by user click on maze
title("Yep, this is fast!","FontSize",18); %title heading
axis off %axis off

%%
%A star implementation for complex maze

rng('default'); %random number generator
map = mapClutter; %using map clutter command creates a map
planner = plannerAStarGrid(mazeforfirstthree); %creates a star grid using input from the complex maze
show(planner) %shows the planner
plan(planner,round(ginput(1)),round(ginput(1))); %plans path using input for start and goal using user click on maze
show(planner) %opens planner agian
title("If only we could make it better...","FontSize",18); %adds title
axis off %turns off axis

%%
% RRT for less complex maze

ss = stateSpaceSE2; %creates a state space object
sv = validatorOccupancyMap(ss); %makes a validator occupancy map with the state space
sv.Map=mazefortactile; %adds the simple maze to the sv object
sv.ValidationDistance = 0.2; %adds the validation distance set by trail and error method
ss.StateBounds = [mazefortactile.XWorldLimits;mazefortactile.YWorldLimits; [-pi pi]]; %creates the state bounds by assigning them as the
world limits and [-180 180] range
planner = plannerRRT(ss,sv);        %creates a RRT planner
planner.MaxConnectionDistance =0.5; %creates the max connection distance for the planner
show(mazefortactile) %opens the simple maze
rng(70,'twister'); % for repeatable result
[pthObj,solnInfo] = plan(planner,[round(ginput(1)) 0],[round(ginput(1)) 0]); %outputs path and solution by planning path for given inputs by
user clicks on map for start and goal and no roation
hold on %turns on hold
plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-');      % tree expansion
plot(pthObj.States(:,1),pthObj.States(:,2),'r-','LineWidth',2) % draw path
hold off %turns off hold
title("Fancy trees for a simple maze","FontSize",18); %adds title
axis off %axis off

%%
% PRM for complex/simple maze

prmComplex = mobileRobotPRM(mazefortactile,500); %creates a PRM for simple maze with 500 nodes
show(mazefortactile) %shows the maze
show(prmComplex); %shows the prmComplex
path = findpath(prmComplex, round(ginput(1)), round(ginput(1))); %finds the path by taking user input by mouse click for start and goal
locations
show(prmComplex); %opens the PRM complex object
title("Lame, could be more optimal.","FontSize",18); %creates a title
axis off %turns off axis

%%
% All functions

%--------------------------------------------------------------------------
%function for complex maze for D*/A*/Bug2 algorithim
%inbuilt matlab function which creates a binary occupancy matrix for a maze by giving parmeters
%parmeters given here was only the map size and only 10X10 so that,
%complexity and computation power is balaced.
%--------------------------------------------------------------------------
```

```
function generated_maze_for_firstthree=randommaze_for_firstthree() %output is generated_maze_for_firstthree and no input needed
   generated_maze_for_firstthree = mapMaze('MapSize',[10 10]);      %mapMaze() is the inbuilt function
end %function end statement
%-----------------------------------------------------------------------

%-----------------------------------------------------------------------
%function for simple maze for RRT/PRM/custom tactile and Ultrasonic sensors
%inbuilt matlab function which creates a binary occupancy matrix for a maze by giving parmeters
%parmeters given here was only the map size and only 10X10 and map resoultion was set to 3,
%complexity is lowered and computation power is increased for complex and time consuming alogorithims.
%-----------------------------------------------------------------------
function generated_maze_for_RRT=randommaze_for_RRT() %output is generated_maze_for_firstthree and no input needed
   generated_maze_for_RRT = mapMaze('MapSize',[10 10],'MapResolution',3); %mapMaze() is the inbuilt function
end %function end statement
%-----------------------------------------------------------------------

%-----------------------------------------------------------------------
%function for simple maze for left side movement using Ultrasonic sensor
%Compares the binary occupancy matirx(maze) so if a 5 grid movement is-
%possible it will execute it, else it will reach a grid right before it is
%occupied.
%-----------------------------------------------------------------------
function [currentpos,xtemp,ytemp]=leftUS(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and
inputs are the binarymatrix,currentpos,xtemp,ytemp
   for i=1:5 %initalises for loop running 5 iterations
      if maploutinmatrix(currentpos(1,1)-1,30-currentpos(1,2)+1)==0 %checks if left position is occupied or not
         lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position in consideration
         xtemp=[xtemp;currentpos(1,1)-1]; %xtemp gets updated for each iteration
         ytemp=[ytemp;currentpos(1,2)]; %ytemp gets updated for each iteration
         currentpos=[currentpos(1,1)-1 currentpos(1,2)]; %current position gets updated for each iteration
      end %end statement
   end %end statement
end %end statement
%-----------------------------------------------------------------------

%-----------------------------------------------------------------------
%function for simple maze for Right side movement using Ultrasonic sensor
%Compares the binary occupancy matirx(maze) so if a 5 grid movement is-
%possible it will execute it, else it will reach a grid right before it is
%occupied.
%-----------------------------------------------------------------------
function [currentpos,xtemp,ytemp]=rightUS(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and
inputs are the binarymatrix,currentpos,xtemp,ytemp
   for i=1:5 %initalises for loop running 5 iterations
      if maploutinmatrix(currentpos(1,1)+2,30-currentpos(1,2)+1)==0 %checks if right position is occupied or not
         lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position into consideration
         xtemp=[xtemp;currentpos(1,1)+1]; %xtemp gets updated for each iteration
         ytemp=[ytemp;currentpos(1,2)]; %ytemp gets updated for each iteration
         currentpos=[currentpos(1,1)+1 currentpos(1,2)]; %current position gets updated for each iteration
      end %end statement
   end %end statement
end %end statement
%-----------------------------------------------------------------------

%-----------------------------------------------------------------------
%function for simple maze for Upward movement using Ultrasonic sensor
%Compares the binary occupancy matirx(maze) so if a 5 grid movement is-
%possible it will execute it, else it will reach a grid right before it is
%occupied.
%-----------------------------------------------------------------------
```

```
function [currentpos,xtemp,ytemp]=upUS(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and
inputs are the binarymatrix,currentpos,xtemp,ytemp
  for i=1:5 %initalises for loop running 5 iterations
    if maploutinmatrix(currentpos(1,1),30-currentpos(1,2)-1)==0 %checks if up position is occupied or not
      lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position into consideration
      xtemp=[xtemp;currentpos(1,1)]; %xtemp gets updated for each iteration
      ytemp=[ytemp;currentpos(1,2)+1]; %ytemp gets updated for each iteration
      currentpos=[currentpos(1,1) currentpos(1,2)+1]; %current position gets updated for each iteration
    end %end statement
  end %end statement
end %end statement
%---------------------------------------------------------------------

%---------------------------------------------------------------------
%function for simple maze for downward movement using Ultrasonic sensor
%Compares the binary occupancy matirx(maze) so if a 5 grid movement is-
%possible it will execute it, else it will reach a grid right before it is
%occupied.
%---------------------------------------------------------------------
function [currentpos,xtemp,ytemp]=downUS(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and
inputs are the binarymatrix,currentpos,xtemp,ytemp
  for i=1:5 %initalises for loop running 5 iterations
    if maploutinmatrix(currentpos(1,1),30-currentpos(1,2)+2)==0 %checks if down position is occupied or not
      lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position into consideration
      xtemp=[xtemp;currentpos(1,1)]; %xtemp gets updated for each iteration
      ytemp=[ytemp;currentpos(1,2)-1]; %ytemp gets updated for each iteration
      currentpos=[currentpos(1,1) currentpos(1,2)-1]; %current position gets updated for each iteration
    end %end statement
  end %end statement
end %end statement
%---------------------------------------------------------------------

%---------------------------------------------------------------------
%function for simple maze for left side movement using tactile sensor
%Compares the binary occupancy matirx(maze) so if a 1 grid movement is-
%possible it will execute it, else it will not execute.
%---------------------------------------------------------------------
function [currentpos,xtemp,ytemp]=left(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and
inputs are the binarymatrix,currentpos,xtemp,ytemp
  if maploutinmatrix(currentpos(1,1)-1,30-currentpos(1,2)+1)==0 %checks if left position is occupied or not
    lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position into consideration
    xtemp=[xtemp;currentpos(1,1)-1]; %xtemp gets updated
    ytemp=[ytemp;currentpos(1,2)]; %ytemp gets updated
    currentpos=[currentpos(1,1)-1 currentpos(1,2)]; %current position gets updated
  end %end statement
end %end statement
%---------------------------------------------------------------------

%---------------------------------------------------------------------
%function for simple maze for right side movement using tactile sensor
%Compares the binary occupancy matirx(maze) so if a 1 grid movement is-
%possible it will execute it, else it will not execute.
%---------------------------------------------------------------------
function [currentpos,xtemp,ytemp]=right(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and
inputs are the binarymatrix,currentpos,xtemp,ytemp
  if maploutinmatrix(currentpos(1,1)+2,30-currentpos(1,2)+1)==0 %checks if right position is occupied or not
    lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position into consideration
    xtemp=[xtemp;currentpos(1,1)+1]; %xtemp gets updated
    ytemp=[ytemp;currentpos(1,2)]; %ytemp gets updated
    currentpos=[currentpos(1,1)+1 currentpos(1,2)]; %current position gets updated
  end %end statement
```

```
end %end statement
%-----------------------------------------------------------------------

%-----------------------------------------------------------------------
%function for simple maze for upward movement using tactile sensor
%Compares the binary occupancy matirx(maze) so if a 1 grid movement is-
%possible it will execute it, else it will not execute.
%-----------------------------------------------------------------------
function [currentpos,xtemp,ytemp]=up(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and inputs
are the binarymatrix,currentpos,xtemp,ytemp
   if maploutinmatrix(currentpos(1,1),30-currentpos(1,2)-1)==0 %checks if up position is occupied or not
      lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position into consideration
      xtemp=[xtemp;currentpos(1,1)]; %xtemp gets updated
      ytemp=[ytemp;currentpos(1,2)+1]; %ytemp gets updated
      currentpos=[currentpos(1,1) currentpos(1,2)+1]; %current position gets updated
   end %end statement
end %end statement
%-----------------------------------------------------------------------

%-----------------------------------------------------------------------
%function for simple maze for downward movement using tactile sensor
%Compares the binary occupancy matirx(maze) so if a 1 grid movement is-
%possible it will execute it, else it will not execute.
%-----------------------------------------------------------------------
function [currentpos,xtemp,ytemp]=down(maploutinmatrix,currentpos,xtemp,ytemp) %function outputs are currentpos,xtemp,ytemp and
inputs are the binarymatrix,currentpos,xtemp,ytemp
   if maploutinmatrix(currentpos(1,1),30-currentpos(1,2)+2)==0 %checks if down position is occupied or not
      lastposition=[currentpos(1,1) currentpos(1,2)]; %takes the last position into consideration
      xtemp=[xtemp;currentpos(1,1)]; %xtemp gets updated
      ytemp=[ytemp;currentpos(1,2)-1]; %ytemp gets updated
      currentpos=[currentpos(1,1) currentpos(1,2)-1]; %current position gets updated
   end %end statement
end %end statement
%-----------------------------------------------------------------------
```

Appendix

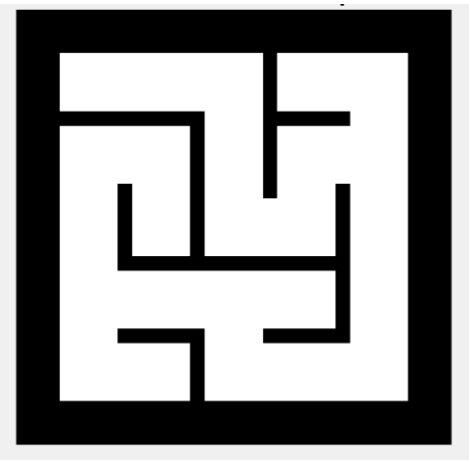

Figure 1: A ‘simple’ randomly generated maze

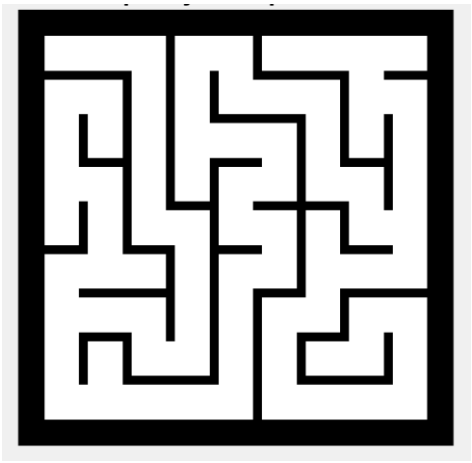

Figure 2: A ‘complex’ randomly generated maze

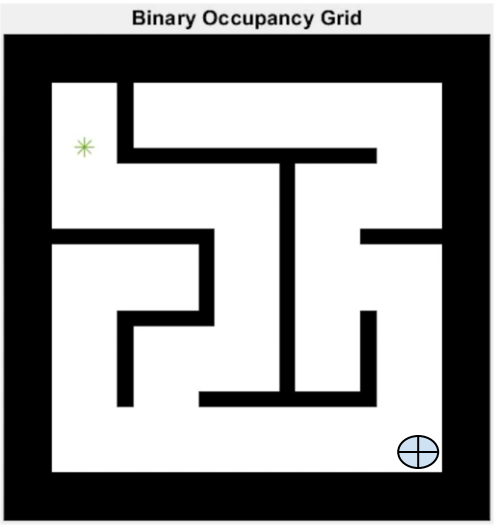

Figure 3: Simple maze solved with Ultrasonic and Tactile Sensing

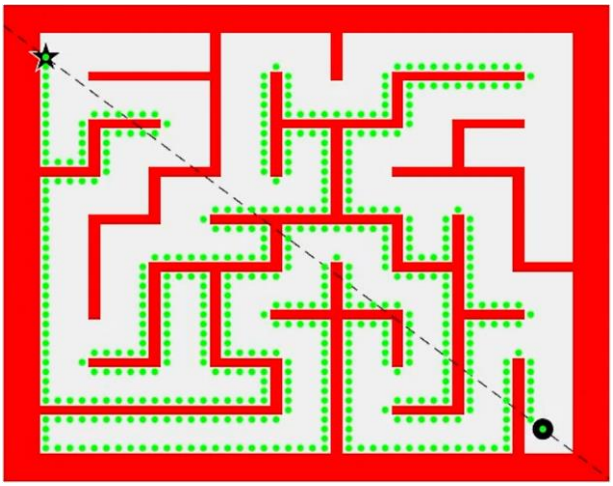

Figure 4: A complex maze solved with Bug 2

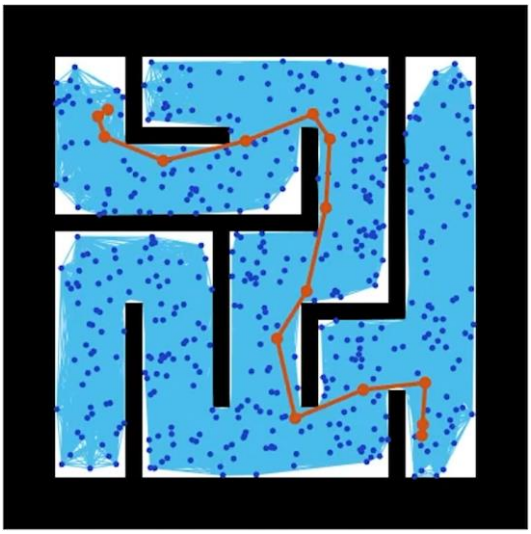

Figure 5: A simple maze solved with PRM

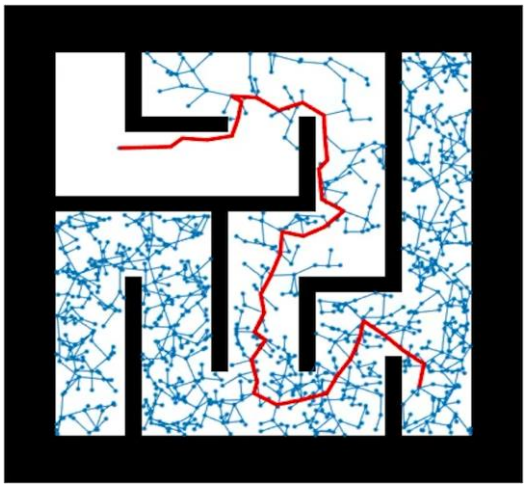

Figure 6: A simple maze solved with RRT

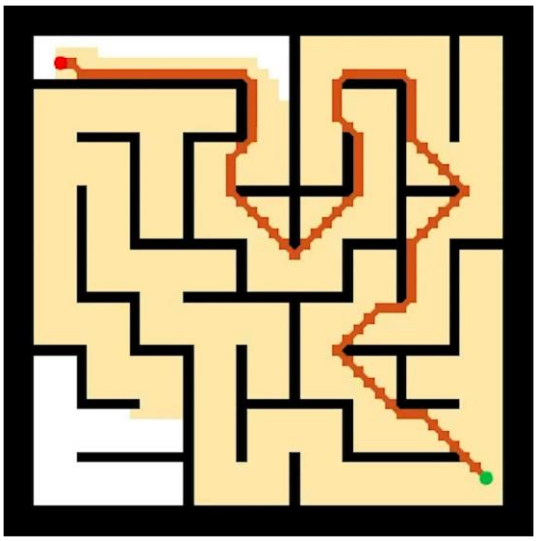

Figure 7: A complex solved with A* Search Algorithm

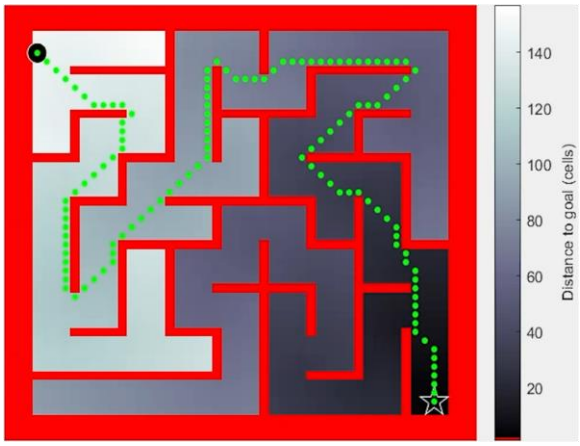

Figure 8: A complex maze solved with D* Search Algorithm