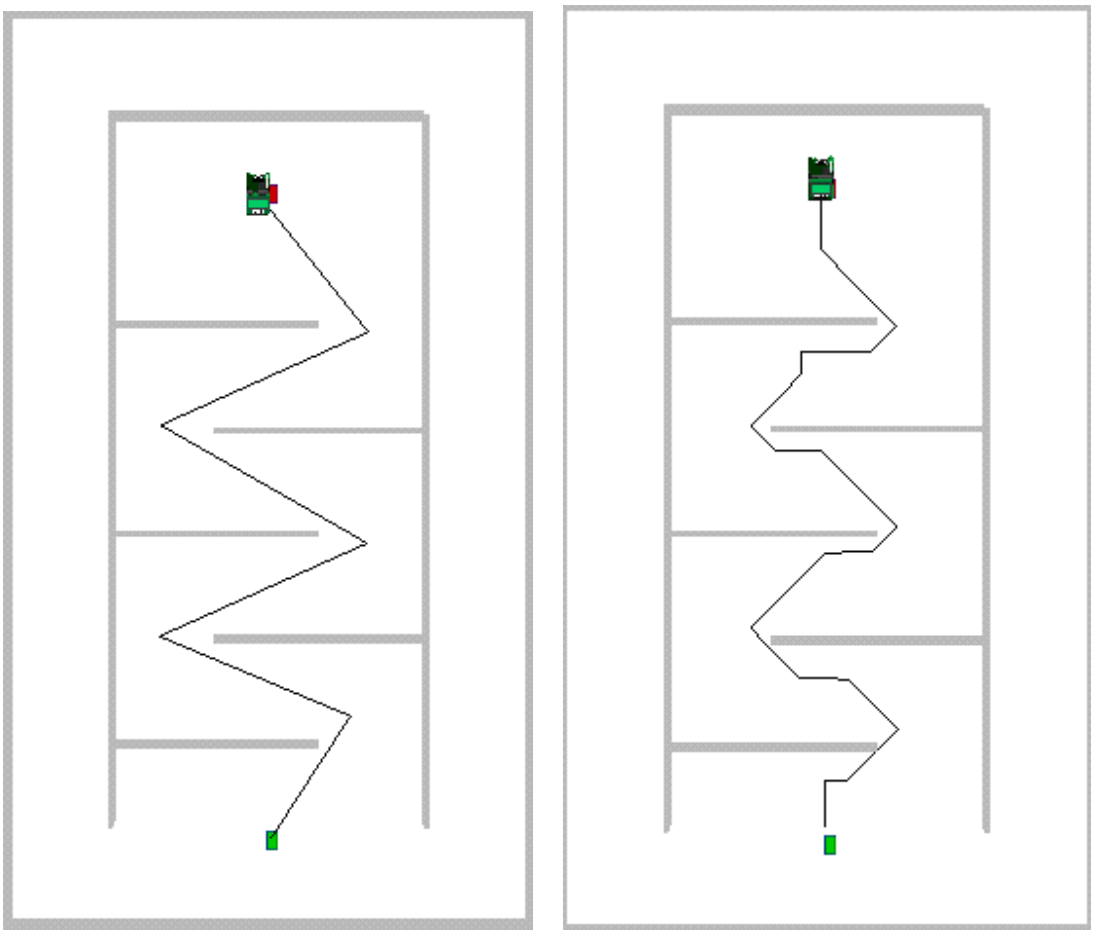
A Practical Comparison of Robot Path Planning Algorithms given only Local Information

James Ng



Supervisor: Associate Professor Thomas Bräunl

Centre for Intelligent Information Processing Systems

School of Electrical and Electronic Engineering

The University of Western Australia

Acknowledgements

First and foremost, I would like to thank Thomas Bräunl for his support, guidance and direction both before and during this project. Given my honours result and the first programming test, I was definitely not the most ideal postgraduate student. However, Thomas has given me a second chance and I am most grateful for that. During the project, Thomas has always provided frank assessments, constructive feedback and criticisms of the more controversial results, all of which are greatly appreciated.

I would also like to thank Gary Bundell for extending the PhD scholarship by 6 months which allowed this project to be covered. I am also grateful for the second chance he has given me.

I would also like to my family for their support and advice this year, especially during times of crisis.

Abstract

This thesis compares and analyses the practical aspects of path-planning and navigation algorithms for autonomous robots. The algorithms bug1, bug2, alg1, alg2, distbug, tangentbug and D* were implemented and simulated on the EyeSim simulator. For each algorithm, data was gathered about its relative complexity, memory requirements, path length, rotation, computation time and robustness against error.

Subsequent analysis shows that tangentbug and D* produce the shortest path length and lowest rotation. Surprisingly, tangentbug and D* also had very low average total computation times and D* was the most inexpensive algorithm on enclosed and open maps. D* was also the most robust algorithm due to its segmentation. Conversely, Bug1 and Bug2 were the simplest, required the least amount of memory and were very competitive on total computation time.

Overall, if a cheap, computationally inexpensive, low memory and simple solution is required, bug1 is recommended on enclosed maps otherwise bug2 should be chosen. If fast convergence, robustness and low computation time are desired attributes, D* should be chosen unless the map involves a local minimum. In that case, tangentbug should be chosen. If a compromise is sought, alg2 should be chosen on enclosed maps, otherwise pick distbug.

Table of Contents

1. Introduction

Imagine driving to work, walking to a colleague's office or simply taking a walk along the beach. This appears easy because humans navigate subconsciously, taking for granted their navigation abilities. However, writing a formal navigation procedure to replicate this is not easy.

It is envisaged that robots will someday replace human labour in menial tasks such as cleaning and driving [1]. Robocup has an ambitious goal that by 2050 a team of robot soccer players will defeat the human FIFA world champions [2]. These ambitious tasks all rely on successful navigation to move around in the real world. Therefore, successful navigation is a fundamental requirement for autonomous, independent robots.

This thesis examines several navigation algorithms using the EyeSim simulator. Whilst acknowledging the theory, the emphasis is on the practical aspects. In particular, implementation issues will be discussed and observable results used to draw key conclusions about the algorithm's practical usability.

Firstly, the objective is presented and the algorithms are introduced. Secondly, the implementation of each algorithm is discussed along with implementation problems encountered. Thirdly, results and analysis are presented for experiments conducted.

2. Objective

A 2-dimensional environment is set with a start and a goal. A finite number of arbitrarily shaped obstacles, each of finite area, are then placed in the environment. These obstacles cannot overlap the start or goal. The robot starts at the start and its objective is to find an obstacle-free, continuous path from start to the goal. Figure 2-1 shows sample environments with the green tile marking the start and the red tile marking the goal.

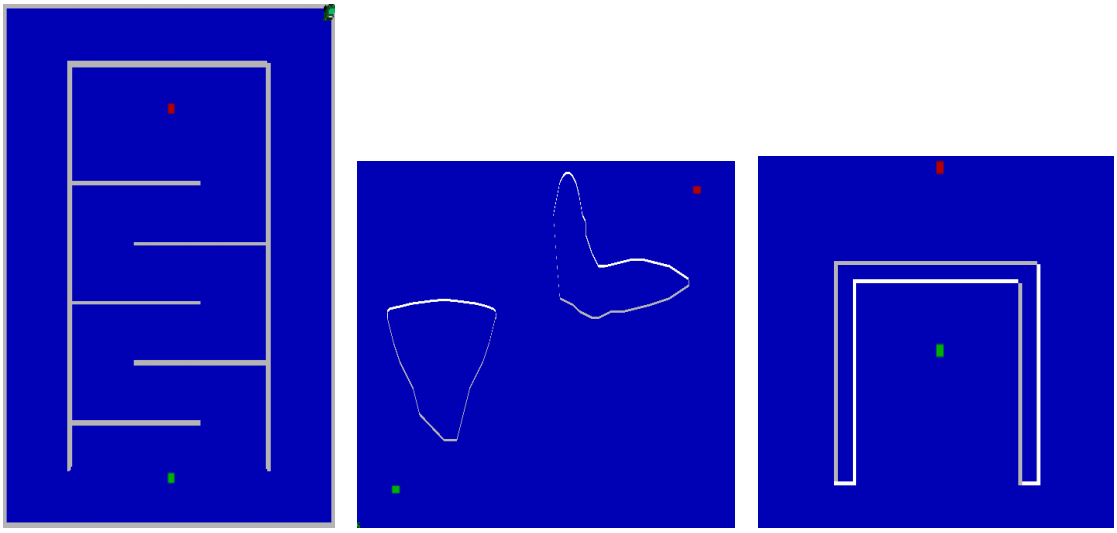


Figure 2-1 Sample navigation environments

Initially, the robot is only given the direction and distance of the goal. No knowledge about the map is given prior to starting. The robot is equipped with Position Sensitive Devices (PSDs) which return the distance to the nearest obstacle in its pointed direction. The robot is modeled as a point object with no size and is able to travel through gaps of any size.

A successful algorithm needs to be convergent as a prerequisite. That is, it needs to find a path to the goal if such a path exists. If no such path exists, it must stop and inform the user that the target is unreachable. If an algorithm is convergent, it is then assessed on the following attributes:

- Path Length. The distance of the path from start to finish. This should be as short as possible.
- Computation time. The algorithm's total execution time excluding time spent driving. This should be as short as possible and is driven by the following sub-attributes:
    - Number of calls to the math-library. A factor which affects computation time is the number of calls to the math library.
    - Computation time per metre traveled. Algorithms which have a short path length carry this advantage into computation time calculations. Calculating computation time per metre traveled removes this advantage.
- Rotation. The amount of turning which is performed along the path from start to finish. This should be as low as possible.
- Inherent rotation. Some rotation is hardware dependant and this is filtered out in this measurement.
- Robustness. The algorithm's ability to tolerate PSD error, linear driving error and rotational driving error. This should be as high as possible.
- Memory requirements. The amount of global memory reserved by the algorithm. This should be as low as possible.
- Simplicity. This is measured by the lines of code required for implementation. This should be as low as possible.

Attribute measurements are taken from the simulator or the source code. However, the algorithm implementation will undoubtedly influence the measurements. Therefore, it is imperative that the implementation be understood before the measurements can be used to draw general conclusions.

3. Convergent Algorithms

This section introduces algorithms which were implemented and simulated. All algorithms are proven convergent in their respective papers. Firstly, it is convenient to introduce notation common to all bug algorithms. These are:

- H_i – the ith hit point. This is the first point of contact between the robot and the ith obstacle.
- L_i – the ith leave point. This is the point at which the robot leaves the ith obstacle.
- S – the starting position.
- T – the goal position. Also called the target or finish.
- x – the robot's current position.
- $d(a, b)$ – the Euclidean distance between arbitrary points a and b.
- $d_{path}(a, b)$ – the robot's path length between arbitrary points a and b.
- r – the maximum range of the PSD sensors.
- $r(\theta)$ – the free-space in a given direction θ. This is the distance between the robot and the first visible obstacle in the direction θ.
- F – the free-space in the target's direction. It should be noted that $F = r(\theta)$ where θ is the target's direction.

3.1 The Bug1 Algorithm

The bug1 algorithm was the first bug algorithm created by Lumelsky and Stepanov in 1986 [3]. Bug1's statechart diagram is depicted in Figure 3-1 and it works as follows:

0) Initialize variable i to 0

1) Increment i and move toward the target until one of the following occurs:

- The target is reached. Stop
- An obstacle is encountered. Label this point H_i and proceed to step 2.

2) Keeping the obstacle on the right, follow the obstacle boundary. Whilst doing so, record the point where $d(x,T)$ is minimal. Label this point L_i. Do this until one of the following occurs:

- Point H_i is reached. Go to Step 3.
- The target is reached. Stop

3) Test whether the target is reachable. To do this, check if $H_i = L_i$. If so, the target is unreachable. Stop. Otherwise, choose the wall-following direction which minimises $d_{path}(H_i, L_i)$ and maneuver to L_i. At L_i , proceed to step 1.

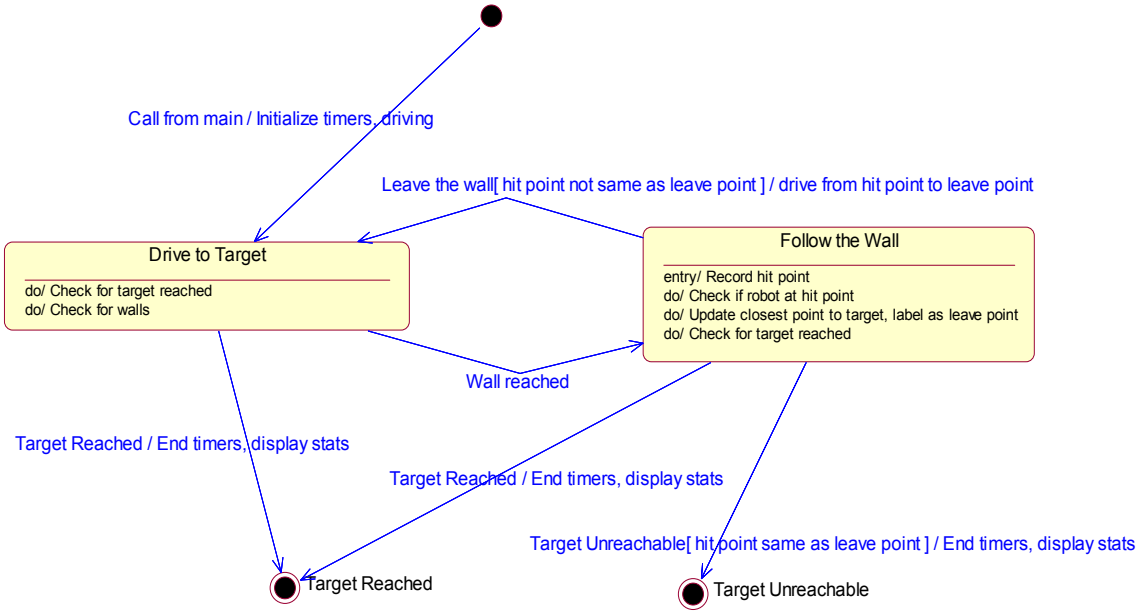


Figure 3-1 Bug1's statechart diagram

3.2 The Bug2 Algorithm

The bug2 algorithm was also created by Lumelsky and Stepanov in 1986 [3]. It is less conservative than bug1 because it introduces a new concept called the M line. Bug2's statechart diagram is depicted in Figure 3-2 and it works as follows:

0) Initially, plot an imaginary line, M, directly from start to target and initialise *i* to 0.
1) Increment *i* and follow the M line towards the target until either:

- The target is reached. Stop
- An obstacle is hit. Label this point H_i. Go to step 2

2) Keeping the obstacle on the right, follow the obstacle boundary. Do this until either:

- A point along M is found such that $d(x, T) < d(H_i, T)$. Label this point L_i. Go to step 1.
- The target is reached. Stop.
- The robot returns to H_i. The target is unreachable. Stop.

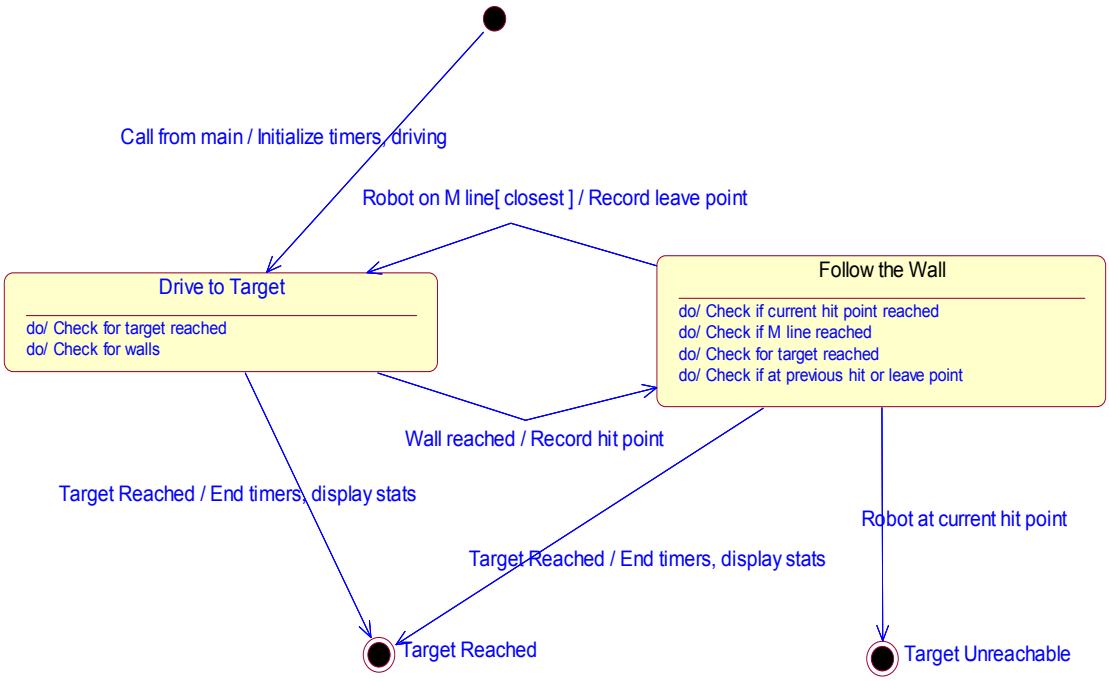


Figure 3-2 Bug2's statechart diagram

3.3 The Alg1 algorithm

The Alg1 algorithm is an extension of the Bug2 algorithm invented by Sankaranarayanan and Vidyasagar in 1990 [4]. Bug2's vulnerability is that it can trace the same path twice. To avoid this, Alg1 remembers previous hit and leave points and uses them to generate shorter paths. Alg1's statechart diagram is depicted in Figure 3-3 and it works as follows:

0) Initially, plot an imaginary line M directly from start to target and initialize i to 0.
1) Increment i and follow the M line toward the target until either:

- The target is reached. Stop
- An obstacle is hit. Define this point H_i. Go to step 2

2) Keeping the obstacle on the right, follow the obstacle boundary. Do this until one of the following occurs:

- The target is reached. Stop.
- A point y is found such that
    - it is on M
    - $d(y, T) < d(x, T)$ for all x ever visited by the robot along M and
    - The robot can move towards the target at y.

  Define this point L_i and go to step 1.
- A previously defined point H_j or L_j is encountered such that $j<i$. Turn around and return to H_i. When H_i is reached, follow the obstacle boundary keeping the wall on the left. This rule cannot be applied again until L_i is defined.
- The robot returns to H_i. The target is unreachable. Stop

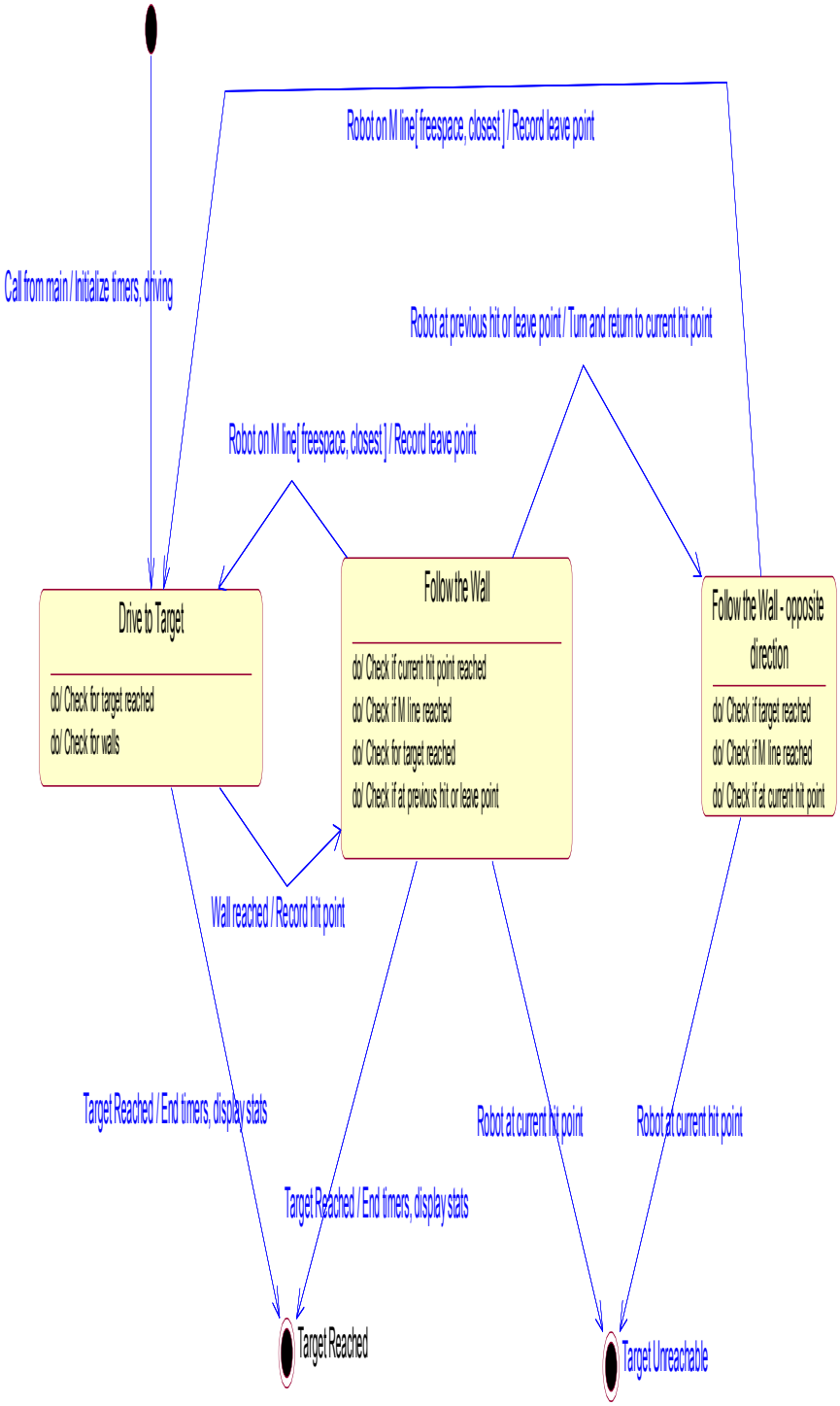


Figure 3-3 Alg1's statechart diagram

3.4 The Alg2 algorithm

The Alg2 algorithm is an improvement from the Alg1 algorithm invented by Sankaranarayanan and Vidyasagar in 1990 [5]. The robot abandons the M-line concept and a new leaving condition is introduced. Alg2's statechart diagram is depicted in Figure 3-4 and it works as follows:

0) Initialise $Q = d(S, T)$ and i to 0.

1) Increment i and proceed in the direction of the target whilst continuously updating Q to $d(x, T)$ if $Q < d(x, T)$. Q should now represent the closest the robot has ever been to the target. Do this until one of the following occurs:

- The target is reached. Stop
- An obstacle is encountered. Label this point H_i and proceed to step 2.

2) Keeping the obstacle on the right, follow the obstacle boundary whilst continuously updating Q to $d(x, T)$ if $Q < d(x, T)$ until one of the following occurs:

- The target is reached. Stop
- A point y is found such
    - that $y < Q$
    - The robot can move towards the target at y.

  Define this point L_i and proceed to step 1.
- A previously defined point H_j or L_j is encountered such that $j<i$. Return to H_i. When H_i is reached, follow the obstacle boundary keeping the wall on the left. This rule cannot be applied again until L_i is defined.
- The robot returns to H_i. The target is unreachable. Stop.

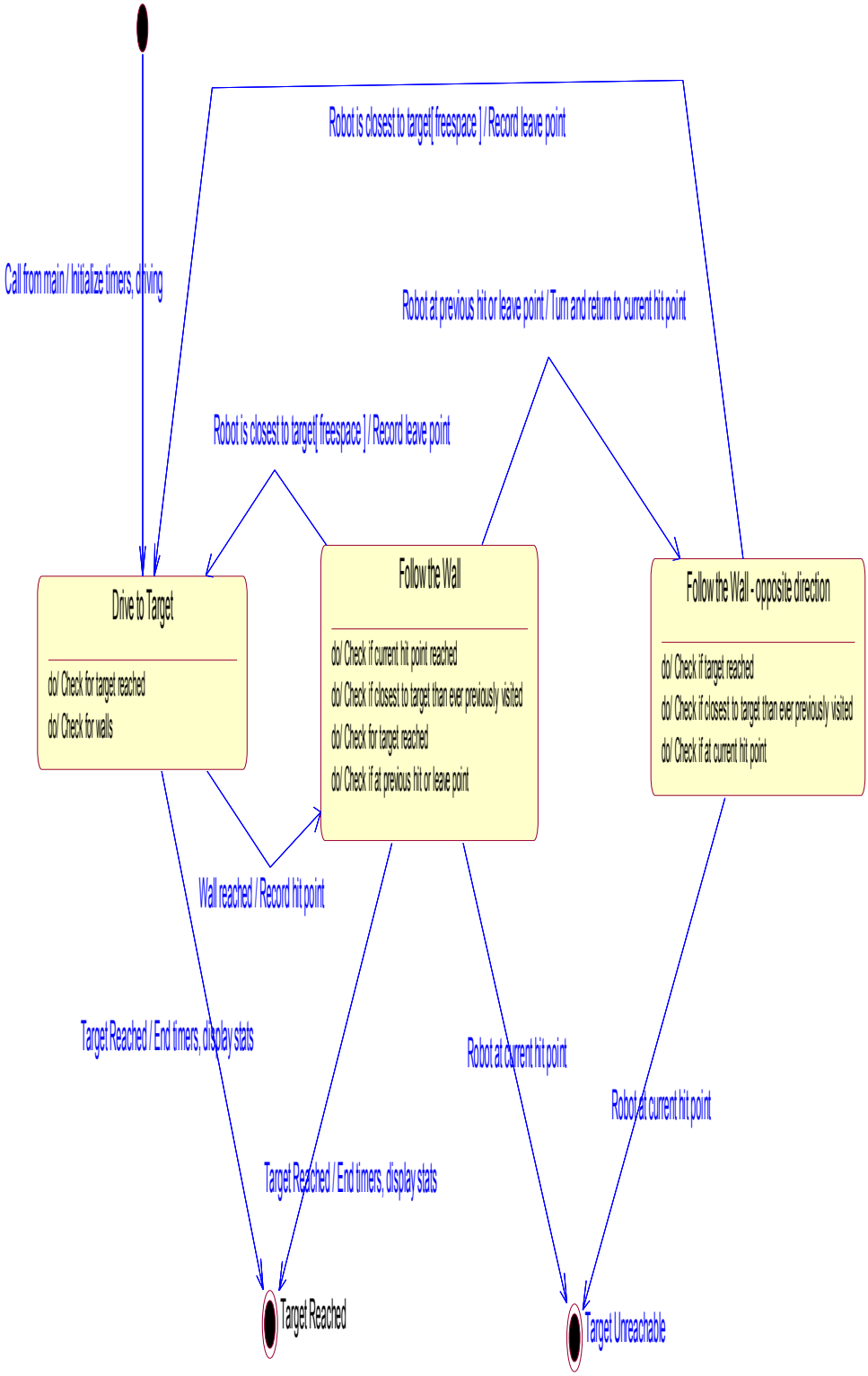


Figure 3-4 Alg2's statechart diagram

3.5 The Distbug algorithm

The distbug algorithm was invented by Kamon and Rivlin in 1997 [6]. It is very similar to the Alg2 algorithm because it has a near identical leaving condition, with just a small, subtle difference. See section 7.2 for more details. The only real difference is that distbug does not maintain a list of previous points whereas Alg2 does. Distbug's statechart diagram is depicted in Figure 3-5 and it works as follows:

0) Initialise i=0 and *Step* to the wall thickness.

1) Increment i and move toward the target until one of the following occurs:

- The target is reached. Stop.
- An obstacle is reached. Denote this point H_i. Go to step 2.

2) Turn left and follow the obstacle boundary whilst continuously updating the minimum value of *d(x, T)* and denote this value $d_{\min}(T)$.

Keep doing this until one of the following occurs:

- The target is visible: $d(x,T) - F \le 0$. Denote this point L_i. Go to step 1.
- The range based leaving condition holds: $d(x,T) - F \le d_{\min}(T) - Step$. Denote this point L_i. Go to step 1.
- The robot completed a loop and reached H_i. The target is unreachable. Stop.

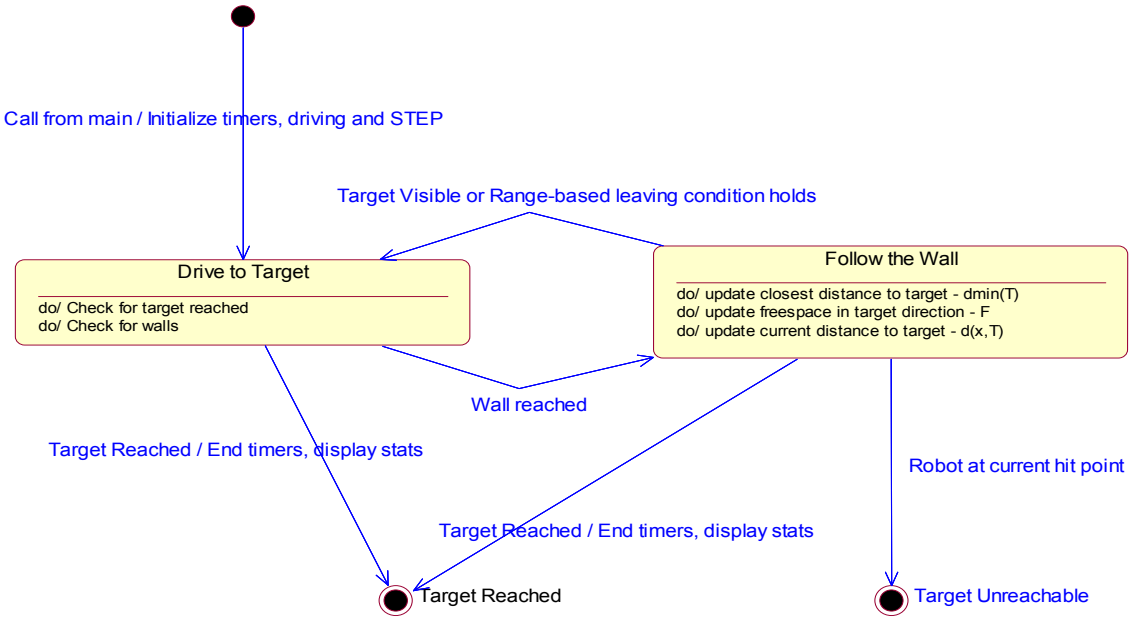


Figure 3-5 Distbug's statechart diagram

3.6 The tangentbug algorithm

The tangentbug algorithm was invented by Kamon, Rivlin and Rimon in 1995 [7]. Since then, tangentbug has been recognised as an algorithm which produces very short paths given purely local information [8].

3.6.1 The Global Tangent Graph

To understand how tangentbug works, consider the environment depicted in figure 3-6(a). Next, consider the convex vertices of all the obstacles which are circled orange in figure 3-6(b). Then, join each pair of non-obstructed vertices and include the start and target. The result is the global tangent graph and this is depicted in figure 3-6(c). It has been shown that the global tangent graph always contains the optimal path from start to finish [9]. As expected, figure 3-6(d) shows the optimal path for this particular map.

3.6.2 The Local Tangent Graph

However, the robot does not have global knowledge and tangentbug compensates by generating the local tangent graph (LTG). A sample LTG graph is shown in figure 3-7. The LTG is generated by firstly gathering data for the function $r(\theta)$ and F. $r(\theta)$ returns the distance to the first visible obstacle in a given direction θ. Then, $r(\theta)$ is processed according to the following rules:

- If $d(X,T) - F \leq 0$, the target is visible. Create a node, called T-node, on the target.
- If $F \geq r$, there are no visible obstacles in the target's direction. Create a T-node in the target's direction. This is illustrated by the T-node in figure 3-7.
- Check the function $r(\theta)$ for discontinuities. If a discontinuity is detected, create a node in θ's direction. This is illustrated by nodes 1, 2, 3 and 4 in figure 3-7.
- If $r(\theta) = r$ (the maximum PSD range) and $r(\theta)$ subsequently decreases create a node in θ's direction. This is illustrated by node 5 in figure 3-7. Similarly, if $r(\theta) \neq r$, and $r(\theta)$ subsequently increases such that $r(\theta) = r$, create a node in θ's direction.

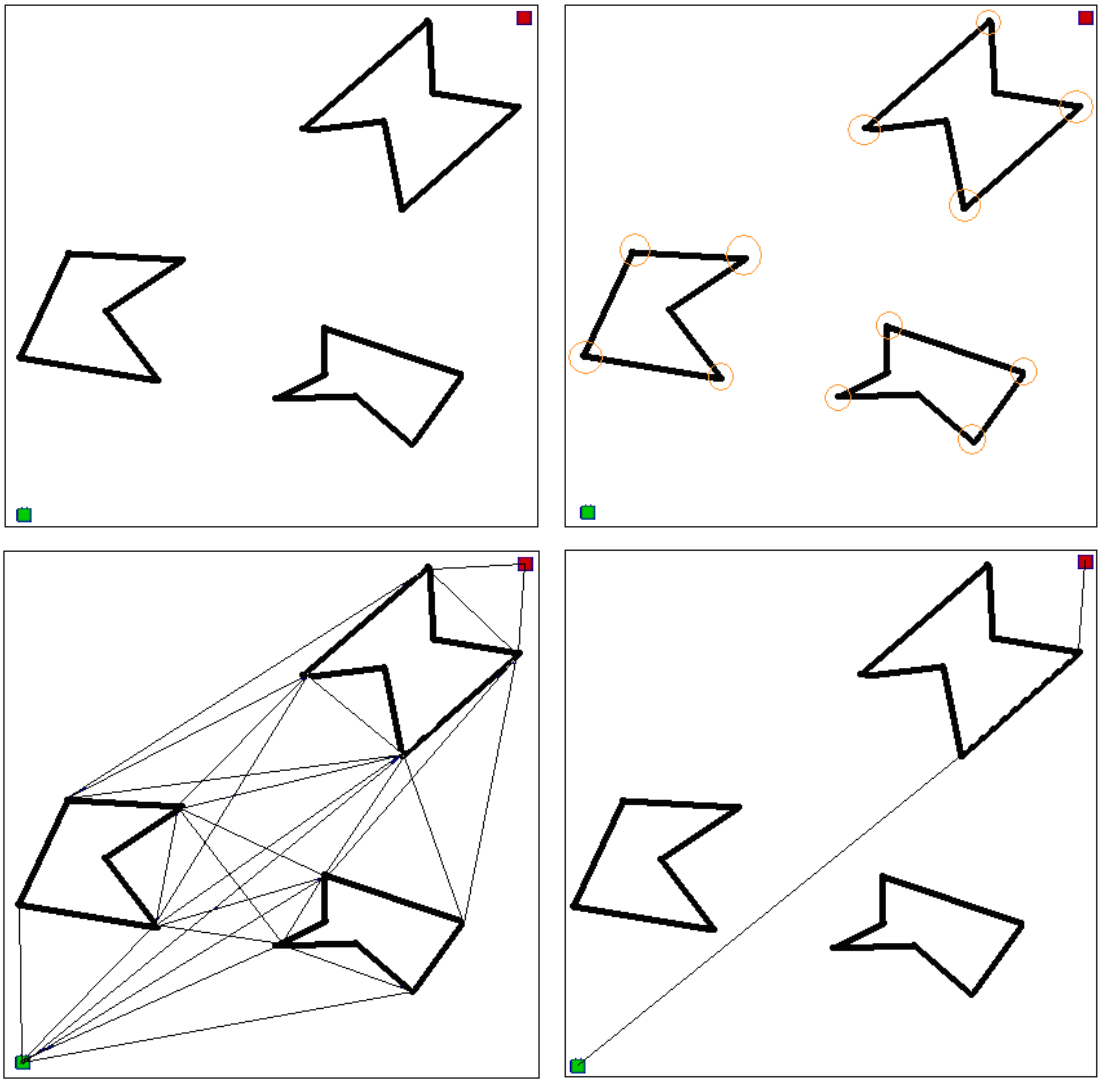


Figure 3-6 (a) Top Left. The environment. (b) Top Right. All convex vertices are circled. (c) Bottom Left. The global tangent graph. (d) Bottom Right. The optimal path.

After identifying the nodes, the optimal direction and distance is determined using the following procedure:

- For each node, evaluate the distance $d(N_i, T)$, where N_i is the i^{th} node.
- The node with the lowest $d(N_i, T)$ is labeled the optimal node, N^*.

The robot should proceed to N^* whilst continuously updating the local tangent graph and proceeding to the most recent N^*. In figure 3-7, N^* is the T-node since the T-node is closest to the target.

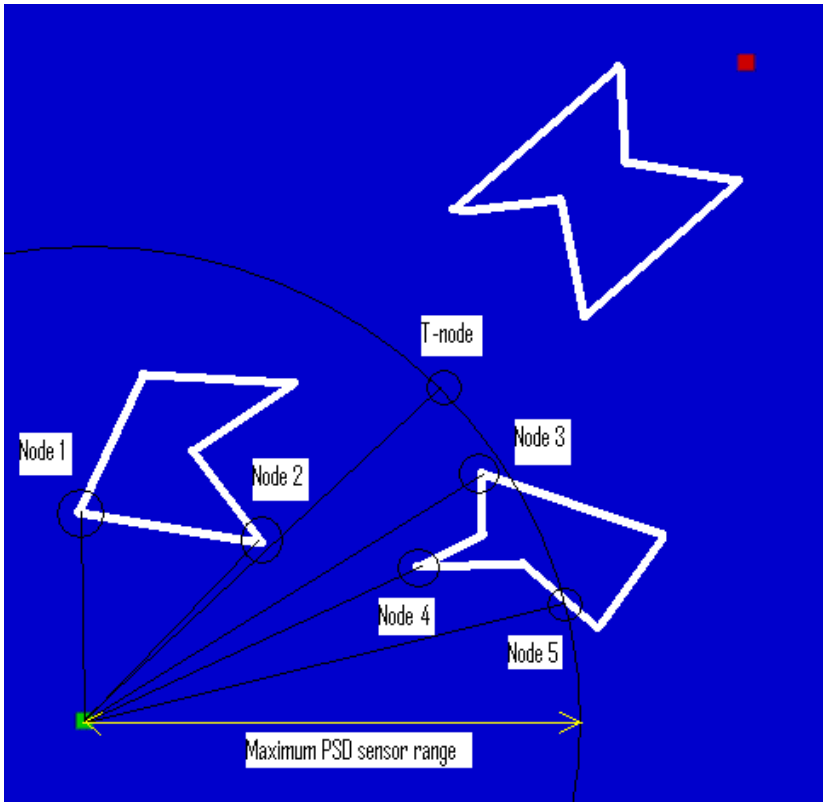


Figure 3-7 The local tangent graph

3.6.3 Local Minimums

Figure 3-8 shows that sometimes the robot must travel away from the target in order to reach it. This is defined as a local minimum. When this happens, tangentbug goes into wall-following mode. This involves choosing a wall following direction and following the wall using the LTG. Whilst following the wall, tangentbug continuously updates two variables:

- $d_{followed}(T)$ - This variable records the minimum distance to the target along the minimum-causing obstacle.
- $d_{reach}(T)$ – Each step, tangentbug scans the visibile environment and for a point P, at which $d(P,T)$ is minimal. $d_{reach}(T)$ is then assigned to $d(P,T)$.

The wall-following mode persists until one of the following occurs:

- $d_{reach}(T) < d_{followed}(T)$.

- The robot has encircled the minimum-causing obstacle. The target is unreachable. Stop.

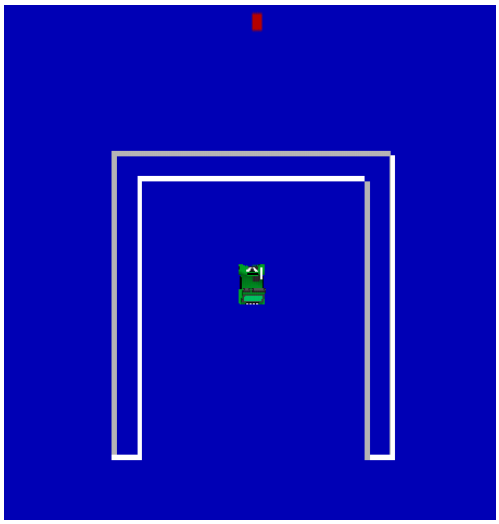


Figure 3-8 The robot in a local minimum

The tangentbug algorithm can be summarised in figure 3-9.

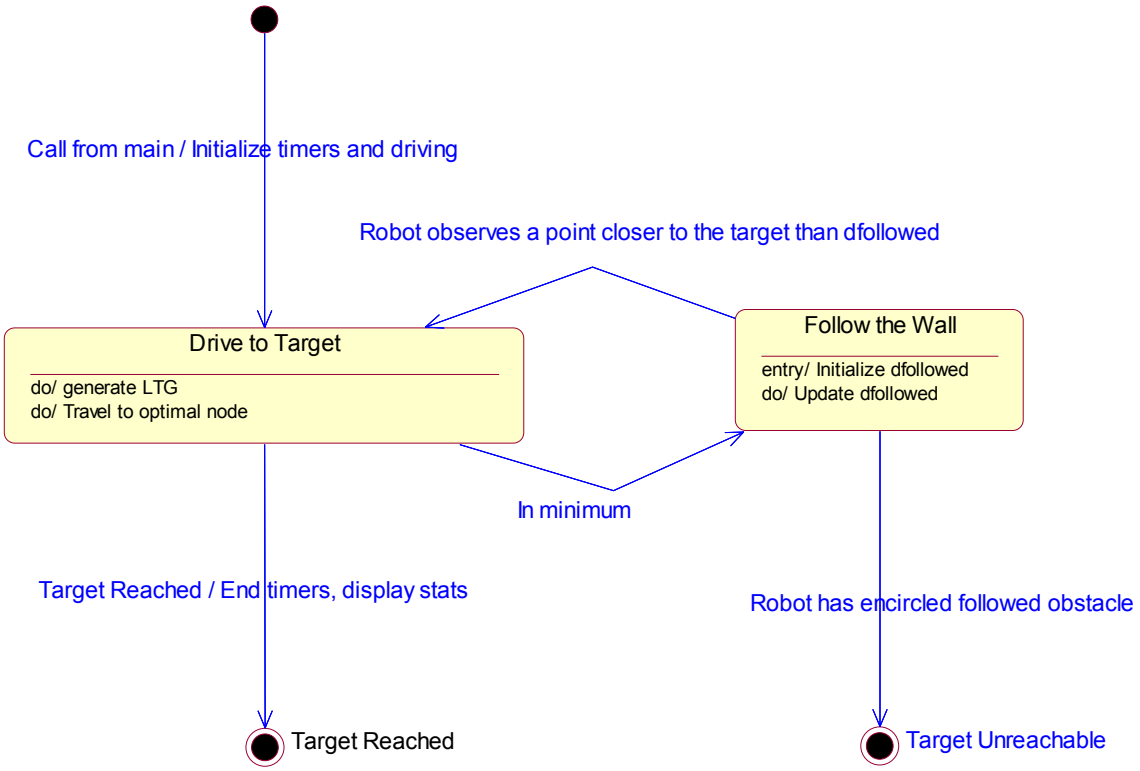


Figure 3-9 Tangentbug's statechart diagram

3.7 The D algorithm*

The D* algorithm was invented by Stentz in 1994 [10]. Since its invention it has been successfully implemented on real world projects with very satisfactory results [8]. Along with tangentbug, D* is acknowledged as producing very short paths given purely local information [8].

The D* algorithm is very different from the bug algorithms. It is a brute force algorithm which has some unique and interesting properties. It segments the map into discrete areas called cells. Each cell has a backpointer, representing the optimal traveling direction in the cell's area, and costs for traveling to neighbouring cells.

The formal low-level algorithm can be found in the source code and those details can be found in Stentz's paper [10]. A more abstract, higher-level example is presented in 3.7.1 and 3.7.2

3.7.1 Generating an optimal path

D* is best explained by example. Let the goal be cell (5,3) and the robot's initial position at (1,3) as depicted in figure 3-10(a). Let the traveling cost be 1 when traveling horizontally or vertically and $\sqrt{2}$ when traveling diagonally.

Then, D* generates table 3-1 for cells surrounding G:

Position (1)	Nearest cell with backpointer or Goal (2)	Cost from (1) to (2)	Cost from (2) to G	Total cost
(5,4)	G	1	0	1
(5,2)	G	1	0	1
(4,3)	G	1	0	1
(4,2)	G	1.414	0	1.414
(4,4)	G	1.414	0	1.414

Table 3-1. The first table generated in the D* algorithm.

Table 3-1 shows that cells (5,4), (5,2) and (4,3) have the lowest total cost. Those cells set their backpointers towards the goal as depicted in figure 3-10(b). Then, the neighbours of G, (5,4), (5,2) and (4,3) are considered for the total minimum cost to goal in table 3-2:

Position (1)	Nearest cell with backpointer or Goal (2)	Cost from (1) to (2)	Cost from (2) to G	Total Cost
(4,4)	G	1.414	0	1.414
(4,2)	G	1.414	0	1.414
(3,3)	(4,3)	1	1	2
(5,1)	(5,2)	1	1	2
(5,5)	(5,4)	1	1	2
(3,2)	(4,3)	1.414	1	2.414
(4,5)	(5,4)	1.414	1	2.414
(4,1)	(5,2)	1.414	1	2.414
(3,4)	(4,3)	1.414	1	2.414

Table 3-2. The second table generated by D*

Table 3-2 shows that cells (4,4) and (4,2) have the lowest total cost. Those cells set their backpointers towards the goal position and the grid is depicted in figure 3-10(c).

This process keeps repeats itself until the robot's position contains a backpointer or the whole grid is filled. If a cell contains a backpointer, it represents the least cost traveling direction to goal. Figure 3-10(d) shows the 5x5 grid with G and backpointers leading to G. As can be verified, following any given backpointer trail will produce a path of least cost. This process is how D* generates optimal paths.

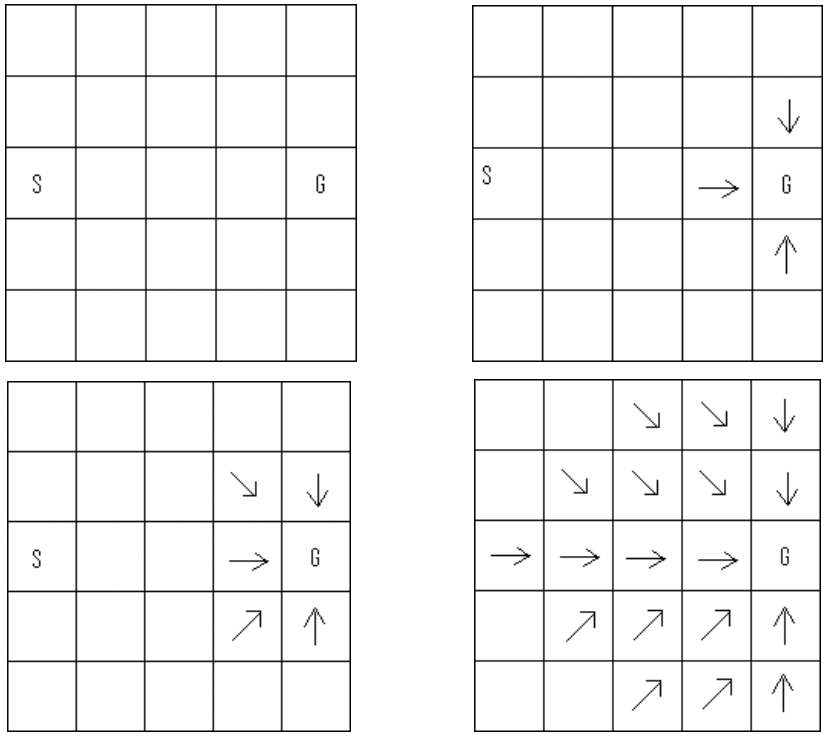


Figure 3-10. (a) Top left. The initial grid. (b) Top right. The grid after data from table 3-1 is entered. (c) Bottom left. The grid after data from table 3-2 is entered. (d) Bottom right. The final grid.

3.7.2 Accounting for Obstacles

D* represents obstacles by largely increasing cost to travel to, but not from, obstacle cells. That is, if an obstacle exists on a cell O, the travel cost from O's neighbour cells to O becomes some large predefined value. Figure 3-11(a) shows that an obstacle at (3,3) has been detected. The arcs shown lead to the obstacle cell and their associated cost becomes very large.

Once travel costs are modified, D* recomputes the cell backpointers to ensure they are still optimal. D* does this by firstly considering cells which have a backpointer to cell (3,3). It generates table 3-3:

Position (1)	Nearest cell with backpointer or Goal (2)	Cost from (1) to (2)	Cost from (2) to G	Total Cost
(2,2)	(3,2)	1	2.414	3.414
(2,4)	(3,4)	1	2.414	3.414
(2,3)	(3,4)	1.414	2.414	3.828

Table 3-3. The first table drawn after an obstacle was detected at (3,3).

Table 3-3 shows that cells (2,2) and (2,4) have a new minimum cost and change their backpointers to the cell specified in column 2. The updated grid is shown in figure 3-11(b). D* repeats this process again and generates table 3-4.

Position (1)	Nearest cell with backpointer or Goal (2)	Cost from (1) to (2)	Cost from (2) to G	Total Cost
(2,3)	(3,4)	1.414	2.414	3.828
(2,1)	(3,2)	1.414	2.414	3.828
(2,5)	(3,4)	1.414	2.414	3.828
(1,4)	(2,4)	1	3.414	4.414
(1,2)	(2,2)	1	3.414	4.414
(1,3) (S)	(2,2)	1.414	3.414	4.828
(1,5)	(2,4)	1.414	3.414	4.828
(1,1)	(2,2)	1.414	3.414	4.828

Table 3-4. The second table drawn after an obstacle was detected at (3,3)

Table 3-4 shows that cells (2,3), (2,1) and (2,5) change their backpointers so that their costs to goal are minimised. Hence, the updated grid is shown in figure 3-11(c).

D* repeats this process until the minimum total cost in the generated table is greater or equal to the robot's cost to goal following its current backpointer trail. Once this occurs, it signals that further computation will not yield less costly paths than the current path. Following the example, table 3-5 is computed:

Position (1)	Nearest cell with backpointer or Goal (2)	Cost from (1) to (2)	Cost from (2) to G	Total Cost
(1,2)	(2,2)	1	3.414	4.414
(1,4)	(2,4)	1	3.414	4.414
(1,3) (S)	(2,3)	1	3.828	4.828
(1,5)	(2,4)	1.414	3.414	4.828
(1,1)	(2,2)	1.414	3.414	4.828

Table 3-5. The third table drawn after an obstacle was detected at (3,3)

The terminating condition holds in table 3-6, and figure 3-11(d) shows the final grid.

Position (1)	Nearest cell with backpointer or Goal (2)	Cost from (1) to (2)	Cost from (2) to G	Total Cost
(1,3) (S)	(2,3)	1	3.828	4.828
(1,5)	(2,4)	1.414	3.414	4.828
(1,1)	(2,2)	1.414	3.414	4.828

Table 3-6. The forth table drawn after an obstacle was detected at (3,3)

Note that cell (2,3) does not point backwards towards the start, unlike the force-field heuristic technique! D* maintains optimality and avoids getting stuck in local minimums which have troubled similar techniques [9]. However, as will be shown later, this comes at the cost of computation time.

In D*, cost modification can be done at any time. This allows the algorithm to dynamically adapt to unseen obstacles and generate new optimal paths. D*'s costing mechanism also allows for terrain which is undesirable, but not necessarily an obstacle. This is far better than the bug algorithms where the terrain is either traversable or an obstacle.

3.7.3 Determining reachability

Unreachability is determined by comparing the backpointer trail's cost to the large threshold value of obstacles. If the backpointer trail's cost is greater than the threshold value, it implies that the optimal path crosses an obstacle and therefore the target is

unreachable. Of course, the large threshold value should be chosen such that the cost of any sequence of backpointers which do not cross an obstacle will never exceed the large threshold value.

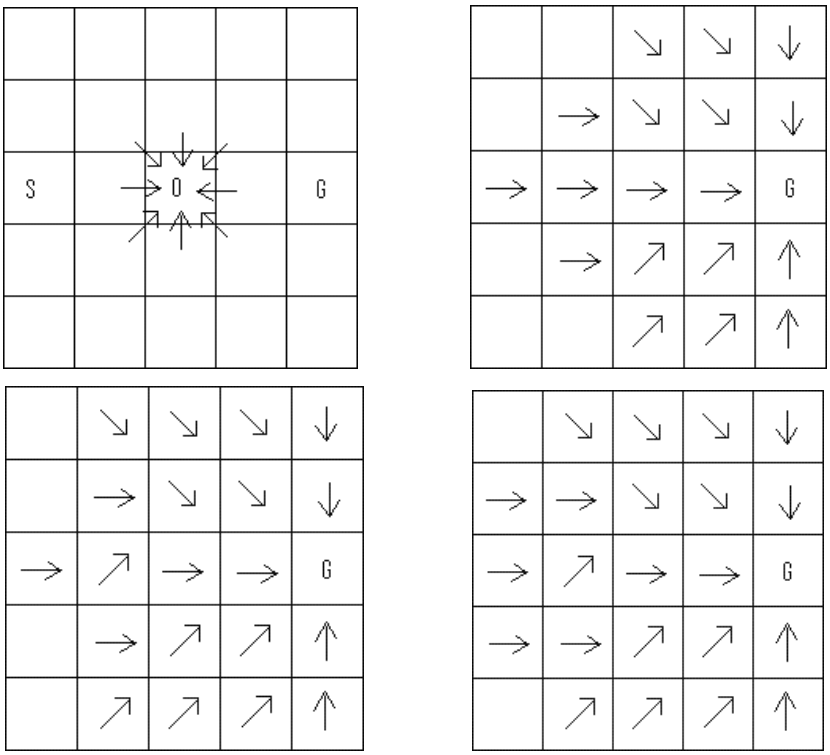


Figure 3-11. (a) Top left. An obstacle cell is identified in position (3,3). (b) Top right. The grid after data from table 3-3 is entered. (c) Bottom left. The grid after data from table 3-4 is entered. (d) Bottom right. The grid after data from table 3-6 is entered. Once again, backpointers now represent optimal traveling directions.

4. Algorithm Implementation

4.1 Common Modules

Typically, an algorithm is implemented in the navigation class and calls the common modules. Common modules are used for consistency between simulations and modularity. For instance, all navigation algorithms require completion time to be measured and the timer module provides methods specifically for that purpose. Figure 4-1 shows the common modules and the navigation module which can be altered for implementing a specific algorithm.

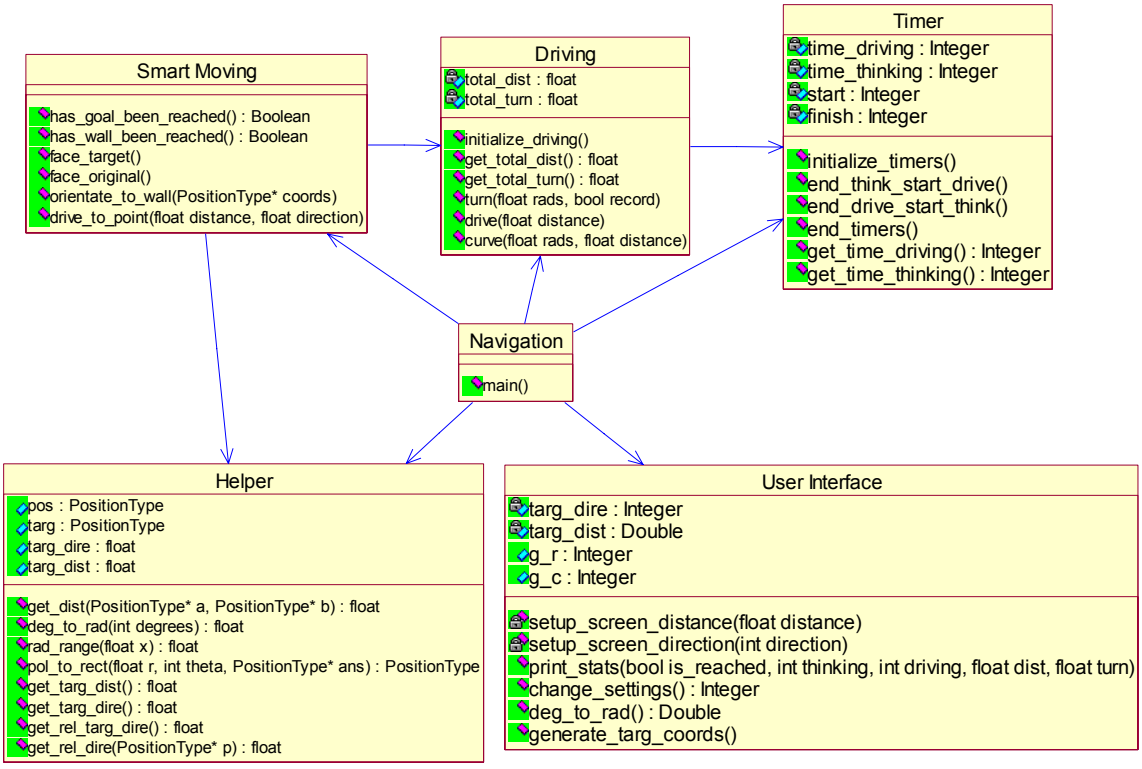


Figure 4-1 The class diagram of the common modules.

4.1.1 The timer module

This module's function is to measure the algorithm's time performance. In particular, it measures computation and driving time and returns these times upon request. Although driving time is not measured in this study, it may be needed in future.

The timer module fulfills its role by providing an abstract interface to the c function clock(). The clock() function returns the time (in milliseconds) spent in the processor of the calling process since execution began. Note that the clock() does not include the time which the thread is sleeping.

4.1.2 The helper module

The helper module provides low-level support to other modules. In particular, the robot can get the target's distance and direction by calling methods found in the helper module. Currently, the helper module relies on dead-reckoning to generate answers. In a future version, if landmark recognition or sensor networks are used, these functions can be changed and the rest of the system need not know.

4.1.3 The user interface module

The user interface module's role is to interface between the program and the user. When the program starts, it allows the user to edit the desired direction and distance of the goal. Figure 4-2(a) shows the screen which allows the user to edit the distance to goal and figure 4-2(b) shows the screen which allows the user to edit the direction to goal.

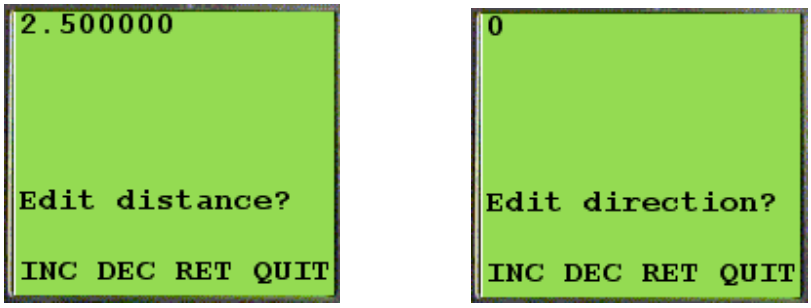


Figure 4-2 (a) Left. The user can edit the distance to goal. (b) Right. The user can edit the direction to goal.

The user interface module also displays the navigation results to the user. Figure 4-3 shows the screens which appear when convergence is achieved. Figure 4-3(a) shows computation and driving time, in milliseconds. Figure 4-3(b) shows distance traveled in metres and the rotation in radians. Figure 4-3(c) shows the number of calls to the math library or process-state() if D* is run.

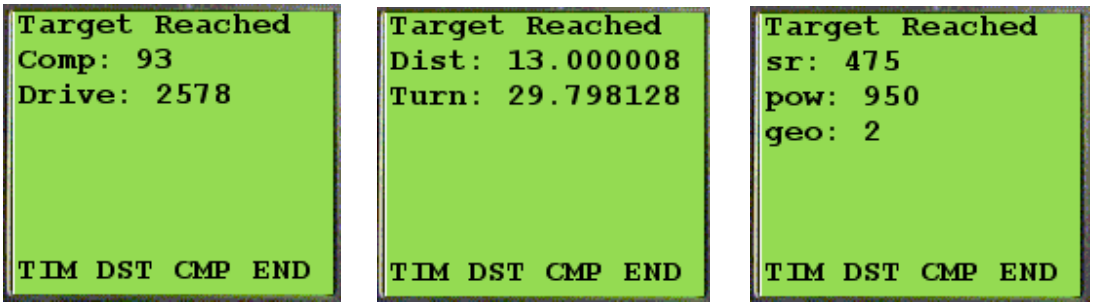


Figure 4-3 (a) Left. Computation and driving times. (b) Centre. Total distance traveled and total rotation performed. (c) Right. Calls to the maths library or process-state() in D*.

4.1.4 The driving module

The driving module's purpose is to record the total distance traveled and the total rotation performed. Essentially, it provides a simpler interface to the VW driving interface and extends functionality by tracking total distance and rotation.

It allows the caller to specify whether it wishes to record a turning request in total_turn. As will be seen later, some turning is not inherently generated by the algorithm. Instead, it is hardware dependant and it may be interesting to remove this component from rotation results.

The driving module calls the timer module so that driving time is properly separated from computation time. Figure 4-4 shows the drive function in the driving module. It calls end_think_start_drive() to denote that driving has started and then end_drive_start_think() to denote that driving has ended. Note that during driving, VWDriveWait() is not called and a busy loop has replaced it. This is because VWDriveWait() puts the navigation process to sleep and this distorts driving time results.

```
void drive(float distance){
      total_dist = total_dist + distance;
      end_think_start_drive();
      VWDriveStraight(vw, distance, LINEAR_VELOCITY);
      while(VWDriveDone(vw) == 0){
            KEYRead();
      }
      end_drive_start_think();
      VWGetPosition(vw, &pos);
}
```

Figure 4-4 The drive method

4.1.5 The smart moving module

The smart driving module's role is to provide abstract driving functions as required by the main navigation algorithm. Most of its public methods are self-explanatory, but its most complicated method, follow_the_wall(), is depicted in figure 4-5.

```
void follow_the_wall(bool is_on_right){
      if(is_on_right){
            if(has_wall_been_reached()){
                  turn_not_move(FALSE);
            }
            else if(PSDGet(psd_right) > WALL_DISTANCE+THRESHOLD){
                  turn_and_move(TRUE);
            }
            else{
                  follow_wall_straight(TRUE);
            }
      }
      else{
            if(has_wall_been_reached()){
                  turn_not_move(TRUE);
            }
            else if(PSDGet(psd_left) > WALL_DISTANCE+THRESHOLD){
                  turn_and_move(FALSE);
            }
            else{
                  follow_wall_straight(FALSE);
            }
      }
}
```

Figure 4-5. The follow_the_wall method

Initially, the method checks if a wall is in front of the robot. If so, the robot calls turn_not_move() and the robot turns on the spot as shown in figure 4-6(a). Otherwise, the robot checks if a wall is to the right of the robot. If so, the robot calls follow_wall_straight() and the robot follows the wall as shown in figure 4-6(c). If not, the

robot calls turn_and_move the robot turns and moves as shown in figure 4-6(b). After calling the above methods robot aligns to the wall by calling the orientate_to_wall() method as depicted in figure 4-6(d).

Presently, this uses a proportional-derivative controller to ensure that the wall is followed closely. However, more advanced PID or fuzzy logic controllers can easily be implemented by updating only the smart moving module.

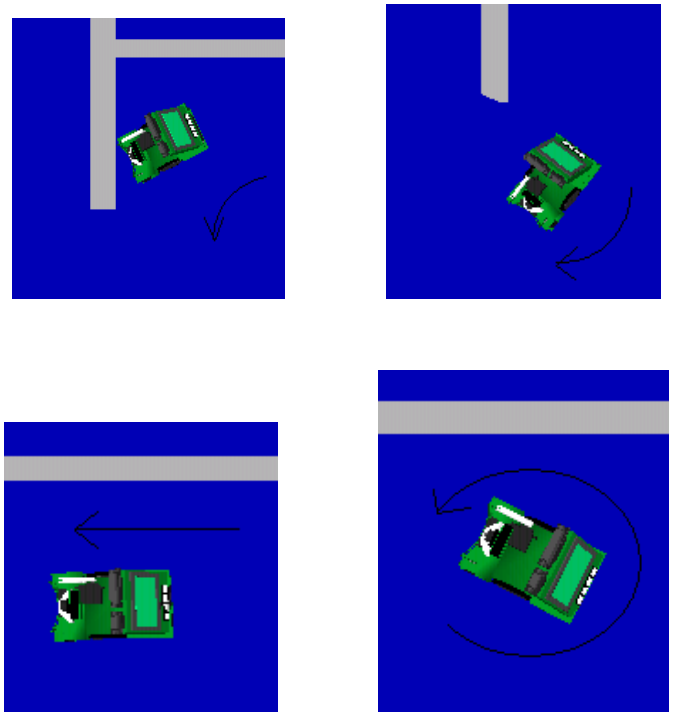


Figure 4-6. (a) Top left. Turn_not_move(). (b) Top right. Turn_and_move() (c) Bottom left. Follow_wall_straight() (d) Bottom right. Orientate_to_wall()

4.2 Bug1 Implementation

The bug1 algorithm is implemented by calling methods from the common modules as shown in figure 4-7. It shows that Bug1 implements the "drive to target" and "follow the wall" states prescribed in figure 3-1 using the methods drive_to_target() and follow_wall_Bug1() respectively. Another method, Bug1(), is used to coordinate the state transitions prescribed in figure 3-1.

```
/*function orientates the eyebot to the target and drives towards it until either
the target is reached or a wall is hit.*/
int drive_to_target(){
	face_target();
	while(TRUE){
		if(has_goal_been_reached()){
			face_original();
			return TARGET_REACHED;
		}
		else if(has_wall_been_reached()){
			return WALL_HIT;
		}
		drive(STEP);
	}
}

/*function follows the wall according to the Bug1 algorithm*/
int follow_wall_Bug1(){

	PositionType leave;  //closest position to the target
	PositionType hit;  //the current hit point
	float min_dist;  //the closest displacement to the target
	float dist_to_min=0;   //the number of steps to leave
	float begin_dist = get_total_dist();

	initialize_PD();
	VWGetPosition(vw, &leave);
	min_dist = get_targ_dist();
	VWGetPosition(vw, &hit);

	orientate_to_wall(FALSE);
	while(TRUE){
		if(get_dist(&hit, &pos)<=TARG_ERROR &&
		(get_total_dist()-begin_dist)>TARG_ERROR){
			break;
		}
		follow_the_wall(TRUE);
		if(get_targ_dist()<min_dist){
			min_dist = get_targ_dist();
			VWGetPosition(vw, &leave);
			dist_to_min = get_total_dist()-begin_dist;
		}
	}

	/*Check the unreachability condition - if true then terminate*/
	if(get_dist(&leave, &hit)<=TARG_ERROR){
		face_original();
		return TARGET_UNREACHABLE;
	}
```

```
        /*Determine the shortest route to the min point and turn left
        or right accordingly.  Then follow the wall to the min point*/
        if(dist_to_min < (get_total_dist()-begin_dist)/2){
               while(get_dist(&leave, &pos)>=TARG_ERROR){
                      follow_the_wall(TRUE);
               }
        }
        else{
               turn(M_PI);
               while(get_dist(&leave, &pos)>=TARG_ERROR){
                      follow_the_wall(FALSE);
               }
        }
        return MIN_REACHED;
}

/*function drives towards the target, using Bug1 algorithm*/
void Bug1(){

     int state = STEP1;
     int response;

     initialize_driving();
     initialize_timers();
     init_helper();

     while(TRUE){
         if(state==STEP1){
              response = drive_to_target();
              if(response==TARGET_REACHED){
                     end_timers();
                     print_stats(TRUE, get_time_thinking(), get_time_driving(),
                     get_total_dist(), get_total_turn(), num_sqrt, num_pow,
                     num_geom);
                     break;
              }
              else if(response==WALL_HIT){
                     LCDPrintf("Wall Hit\n");
                     state = STEP2;
                     continue;
              }
         }
         else if(state==STEP2){
              response = follow_wall_Bug1();
              if(response==MIN_REACHED){
                     LCDPrintf("Minimum Point\n");
                     state=STEP1;
                     continue;
              }
              else if(response==TARGET_UNREACHABLE){
                     end_timers();
                     print_stats(FALSE, get_time_thinking(), get_time_driving(),
                     get_total_dist(), get_total_turn(), num_sqrt, num_pow,
                     num_geom);
                     break;
               }
          }
     }
}
```

Figure 4-7. Bug1 Navigation Module

4.3 Bug2 Implementation

The bug2 navigation class calls the common modules in a similar fashion to bug1 described in section 4.2.1. However, Bug2 requires an extension to the smart moving module to include a method which determines if it is on the M line. A new method has been created in the smart moving module called is_on_M_line for this purpose. The updated class diagram is displayed in figure 4-8.

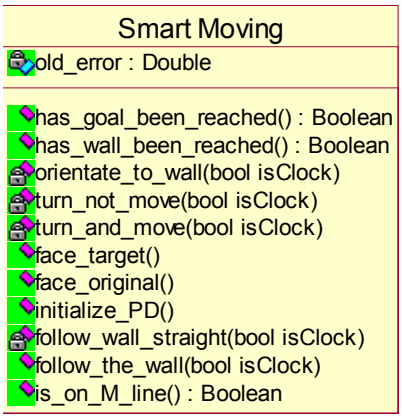


Figure 4-8 The extended Smart Moving module for bug2

To implement the is_on_M_line() function, consider the diagram in figure 4-9. Let T be a vector to the target, P a vector to the robot's current position and a some scalar such that the vectors P-aT and T are perpendicular.

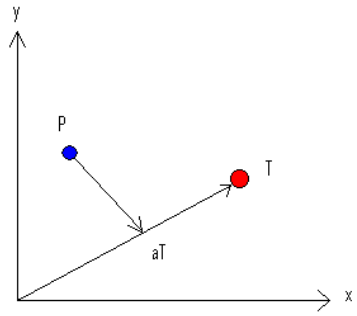


Figure 4-9 The vectors P-aT and T are perpendicular

It follows from the dot product that:

$$(T_x)(P_x - aT_x) + (T_y)(P_y - aT_y) = 0$$

Rearranging for a gives:

$$a = \frac{T_x P_x + T_y P_y}{T_x^{\,2} + T_y^{\,2}}$$

Hence, *aT* can be calculated and its distance to P can be determined. If this distance is less than a threshold value, the robot is on the M line. Figure 4-10 shows the implementation of is_on_M_line()

```
/*function determines whether the given point is on the M line*/
bool is_on_M_line(){

	PositionType closest;
	float t = (targ.y*pos.y + targ.x*pos.x)/(pow(targ.x,2.0)+pow(targ.y,2.0));
	num_pow = num_pow+2;
	if(t<0 || t>1){
		return false;
	}
	else{
		closest.x = t*targ.x;
		closest.y = t*targ.y;
		closest.phi = 0;
		return (get_dist(&closest, &pos) <= TARG_ERROR);
	}
}
```

Figure 4-10 The is_on_M_line() method

4.4 Alg1 Implementation

Alg1 requires two extensions to the smart moving module. It needs to know if the robot is on the M line and the freespace, *F*. The is_on_M_line() method, described in section 4.3, is reused. However, a new method, freespace(), needs to be created to determine *F*. Figure 4-11 shows the updated Smart Moving module which includes the two new methods.

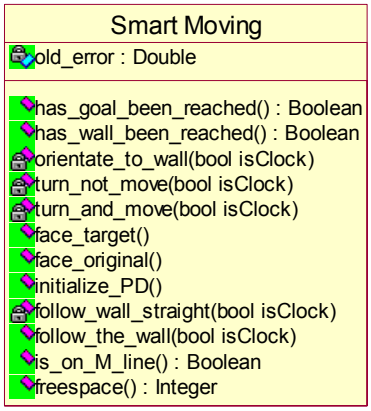


Figure 4-11 The extended Smart Moving algorithm for Alg1

The freespace method returns *F*. When the method is invoked, the target's direction relative to the robot is firstly determined. Then, the PSD rotationally closest to that direction is identified. Subsequently, the robot rotates such that the PSD is facing the target and measures *F* using that PSD. After that, the robot returns to its original orientation. This method is implemented in figure 4-12.

```
/*function returns the freespace in the direction of the target
function assumes that the PSDs are evenly spaced*/
int freespace(){
      float direction = get_rel_targ_dire();
      int index=0;
      int answer;

      /*determine the PSD closest to the relative direction*/
      while(direction < -M_PI/NUM_PSD){
            direction = direction + 2*M_PI/NUM_PSD;
            index--;
      }
      while(direction > M_PI/NUM_PSD){
            direction = direction - 2*M_PI/NUM_PSD;
            index++;
      }
      index = (index + NUM_PSD/2)%NUM_PSD;

      /*turn towards the target and get the freespace*/
      turn(direction, FALSE);
      answer = PSDGet(psd[index]);
      turn(-direction, FALSE);

      return answer;
}
```

Figure 4-12 The freespace method

In this particular robot, there are 8 PSD sensors. Figure 4-13 shows that each PSD covers a 45 degree sector. Hence, the maximum the robot needs to rotate to find *F* is 22.5 degrees. As expected, increasing the number of PSDs lowers the maximum rotation to find *F* and this must be factored into cost against performance decisions.

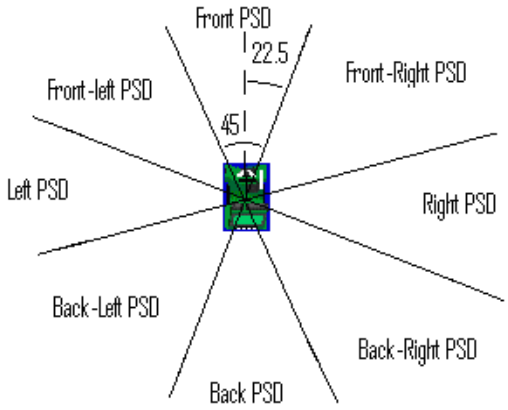


Figure 4-13 Each PSD covers a 45 degree sector. Maximum rotation is 22.5 degrees.

The alg1 algorithm also needs to record all hit and leave points encountered. It does this by implementing a data-structure module which is described in figure 4-14.

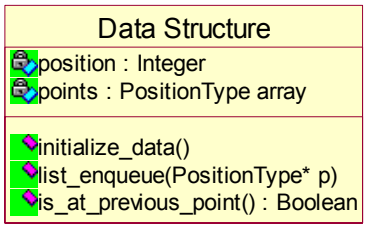


Figure 4-14 The Data Structure Module

The data structure is implemented as an array of PositionTypes. The number of elements is predetermined and a fixed block of memory is allocated when the program is started. Figure 4-15(a) shows the data structure immediately after initialize_data() is called. When list_enqueue() is called, a PositionType is stored in the element referenced by position. Figure 4-15(b) shows the data structure after one such call. When is_at_previous_point() is called, the data structure checks if the robot's current position is near any stored points.

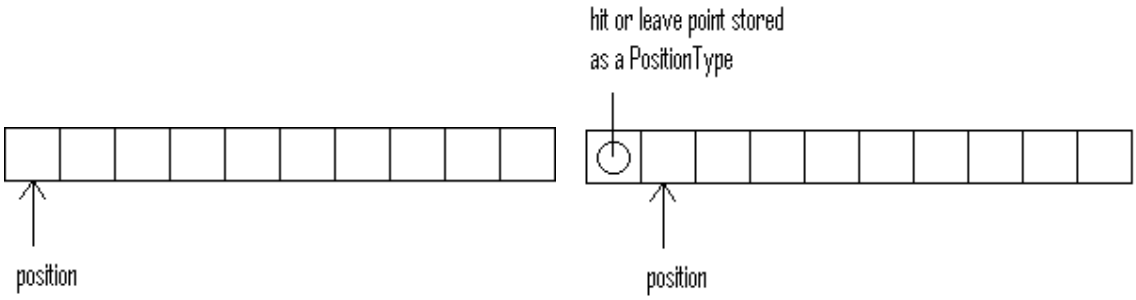


Figure 4-15 (a) Left. The data structure after initialization. (b) Right. The data structure after list_enqueue() is called.

4.5 Alg2 Implementation

The Alg2 algorithm is implemented by using the common modules and the extensions implemented previously. In particular, Alg2 reuses the freespace() and data structure modules discussed in section 4.2.3. It also uses the common modules to implement navigation states similar to the Bug1 implementation in section 4.2.1

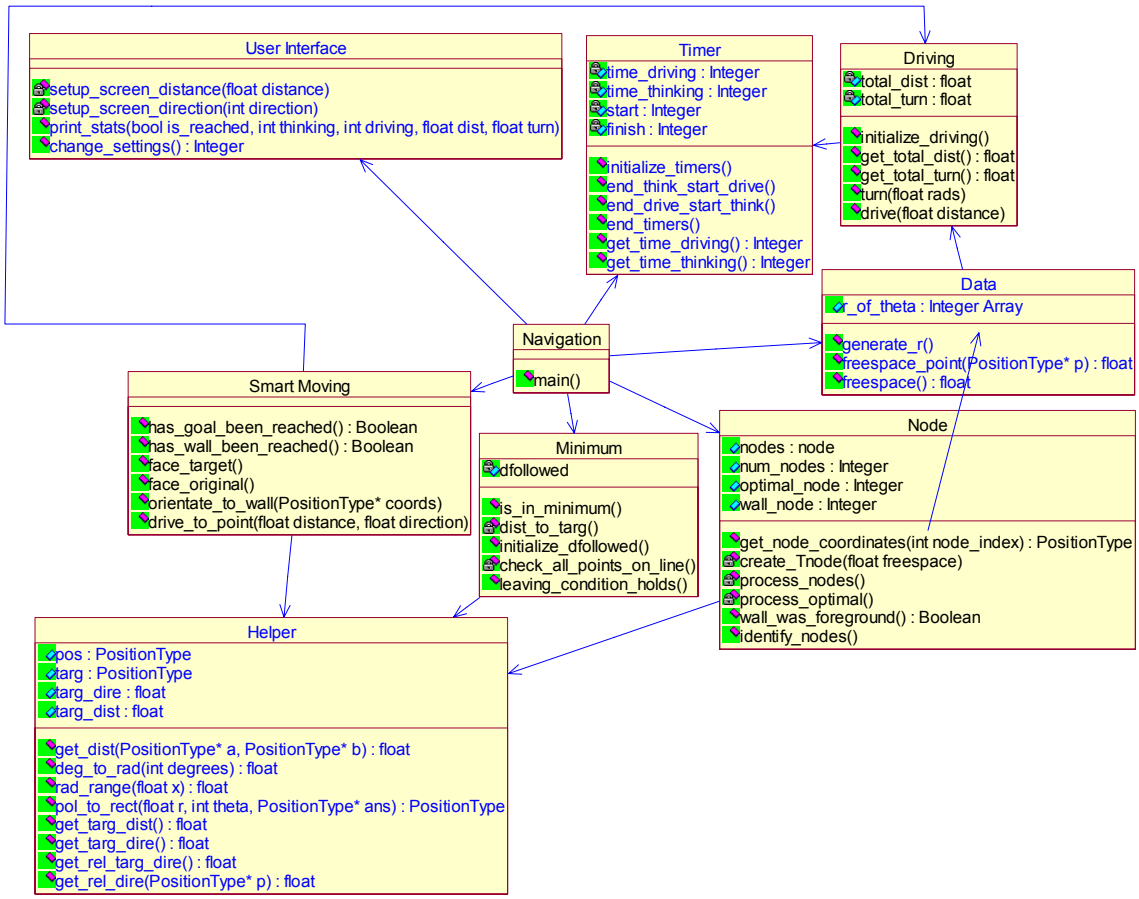
4.6 Distbug Implementation

The distbug algorithm is simpler than the Alg2 algorithm in that it does not require the data structure module. Apart from that, it is very similar to Alg2 and therefore its implementation is also very similar.

4.7 Tangentbug Implementation

The tangentbug algorithm has been modified from the original article. In the original tangentbug generates the LTG continuously when moving. In this implementation, tangentbug only generates the LTG when it has reached node positions. This change is necessary to avoid excessive rotation and does not decrease performance on the three maps.

The tangentbug algorithm is significantly more complicated than any of the previous bug algorithms. It has modified the common modules extensively. A redrawn class diagram is shown in figure 4-16.



4-16 The tangentbug class diagram

4.7.1 The data module

The tangentbug algorithm requires extensive collection of data for $r(\theta)$ and freespace toward a particular point. The data module collects PSD data and stores it for use by the rest of the system.

To achieve this in an optimal and efficient manner, the data module equally divides the scanning task between the eight PSDs. Therefore, each PSD is responsible for collecting data about a 45 degree sector. Each colour in figure 4-17 shows the division of sectors.

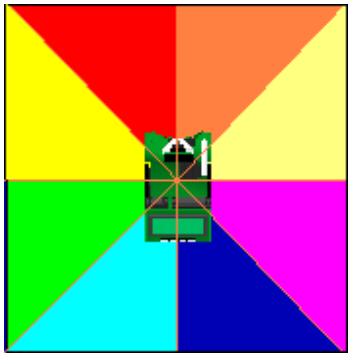


Figure 4-17 Each PSD collects data in its sector

Then, this 45 degree sector is sampled according to a user-defined value DEG_BET_SAMPLES. The default is 3 degrees, but this can be altered for greater accuracy. Hence, each PSD will sample its sector 15 times, turning 3 degrees between each sample. Figure 4-18 shows the source code which implements data gathering and figures 4-19(a) and 4-19(b) shows the robot actually gathering data.

```
void generate_r(){
	int reading_index;
	int psd_index;
	int readings_per_psd = 360/(NUM_PSD*DEG_BET_SAMPLES);

	for(reading_index=0; reading_index<readings_per_psd; reading_index++){
		for(psd_index=0; psd_index<NUM_PSD; psd_index++){
			r_of_theta[readings_per_psd*psd_index+reading_index] =
			PSDGet(psd[psd_index]);
		}
		turn(deg_to_rad(DEG_BET_SAMPLES), FALSE);
	}
	turn(-deg_to_rad(360/NUM_PSD), FALSE);
}
```

Figure 4-18 The generate_r method

The sampled data is stored publicly in an array. The number of elements in the array depends on DEG_BET_SAMPLES, which is assigned a default value of 3. If this default is used, there are 120 elements in the array. The 0^{th} element contains the distance straight ahead of the robot and the i^{th} element contains the distance on a DEG_BET_SAMPLES*i angle measured counterclockwise from straight ahead.

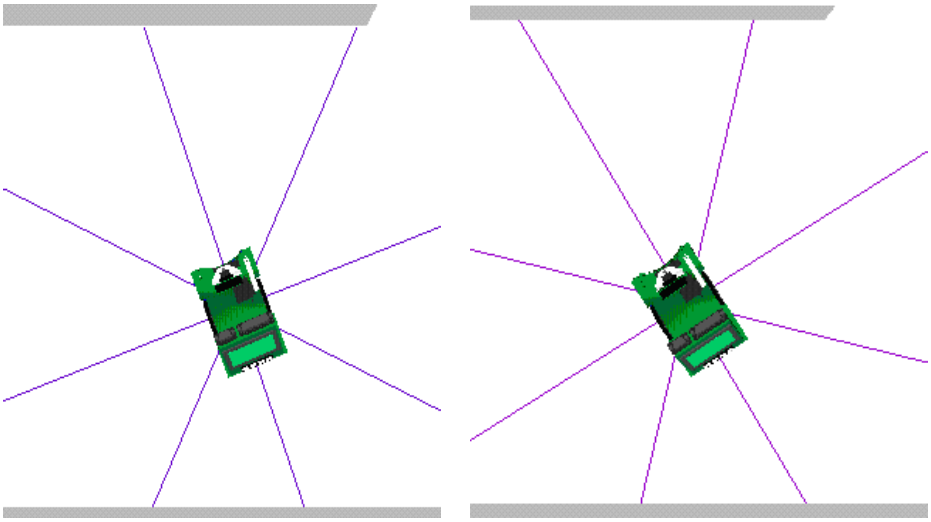


Figure 4-19 (a) Left. The robot gathers data from all 8 PSD sensors (b) Right. After a 3 degree rotation, the robot gathers data from 8 PSD sensors again.

4.7.2 The node module

After the data has been collected, it is processed for nodes. The node module identifies and processes nodes which are subsequently stored in a public array. In addition, the optimal node, N^*, and the wall node are identified. Due to $r(\theta)$'s discrete nature, discontinuity detection must be conducted by comparing values of $r(\theta)$. A node is identified if:

- the difference between two successive values of $r(\theta)$ is greater than a predefined threshold, or
- One, and only one, of two successive values of $r(\theta)$ is equal to r, or
- $F > d(x, T)$, which means the target is visible, or
- $F = r$, which means there are no visible obstacles in the target's path.

Once all nodes are identified, each node is processed by calculating $d(N_i,T)$. Then, the optimal node is identified by finding the node with the lowest value of $d(N_i,T)$. Subsequently, the wall node is identified by finding the node with the lowest θ in $r(\theta)$. This is because $r(\theta)$ records measurements anti-clockwise where $\theta = 0$ is straight ahead. Given that nodes are processed by increasing θ, the wall node is always the first identified node. This process is summarised in the flow diagram in figure 4-20.

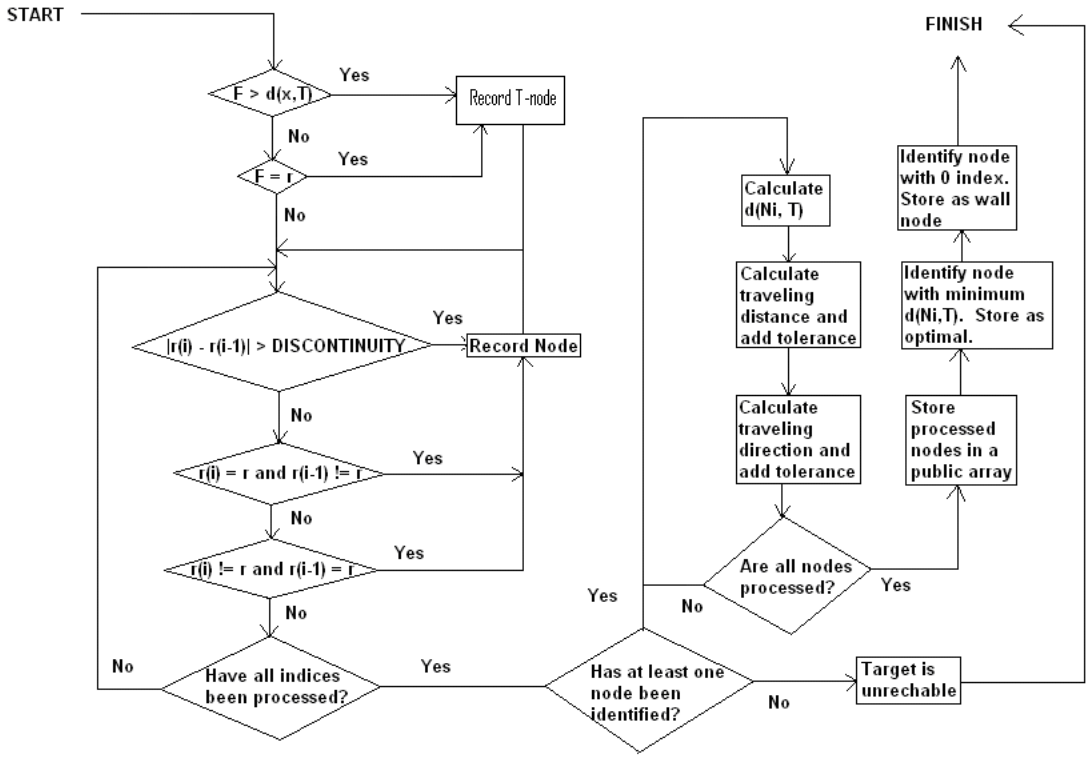


Figure 4-20 The node processing algorithm

4.7.3 The minimum module

If the tangentbug algorithm detects that the robot is in a local minimum, it calls the minimum module. This minimum module's role is to return whether or not the robot has met the leaving condition, $d_{reach}(T) < d_{followed}(T)$. This evaluation must be done using the least amount of computing resources possible.

With this in mind, a strategy was created to evaluate $d_{reach}(T)$ and $d_{followed}(T)$ and its source code is shown in figure 4-21. Firstly, the minimum module queries the node module to find the wall node's index in $r(\theta)$. Denote this index w. The indices from 0 to w represent the minimum causing obstacle and is used to evaluate $d_{followed}(T)$. The remaining indices represent the sector which must be scanned to evaluate $d_{reach}(T)$.

To evaluate $d_{followed}(T)$, for each index from 0 to w determine the distance to target at end-points. In figure 4-22, these indices are indicated by the red lines and the points used for distance to target calculations are indicated by the black squares. The shortest of these distances, since wall-following mode began, is recorded in $d_{followed}(T)$.

To evaluate $d_{reach}(T)$, for the indices w+1 and 360/num_psd, determine the distance to target at regular intervals. For the remaining indices, determine the distance to target only at end-points. In figure 4-22, these indices are indicated by the green lines and the points used for distance to target calculations are indicated by the orange squares. The shortest of these distances, since the robot last refreshed $r(\theta)$, is recorded in $d_{reach}(T)$.

This strategy may appear flawed because the target could be visible but $d_{reach}(T)$ would not record 0. This can easily be remedied by including the freespace() method. If $F > d(x, T)$ this implies the target is visible and the robot would drive straight towards it.

```
bool leaving_condition_holds(){
       int num_samples = 360/DEG_BET_SAMPLES;
       int wall_index = nodes[wall_node].small_index;
       int i;
       float test, dreach;

       /*update (global) dfollowed, if necessary*/
       for(i=0; i<=wall_index; i++){
              test = dist_to_targ(i);
              if(test<dfollowed){
                     dfollowed = test;
              }
       }

       /*evaluate dreach*/
       dreach = check_all_points_on_line(i);
       i++;
       for(; i<num_samples-1; i++){
              test = dist_to_targ(i);
              if(test<dreach){
                     dreach = test;
              }
       }
       test = check_all_points_on_line(i);
       if(test<dreach){
              dreach=test;
       }

       /*evaluate leaving condition*/
       return dreach < dfollowed;
}
```

Figure 4-21 The leaving_condition_holds() method

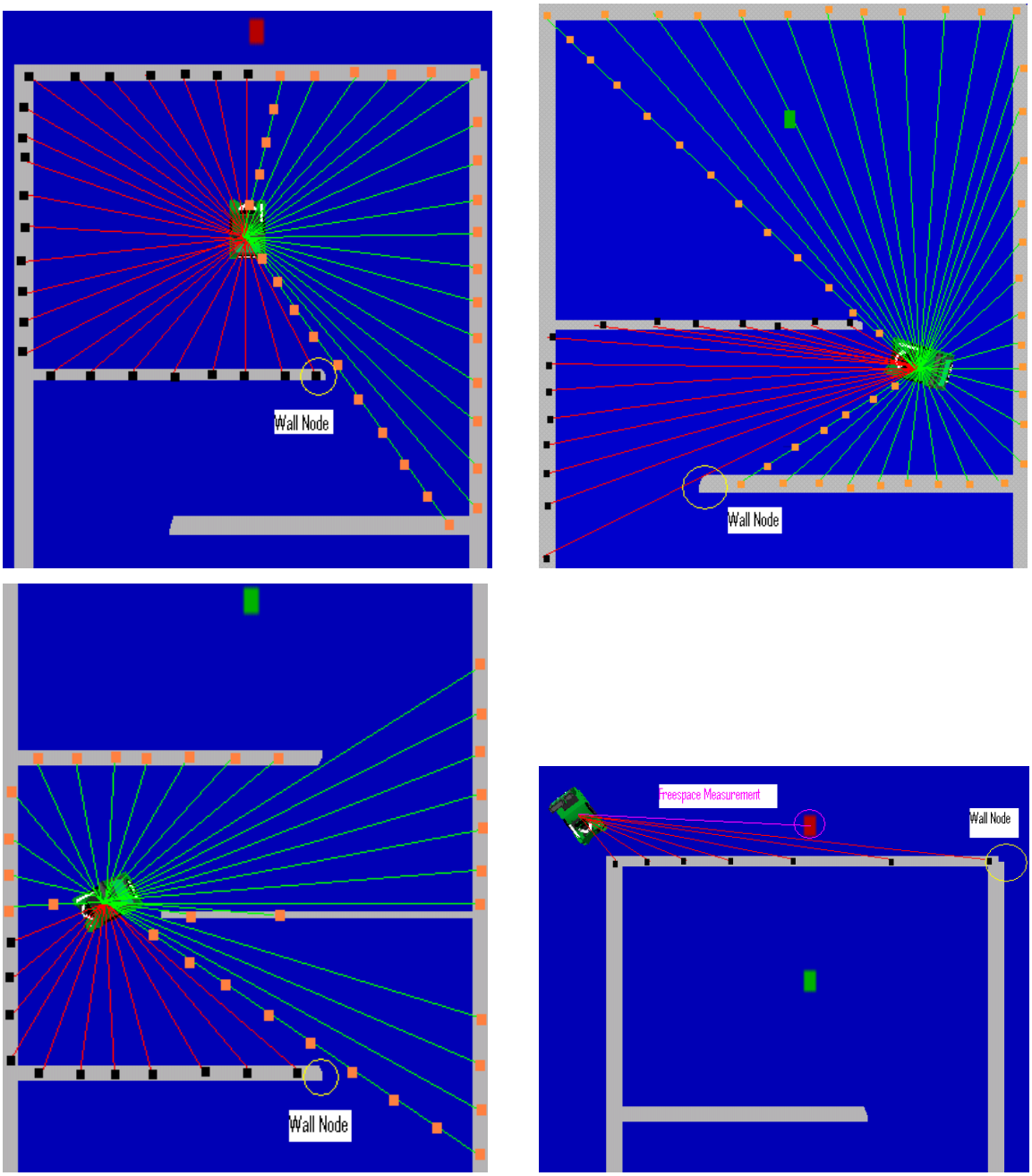


Figure 4-22. (a) Top left. The scanning performed at its initial position. (b) Top right. The scanning performed after traveling one node. (c) Bottom Left. The scanning performed after traveling two nodes. (d) Bottom Right. Freespace identifies a visible target.

4.8 D Algorithm Implementation*

The D* algorithm only reuses the timer module because it is fundamentally different than the other algorithms. The implementation is heavily object-oriented due to the greatly increased complexity. Figure 4-23 shows the D* class diagram.

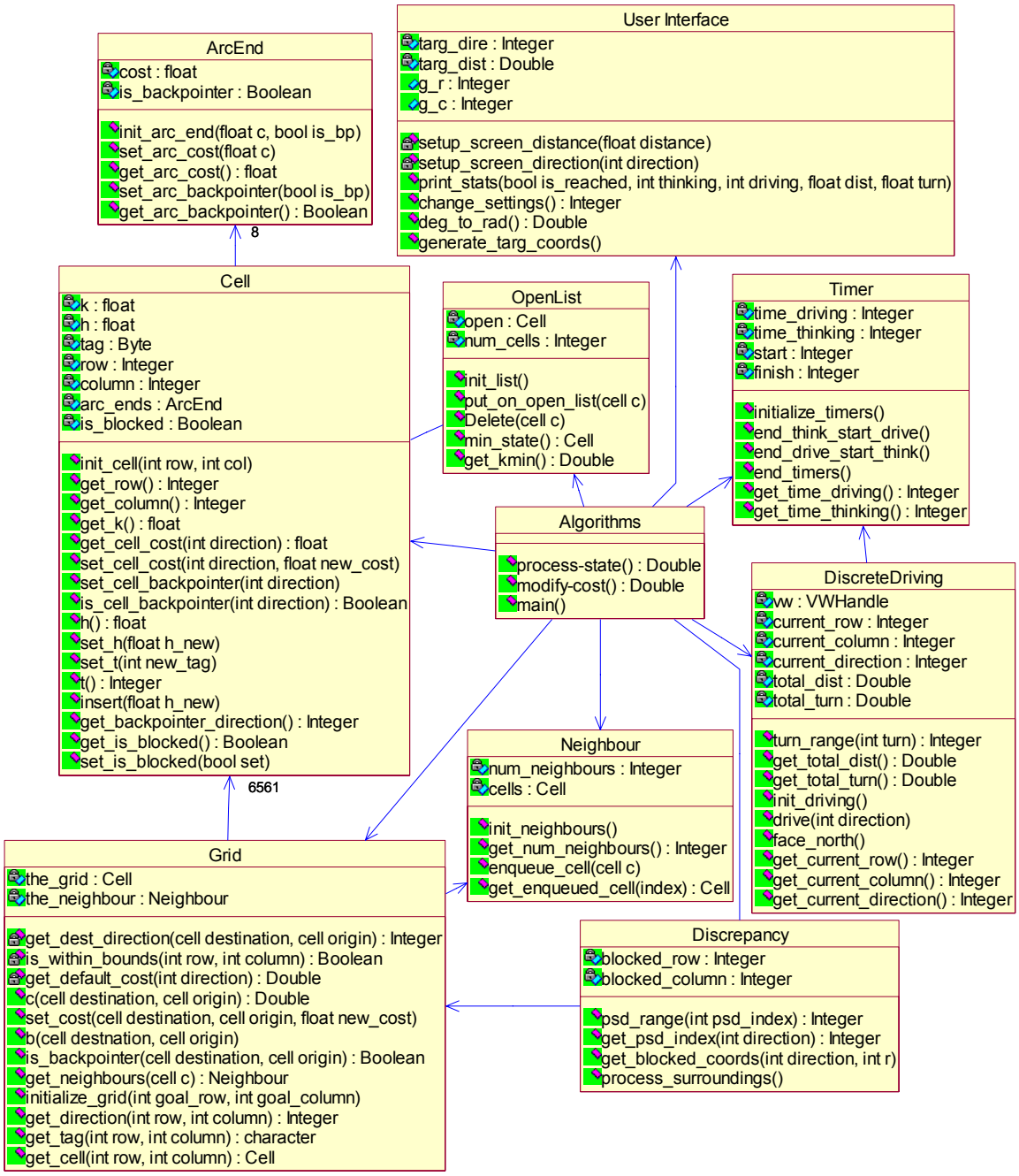


Figure 4-23 The D* class diagram

4.8.1 The cell class

A cell represents an area which is treated as a discrete location. Although this area can be of arbitrary shape, it has been implemented as a square of length 100mm.

Each cell records:

- its position on the grid. This is purely for identification, cells do not need to know their position and behave in the same manner regardless of position.
- its h value, as specified by the original article. This represents the cost of reaching the goal following the cell's current backpointer trail.
- its k value, as specified by the original article. This represents the lowest cost of reaching the goal ever recorded by the cell.
- its arc-ends. Each cell possesses 8 arc-end objects to record transition costs and backpointers.
- if an obstacle exists on its position. If so, blocked will be true.
- its tag, as specified by the original article. This can be one of three possible values: closed, open and new.
    - Closed means that process-state() has been run on that cell. This implies that the cell has a backpointer and a minimum cost to goal established.
    - Open means that the cell is a neighbour of the goal cell or a cell which is closed. Open cells are continually evaluated for minimum cost to goal in the table fashion described in section 3.7. Once an open cell has the minimum cost on the table, process-state() is called and it becomes a closed cell.
    - New is the initial cell state and refers to cells which have not been processed and are not neighbours of closed cells.

The cell class implements three functions required by the original article. The h() and t() methods return the cell's h value and tag respectively. The insert() method updates the tags, k and h values.

4.8.2 The arc-end class

Each cell possesses 8 arc-end objects, one for each direction: north, north-east, east, south-east, south, south-west, west and north-west. Each arc-end object stores the cost of moving from that particular cell in a specified direction. In addition, arc-end stores whether the specified direction is the backpointer for the owning cell.

4.8.3 The open-list class

The open-list class maintains a table of open cells sorted by ascending k value similar to the tables in section 3.7. It is implemented as a large array of cell pointers with the number of elements equal to the number of cells on the grid. When a new cell is to be enqueued, it is sorted according to its k value.

The open-list class implements the min-state() and get_kmin() calls prescribed by the original article. min_state() returns the state with the minimum k value and is implemented by returning a pointer to the cell on top of the list. get_kmin() returns the minimum k value and is implemented by querying and returning the k value of the cell on top of the list.

The delete(cell x) function is also implemented by this class. Although this function is supposed to remove any given cell from the open-list, the implementation disregards the parameter and simply deletes the cell with the minimum k-value, which always the cell at the top of the list. This is because only process-state() calls this function and the only time when process-state() calls delete() is when it is deleting the cell with the minimum k-value.

4.8.4 The grid class

The grid class is composed of all cells in a grid-like formation analogous to the grid diagrams in section 3.7. Since each cell is unaware of any other cell, the grid class serves as an interface when a caller requires operations conducted between two or more cells. This is particularly important when interfacing with functions prescribed by the original article.

A function prescribed by the original article is c(cell destination, cell origin) which returns the travel cost from the target cell to the destination cell. Figure 4-24 shows how the grid class handles the call.

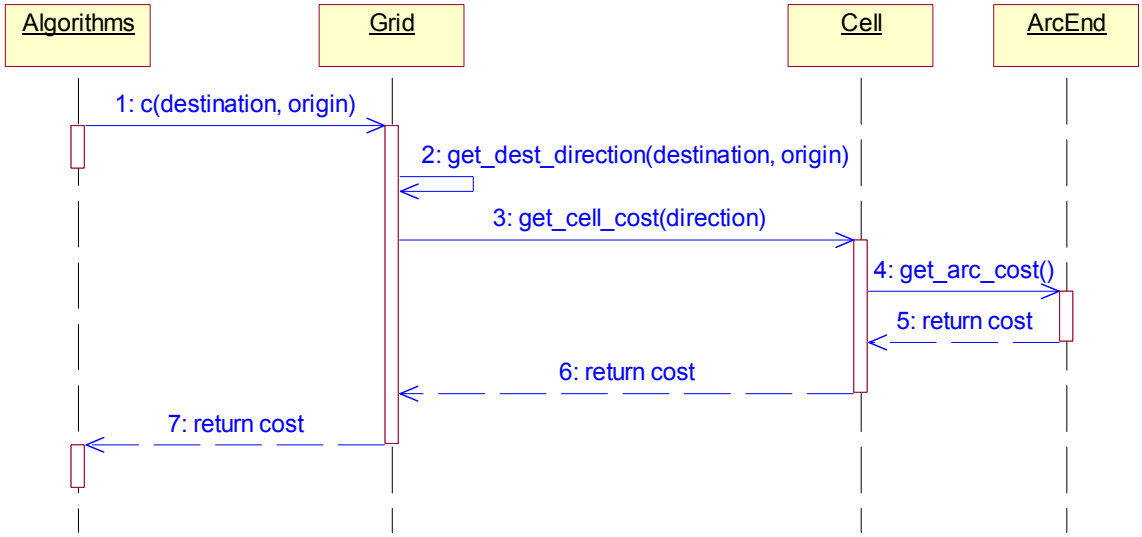


Figure 4-24 The sequence diagram for the c call.

Another function prescribed by the original article is b(destination, origin) which sets the origin's backpointer in the destination's direction. Figure 4-25 shows how the grid class handles the call.

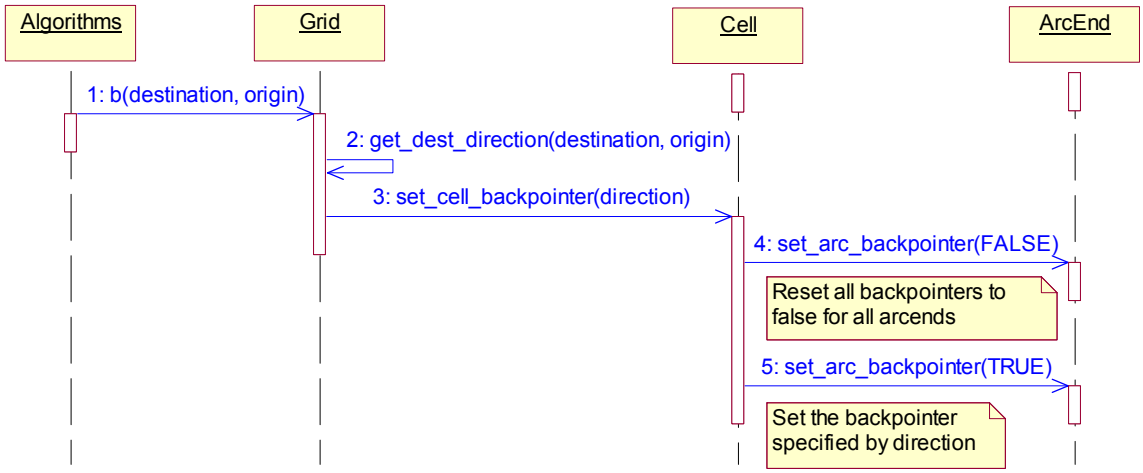


Figure 4-25 The sequence diagram for the b call.

4.8.5 The neighbour class

The neighbour class is a small data-structure designed to facilitate the transfer of valid neighbours surrounding a target cell.

4.8.6 The discrepancy class

The discrepancy class's role is to use the PSD sensors to detect any differences between the robot's map and the actual surroundings. If it detects a discrepancy, it calculates the cell's position based on the PSD reading. Then, it sets the cell's blocked attribute to TRUE and calls modify-cost() to generate the new optimal backpointer trail according to the procedure outlined in section 3.7.2.

4.8.7 The algorithm class

The algorithm class implements process-state() and modify-cost() functions exactly as specified in Stentz's article. process-state() and modify-cost() call functions implemented in the modules discussed previously. The main() method is also included in the algorithm class and it coordinates navigation as a whole. Figure 4-26 shows the main() method.

```
num_calls=0;
generate_target_coords();
initialize_timers();
init_driving();
initialize_grid(g_r,g_c);
do{
       kmin = process_state();
}
while(get_tag(get_current_row(), get_current_column()) != CLOSED && kmin!=NONE);
while(!(get_current_row()==g_r && get_current_column()==g_c)){
       process_surroundings();
       drive(get_direction(get_current_row(), get_current_column()));
}
face_north();
end_timers();
print_stats(TRUE, get_time_thinking(), get_time_driving(),get_total_dist(),
get_total_turn(), num_calls);
```

Figure 4-26 the main method

4.9 Implementation issues

4.9.1 Evaluation of Navigation conditions

Evaluation of navigation conditions involves checking conditions specified by the navigation algorithm. For example, updating *F* by calling freespace() or checking if the robot has reached the M line. In theory, checking should occur continuously. That is, as soon as a leaving condition holds, the robot should instantaneously realise it and take appropriate action. Obviously, this cannot occur in practice.

In practice, navigation conditions are regularly polled because it is impossible to generate an interrupt which doesn't rely on polling at some fundamental level since this is measurement of the external environment. The issue which needs to be resolved is how often (in terms of distance or time) the robot evaluates navigation conditions.

In figure 4-27(a) the robot is polling too seldom and does not utilise the shortcut to goal. In figure 4-27(b) the robot polls when it is over the gap allowing it to utilise the shortcut. However, polling too often is computationally expensive. Therefore, the correct balance must be found between the two extremes.

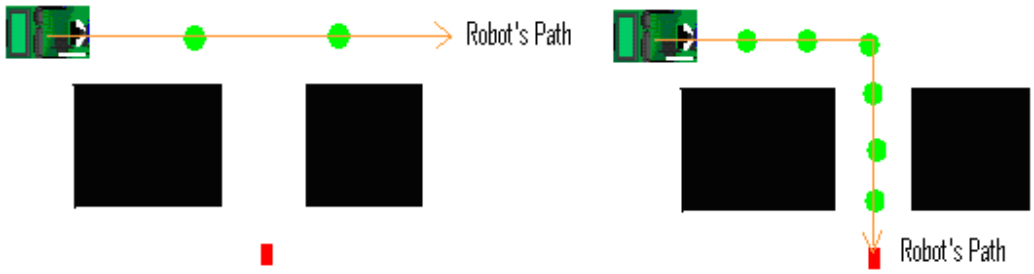


Figure 4-27 The green circles represent polling locations. (a) Left. The robot does not take advantage of a shortcut because it polls too infrequently. (b) Right. The robot takes advantage of a shortcut but it polls too frequently.

In this particular implementation, the robot takes a step of 40mm between each evaluation. It was found through numerous functionality iterations that this value achieved a good balance; however it is by no means perfect. It can be altered using the `STEP` macro.

In light of this implementation issue, algorithms which are proactive rather than reactive should be favoured. That is, algorithms which decide on fixed traveling destinations, travel to them, collect data and then decide on the next destination should be favoured over those which react to the external environment spontaneously.

The bug1, bug2, alg1, alg2 and distbug algorithms are all reactive since they all take measurements of the external environment according to the step size. The modified tangentbug algorithm is proactive because it navigates to nodes and then evaluates navigation conditions. D* evaluates conditions at each cell and sleeps during moving. Hence, tangentbug and D* are most desirable due to their proactive nature.

4.9.2 Recognition of previous points

The bug1, bug2, alg1, alg2 and distbug algorithms all require recognition of previously visited points. This concept is fine in theory, but it poses implementation problems in practice.

In practice, when the robot first encounters a point of interest, it records its current coordinates. The problem is that the robot is very unlikely to ever visit those exact coordinates again for two reasons. Firstly, the trail is very unlikely to be the same for any two boundary traversals because of the PD controller. Secondly, the robot drives a small distance forward, its "step", before evaluating navigation conditions as described in section 4.9.1. This means that even if the trail is the same, the robot also has to poll when it is exactly over the previously recorded point.

Due to this problem, all recorded points are given a radius as shown in figure 4-28. If the robot realises that it is within the radius of a previous point, it deems that it is at the previous point.

Obviously, the radius must be chosen such that false positives are minimised and all genuine previous points are maximised. If the radius is too large, false positives can be generated. For instance, in figure 4-28 the robot's radius extends beyond the wall. If the robot travels on the opposite side, it could mistakenly believe that it is at a previous hit or leave point. Conversely, if the radius is too small, genuine previous points can be missed.

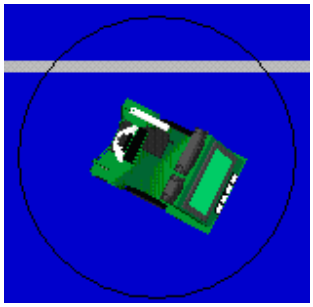


Figure 4-28 The radius is too large and extends beyond the wall.

In this particular implementation, the radius is set to 80mm inside the macro `TARG_ERROR`. This value was established through numerous functionality iterations. However, it is by no means a perfect solution.

Given this implementation issue, algorithms which rely very little on previous points should be favoured over algorithms which rely on previous points heavily. The Alg1 and Alg2 algorithms store all the previous points making them particularly undesirable. The Bug1, Bug2 and Distbug algorithms rely on previous points to a lesser extent.

In contrast, D* tracks the robot's position within the grid and does not require previous point recognition making it highly desirable. Tangentbug relies very little on previous point recognition, only using it to determine if the robot has completely encircled the obstacle. However, unlike the other bug algorithms, Tangentbug uses PSD measurements to determine if it can see a previous point. This is desirable because the radius size can be much smaller and it does not rely on polling.

4.9.3 Hardware requirements

From a practical perspective, hardware costs should be minimised. The bug1 and bug2 algorithms can operate at peak performance using just tactile sensors. PSDs do not improve performance. Clearly, this is the cheapest option.

The alg1 and alg2 algorithms have a minimum requirement of tactile sensors. When following the wall, they need to know if the robot can travel in the target's direction. This is possible with tactile sensors; however it may enhance performance if several PSDs are used to measure *F* as depicted in figure 4-13. Even better is a PSD mounted on a servo which always points in the target's direction.

The distbug algorithm has fairly similar requirements to alg1 and alg2 except that it must have at least one PSD to operate. A PSD mounted on a servo would be ideal, but if not, multiple PSDs should be employed to measure *F*.

Tangentbug and D* algorithms require at least one PSD to operate, with more PSDs improving performance. The PSD range is also an important factor in these algorithms because they do not use it simply to it to determine *F*. In tangentbug, a longer PSD range results in a larger LTG graph, and this may result in shorter paths. In D*, a longer PSD range allows the robot to detect obstacles further away allowing the map to be built more efficiently.

4.9.4 Tangentbug's data gathering

Ideally, $r(\theta)$ would be a continuous function. However, PSD readings can only measure distance in a fixed direction and therefore $r(\theta)$ must be discretely sampled. Unfortunately, this leads to error.

To reduce error, degrees between samples can be decreased at the expense of increased data collection time, increased memory requirements and decreased robustness because it is error-prone to rotate in small angles.

This problem is manifested when the robot is placed near a wall. Figure 4-29 demonstrates that the tangentbug mistakenly identifies nodes. To reduce this error, the robot checks if it is near a wall before collecting data. If so, it drives backwards slightly, making it less likely there will be an incorrect identification. This behaviour is demonstrated in figure 4-30.

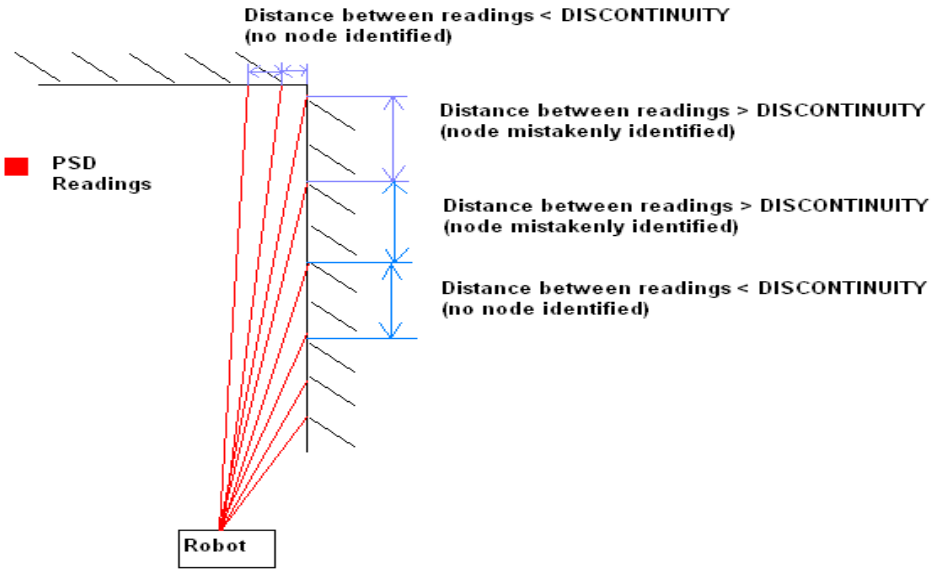


Figure 4-29 The robot mistakenly identifies nodes.

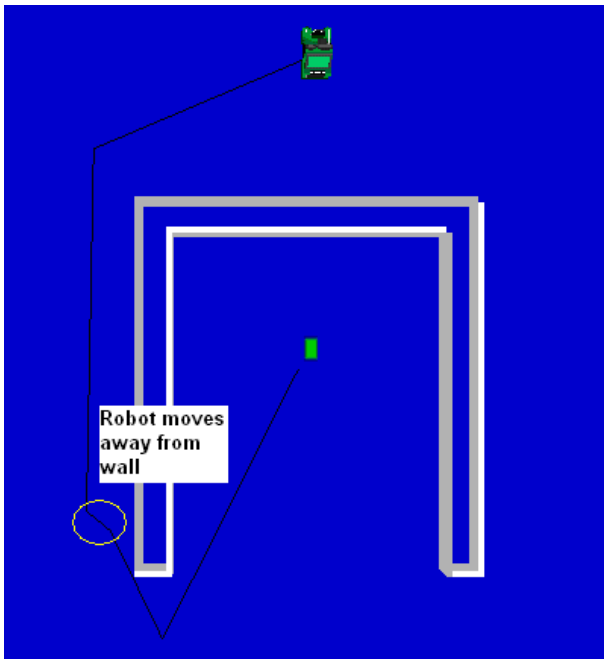


Figure 4-30 The robot moves away from the wall when it is too close.

The data gathering technique can be improved. For example, a variable threshold can be implemented which increases when the differences become larger. Another possible technique is to vary the angle between PSD readings depending on the relative difference. The merits of these techniques are left for future study.

4.3.5 Limited map size in D*

Due to its reliance on cells, D* has a unique problem with map out-of-bound areas. Consider the environment depicted in figure 4-31. The target is reachable, however D* concludes that the target is unreachable because no cells cover the areas around the wall. Clearly, strategies must be developed to handle these situations.

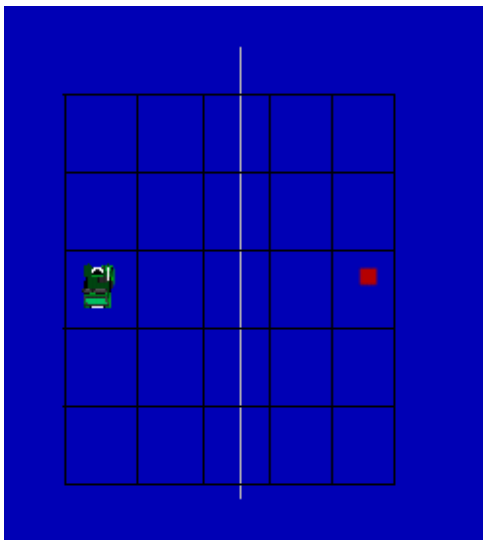


Figure 4-31 using a 5x5 grid, the robot is unable to reach the target.

A possible approach to solving this problem is to provide a cell for each possible position as suggested in Stentz's original paper [10]. However, several issues arise when this happens. Firstly, it implies that the map size is known prior to starting. Clearly, this is not permitted in the problem statement defined in section 2. Secondly, it also implies that the map size is finite, though this may not necessarily be the case. Thirdly, under this scheme, D*'s memory use would be far greater than it is already. A better approach may be to dynamically allocate cells as the robot is moving. Exactly how this scheme is implemented is left for future research.

5. **Simulations**

5.1 Measuring Attributes

Simulations were conducted on the Eyesim simulator to meet the objectives stated in section 2. Convergence is verified and then the attributes are measured. The path length, rotation, PSD limit, linear VW limit and rotational VW limit are constant throughout the 10 simulations. However, the computation and driving time varies between simulations. Therefore, each algorithm is simulated 10 times and the average is taken for computation time.

It is also important to note that clock() measures the time spent in the processor by the calling thread. It does not take into account the simulation-to-time ratio. Hence, if the simulation-to-time ratio is increased for some simulations and not others, the results will be inaccurate. To account for this, all simulations are run on the maximum simulation-to-time ratio setting.

Each algorithm is simulated on three different maps which are designed to illustrate their individual strengths and weaknesses.

5.2 Simulation Maps

5.2.1 Map1

Map1 is shown in figure 5-1 where the coordinates are in millimetres from the bottom left. It is designed to test algorithms in an enclosed setting. In enclosed settings, algorithms require the ability to choose good leaving points and correct bad wall-following decisions. It has been used in previous studies [3, 4].

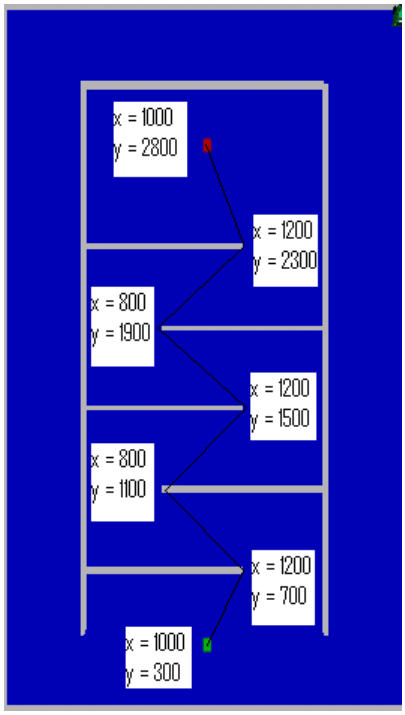


Figure 5-1 Map1 tests an algorithm in an enclosed setting

For interest and comparison purposes, the ideal path length and rotation have been calculated:

Distance: $\frac{\sqrt{20}}{10} + \frac{4\sqrt{32}}{10} + \frac{\sqrt{29}}{10} = 3.248 \text{ m}$

Rotation: $2\left(\tan^{-1}\left(\frac{200}{400}\right) + 4\tan^{-1}\left(\frac{400}{400}\right) + \tan^{-1}\left(\frac{200}{500}\right)\right) = 7.971 \text{ Radians}$

5.2.2 Map2

Map 2 is shown in figure 5-2. It is regarded as an open setting. In an open setting, an algorithm's ability to seek out and follow greedy, locally optimal paths is tested. This map was used by Lumelsky in demonstrating the original bug1 and bug2 algorithms [5].

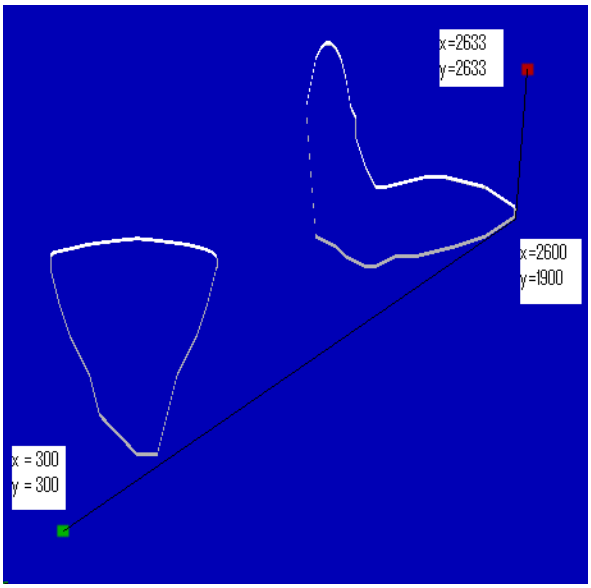


Figure 5-2 Map2 tests the algorithm on an open setting

The ideal path lengths and minimum rotation have been computed:

Distance: $\sqrt{2.3^2+1.6^2}+\sqrt{0.033^2+0.763^2}=3.5647$ m

Rotation: $2\tan^{-1}\left(\frac{2300}{1600}\right)=1.926$ Radians

5.2.3 Map3

Map3 is shown in figure 5-3. It tests a very important ability for all convergent algorithms, to escape from a local minimum.

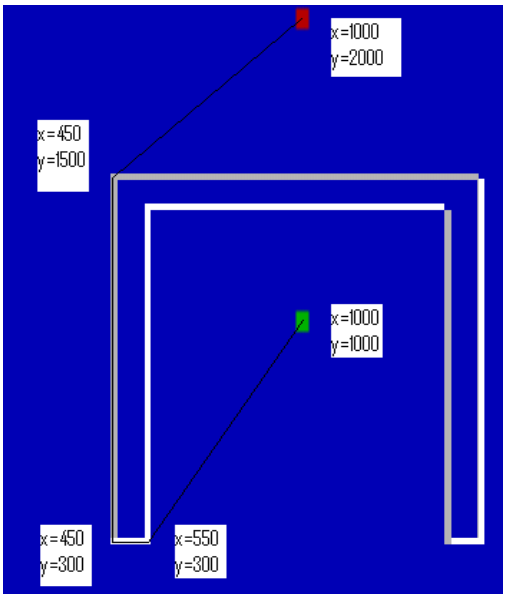


Figure 5-3 Map3 tests an algorithm in a local minimum

The ideal path length and rotation have been computed:

Distance: $\sqrt{0.45^2 + 0.7^2} + 0.1 + 1.2 + \sqrt{0.55^2 + 0.5^2} = 2.875$ m

Rotation: $\frac{\pi}{2} + \tan^{-1}\left(\frac{700}{450}\right) + \tan^{-1}\left(\frac{700}{450}\right) + \frac{\pi}{2} + \tan^{-1}\left(\frac{550}{500}\right) + \tan^{-1}\left(\frac{550}{500}\right) = 6.808$ Radians

5.3 Floor Generator

The appearance of green and red squares can be observed in the background of simulation maps. These represent the start and goal respectively. This is produced by generating a bitmap file which Eyesim uses as background.

Floor Generator was written specifically to generate the background bitmap file and its GUI is shown in figure 5-4. The program is object-oriented and written in C#.net. It takes input from the user and from the world file. The user specifies the desired bitmap size, the distance to the target and the direction to target. Floor Generator extracts the starting position and the world's size from the world file. Then, it creates a bitmap to specification.

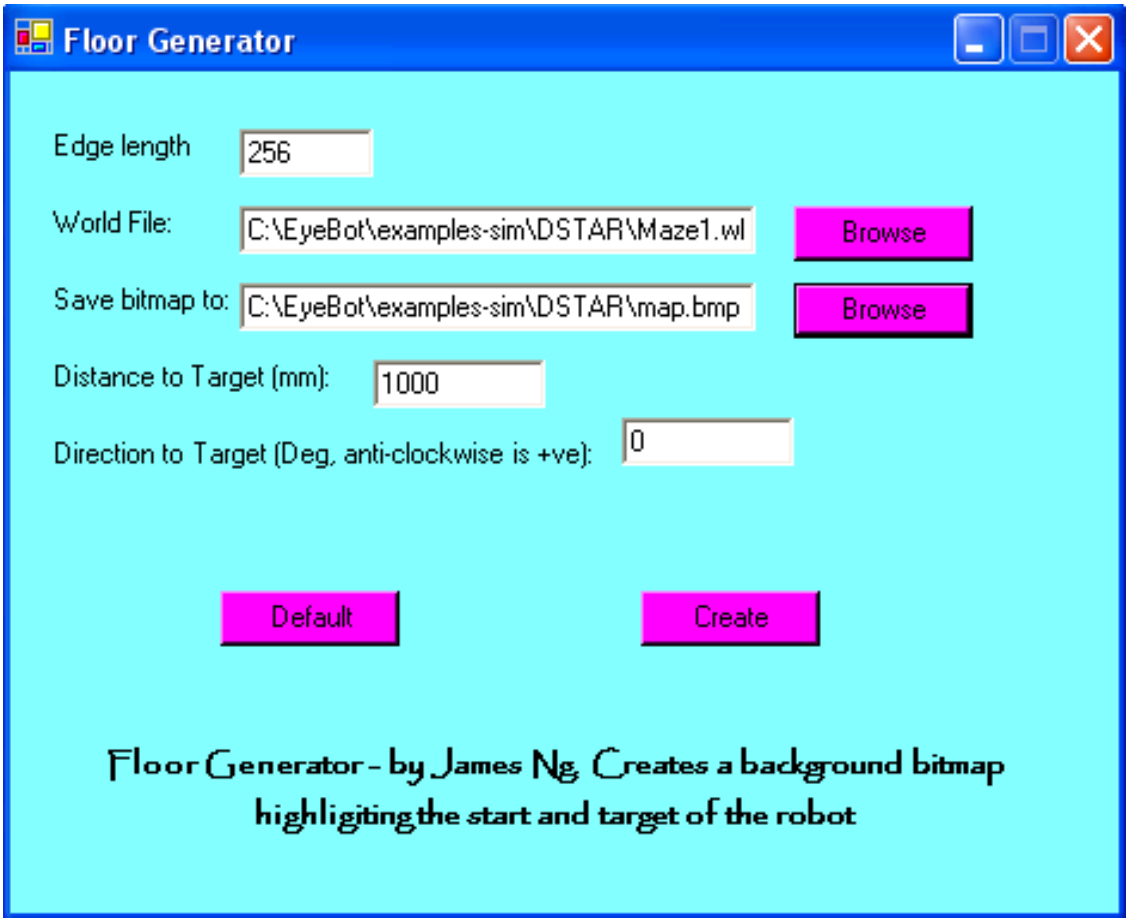


Figure 5-4 The floor generator GUI

6. Results and discussion

6.1 Convergence Verification

The first objective is to verify algorithm convergence. The simulation results show that all algorithms are convergent.

6.1.1 Bug1

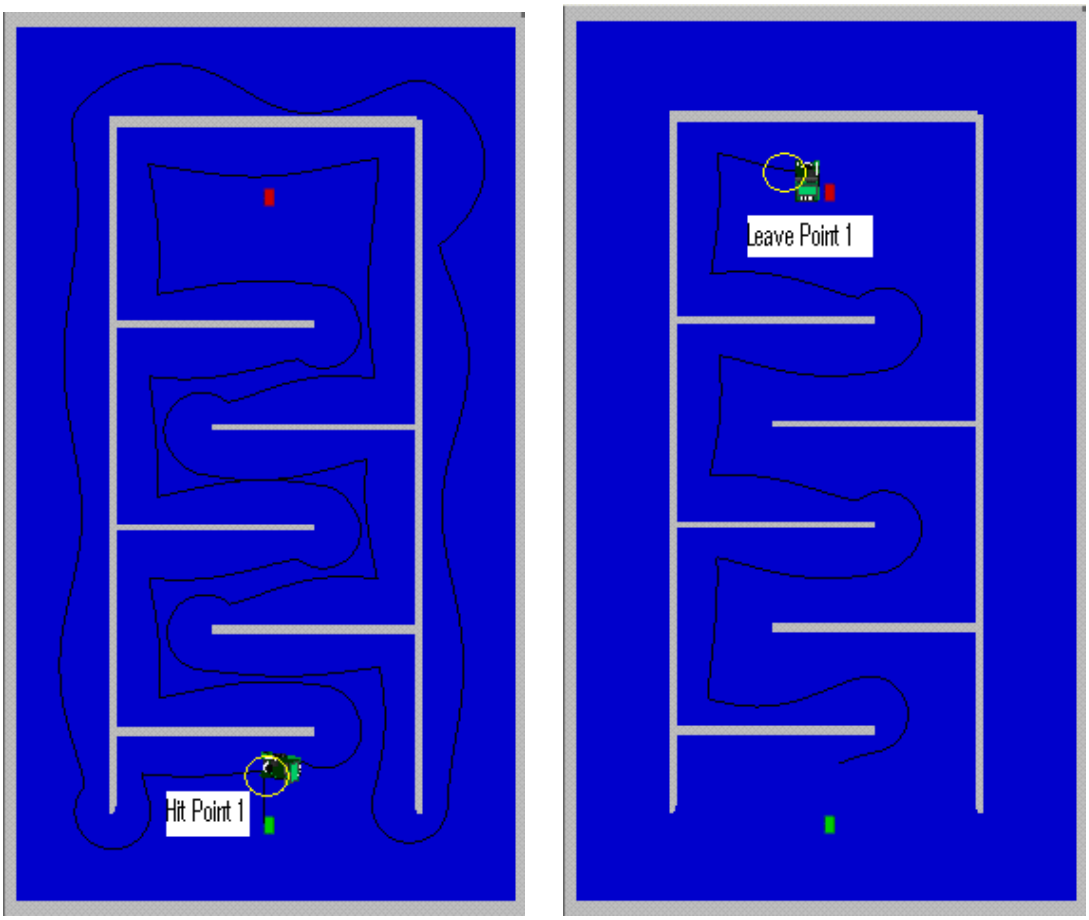


Figure 6-1(a) Bug1 on Map1

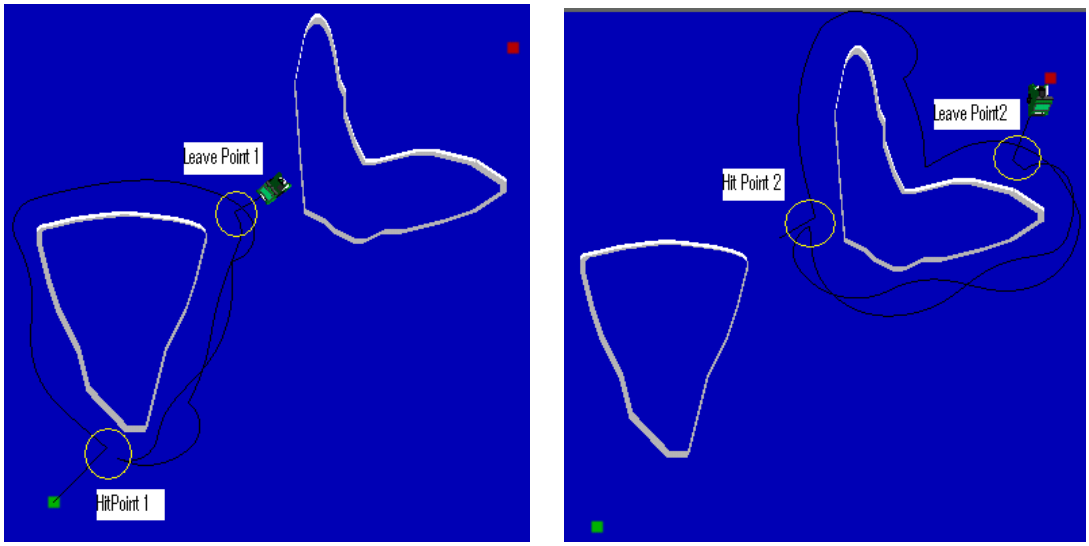


Figure 6-1(b) Bug1 on Map2

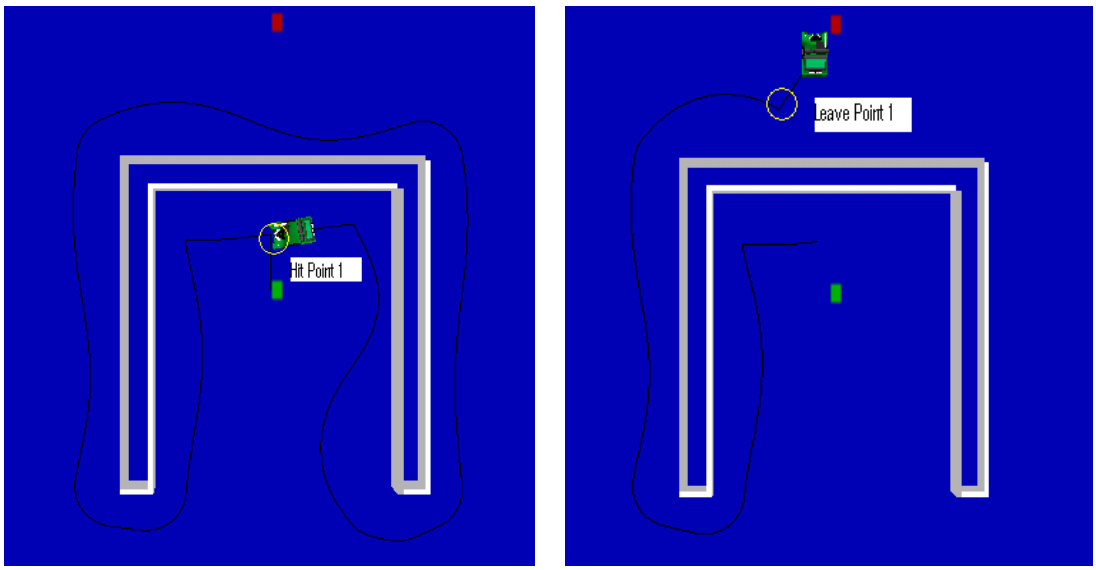


Figure 6-1(c) Bug1 on Map3

6.1.2 Bug2

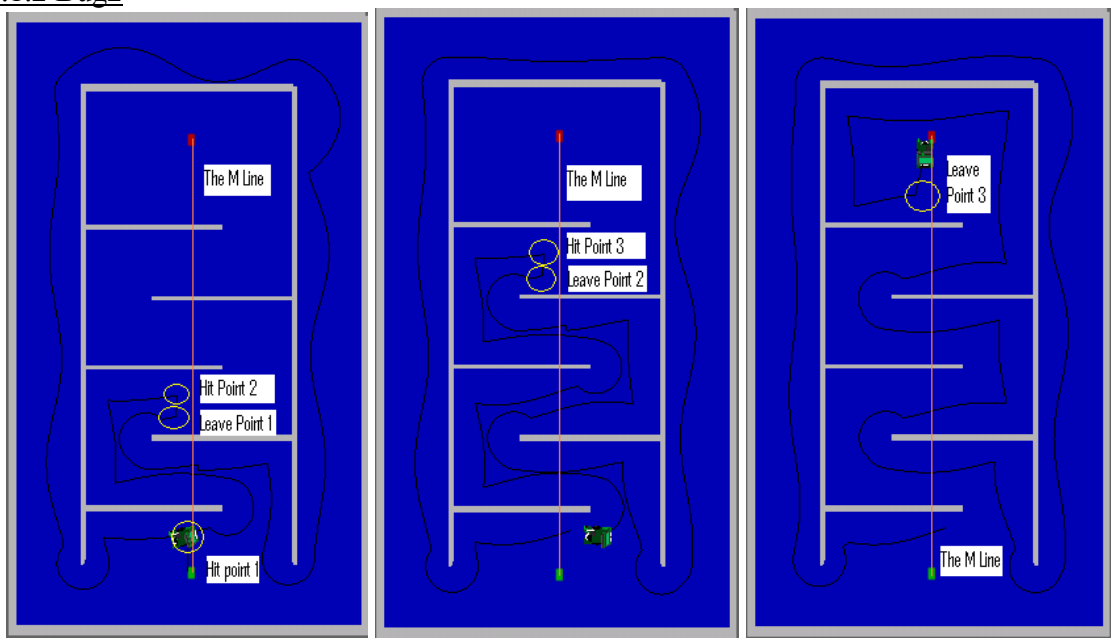


Figure 6-2(a) Bug2 on Map1

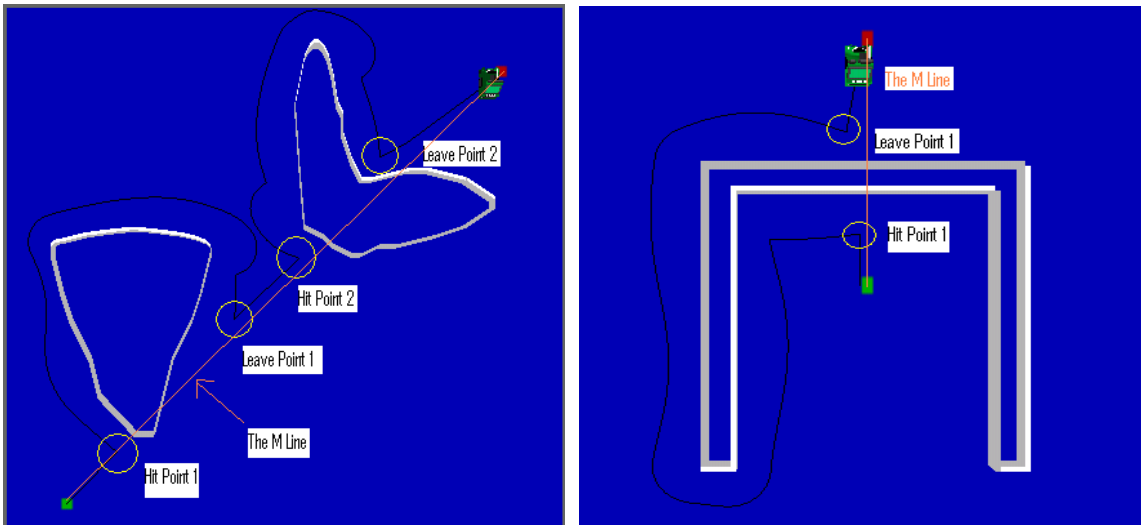


Figure 6-2(b) Left. Bug2 on Map2. (c)Right. Bug2 on Map3.

6.1.3 Alg1

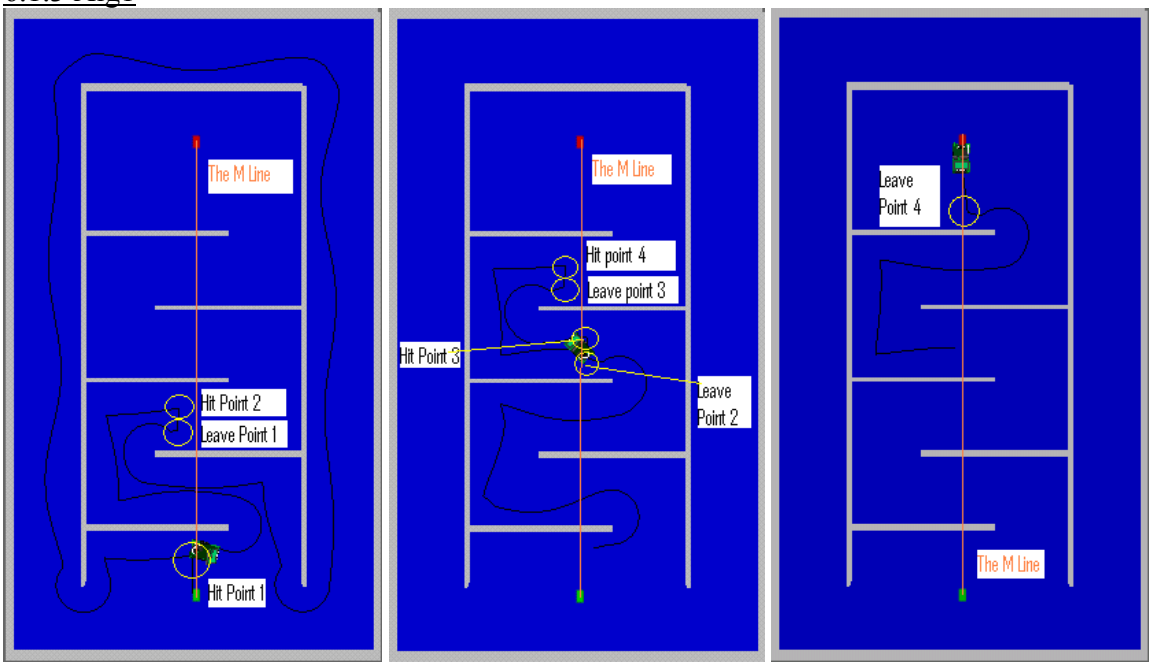


Figure 6-3(a) Alg1 on Map1

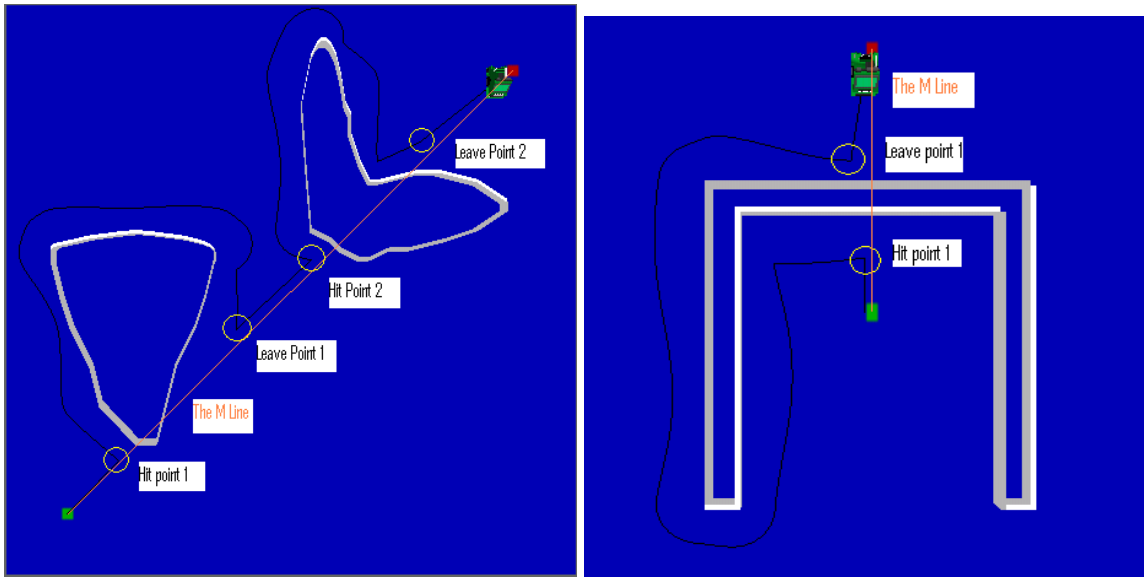


Figure 6-3(b) Left. Alg1 on Map2. (c) Right. Alg1 on Map 3.

6.1.4 Alg2

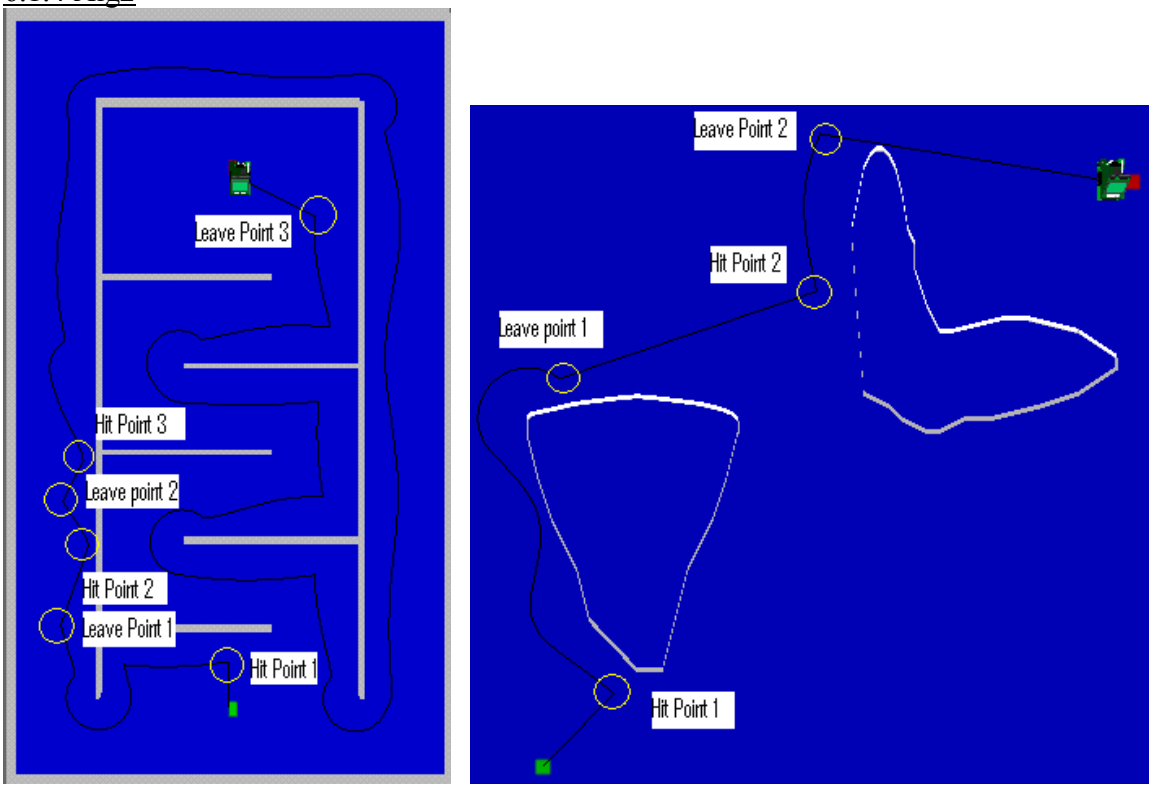


Figure 6-4(a) Left. Alg2 on Map1. (b) Right. Alg2 on Map2.

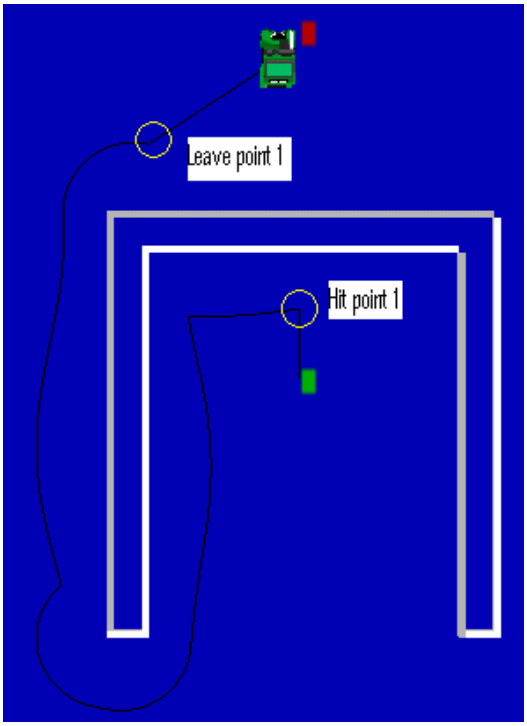


Figure 6-4(c) Alg2 on Map3.

6.1.5 Distbug

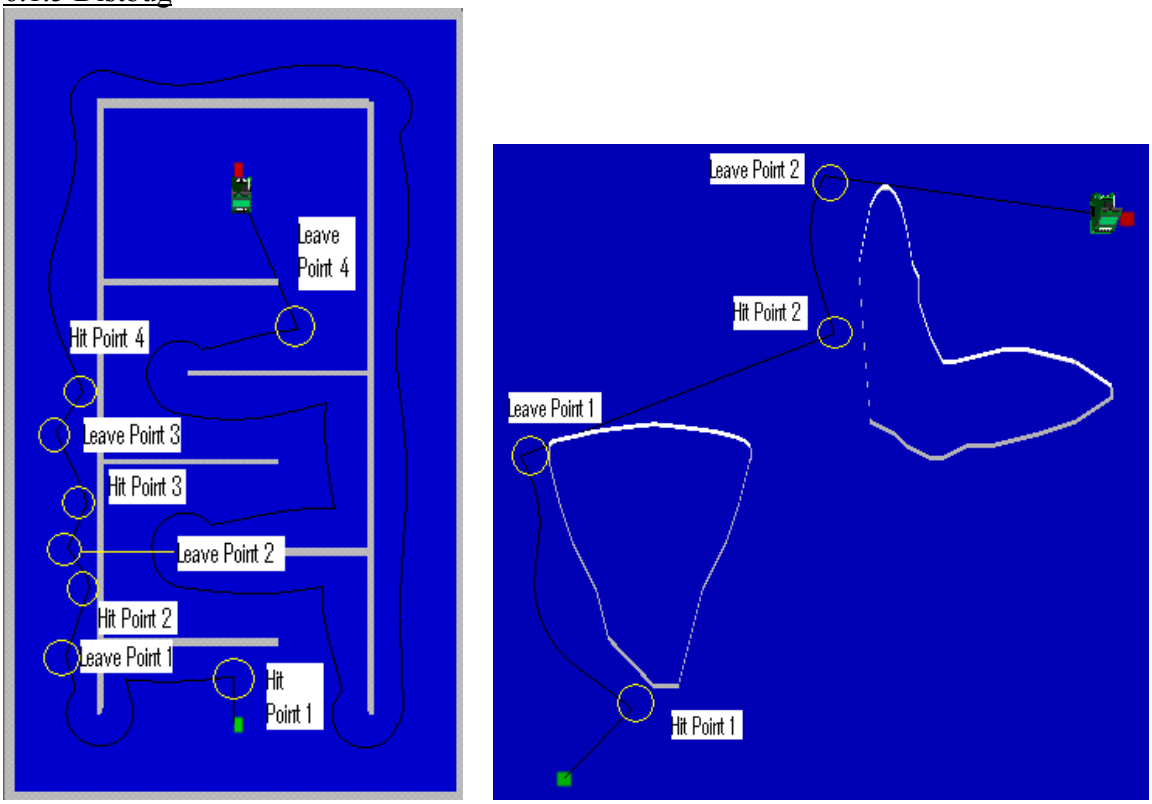


Figure 6-5(a) Left. Distbug on Map1. (b) Right. Distbug on Map2

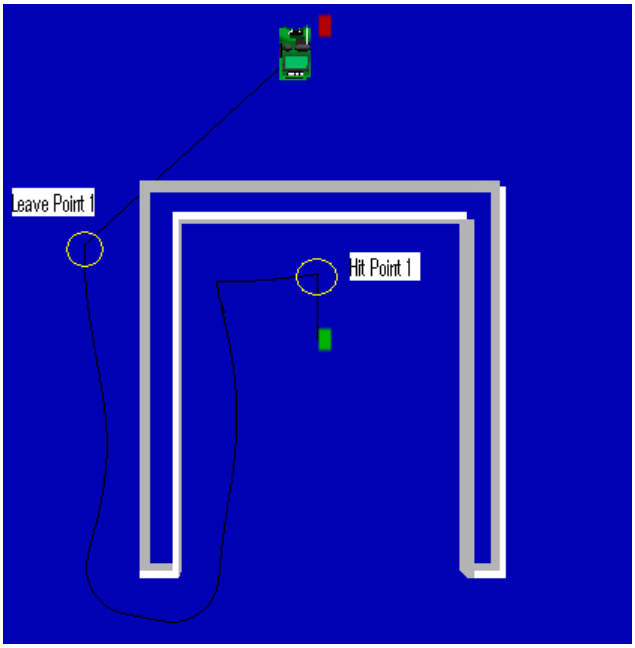


Figure 6-5(c) Distbug on Map3

6.1.6 Tangentbug

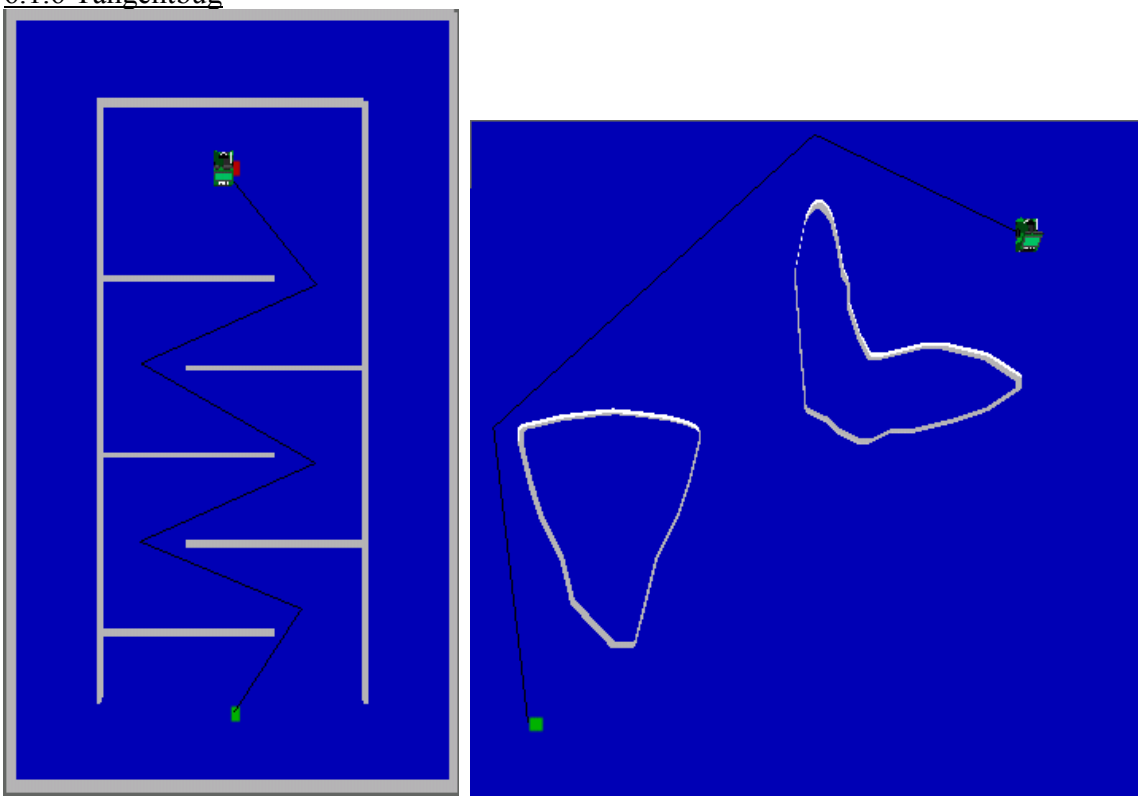


Figure 6-6(a) Left. Tangentbug on Map1. (b) Right. Tangentbug on Map2.

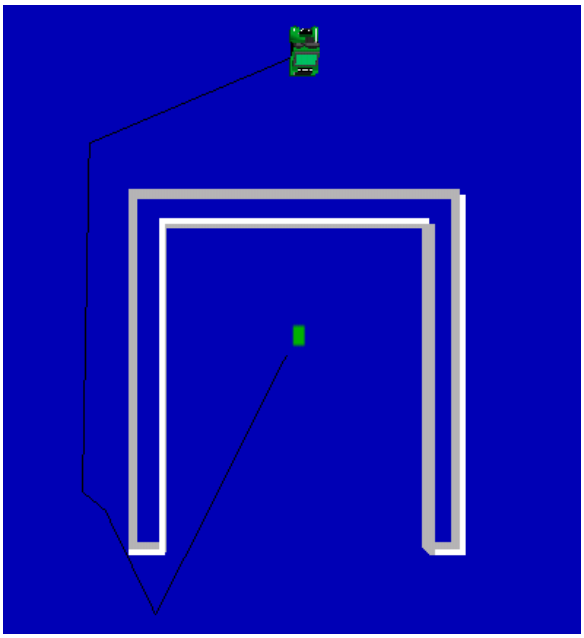


Figure 6-6(c) Tangentbug on Map3.

6.1.7 D*

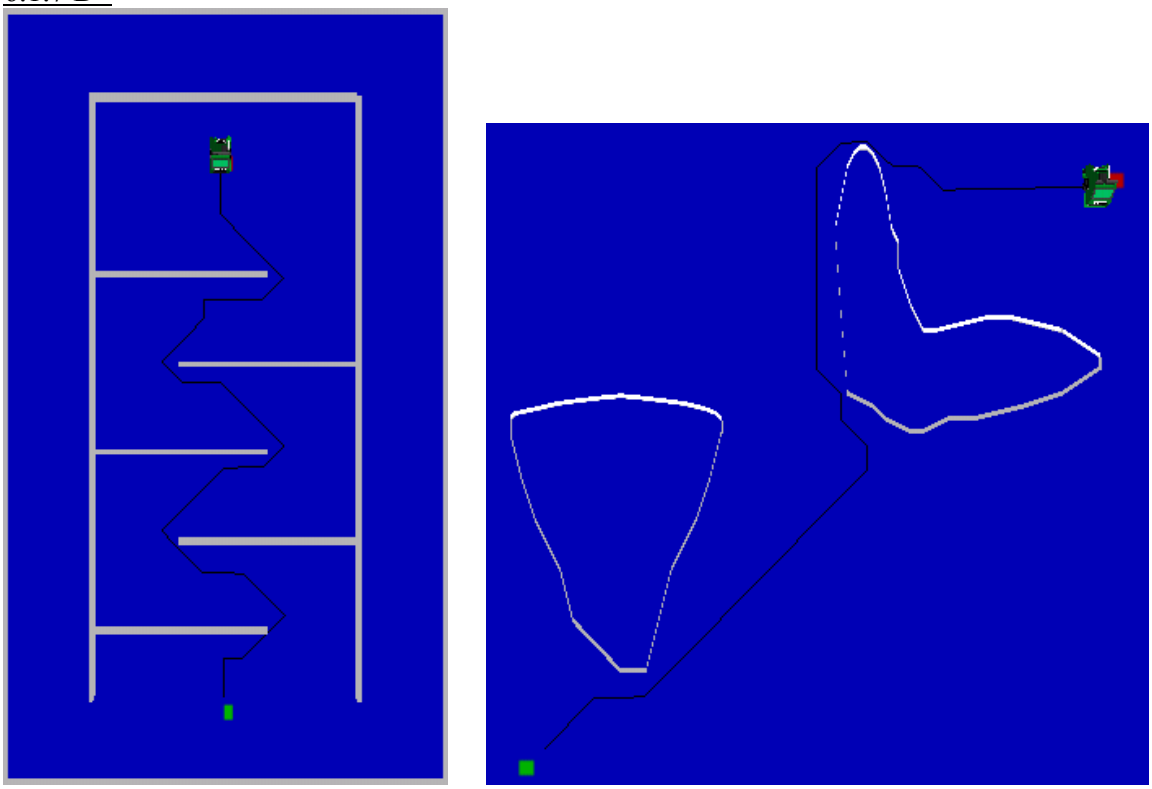


Figure 6-7(a) Left. D* on Map1. (b) Right. D* on Map2.

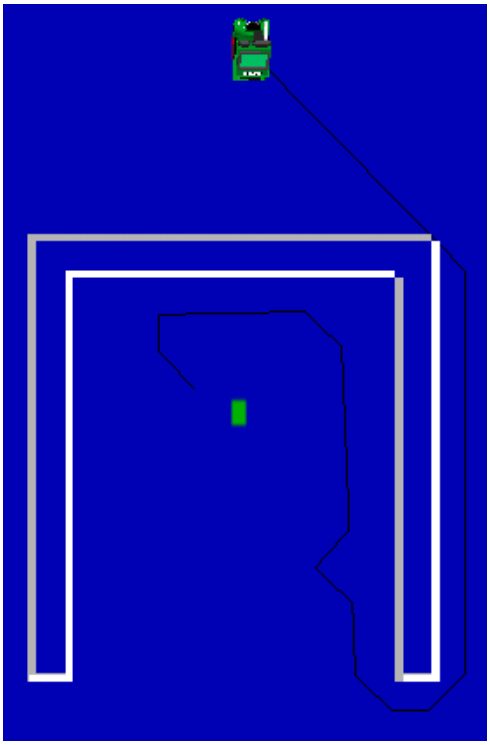


Figure 6-7(c) The D* algorithm on Map3

6.2 Path Length

A low path length is an extremely desirable attribute because it allows the robot to reach the target faster. Also, robots need to be maintained and require a fuel source. Obviously, a low path length reduces both expenditures. The path length depends on map, but figure 6-8 shows that the general descending order is: Bug1, Bug2, Alg1, Alg2, Distbug, Tangentbug and D*.

The results show that bug1 outperformed bug2 on Map1, but bug2 outperformed bug1 on Map2 and Map3. Clearly, the bug1 and bug2 algorithms are suited to particular types of maps. The bug1 algorithm is suited to enclosed maps because it gathers all data before deciding on the leave point. The bug2 algorithm less conservative and will leave the obstacle as soon as it encounters the M-line. This makes it suited to open maps, such as Map2 and Map3.

Then, the results show that Alg1 outperforms bug1 and bug2 on all three maps. This is because Alg1 combines the best features of bug1 and bug2. Whilst retaining the advantage of the M-line for open maps, Alg1 can recall previous hit and leave points. This feature allows it to generate short paths on enclosed maps, such as Map1.

After that, the results show that Alg2 and Distbug performed better than alg1, bug2 and bug1. This is due to their superior choice of leaving points generated by their leaving conditions. Alg2 and Distbug hold back leave points on enclosed maps which gives advantages similar to bug1. However, Alg2 and Distbug make early leave points on open maps which gives advantages similar to bug2.

The results show that Alg2 and Distbug were fairly even. Indeed, the paths on all maps were very similar. This is because of the similarity in their leaving conditions. The leaving condition for Distbug is $d(X,T) - F \leq d_{\min}(T) - Step$. Compare this with the leaving conditions for Alg2: 1) a point y is found such that it is closer to the target than any point ever visited by the robot previously and 2) the robot can travel towards the target at that point.

Assume that the *F*=0 and the robot is at *y*. In Alg2, a leave point can never be generated. In distbug, *F*=0 implies that $d(X,T) \le d_{\min}(T) - Step$. Since the robot is at *y*, $d(X,T) = d_{\min}(T)$. Then, the *STEP* constant biases the comparison such that it can never be true. Assume that *F>STEP* and that the robot is at *y*. In Alg2, a leaving point would be defined. In Distbug, $d(X,T) = d_{\min}(T)$, however *F>STEP*, making condition true.

Therefore, Distbug implicitly specifies that *F* must be greater than *STEP*. Alg2 simply states that the robot must be able to move in the target's direction at *y*, but figure 6-8 shows that its implementation requires a threshold, FS_THESHOLD. This threshold is, in principle, no different than *STEP* in distbug.

```
while(TRUE){
      fs = freespace();
      if(fs==OUT_OF_RANGE){
            fs = MAX_RANGE;
      }
      if(is_at_y && fs>=FS_THRESHOLD){
            return LEAVE_POINT;
      }
      ...
}
```

Figure 6-8 Alg2 requires FS_THRESHOLD value

Alg2 and Distbug's leaving conditions are very similar but they do differ in one subtle aspect. When *F>STEP* and the robot has not quite reached *y*, distbug allows a leave point to be defined, whereas Alg2 requires the robot to reach *y*. This difference is best illustrated by comparing figures 6-4(a) and 6-5(a). Observe leave point 3 in Alg2 and leave point 4 in distbug. It is this subtle difference which has allowed distbug to generate slightly lower path lengths on all three maps.

The performance of Tangentbug and D* was superior to Bug1, Bug2, Alg1, Alg2 and Distbug on all three maps. In fact, they were within close proximity to the ideal path length. D* faltered slightly on Map3 due to its vulnerability in local minimums but is still competitive.

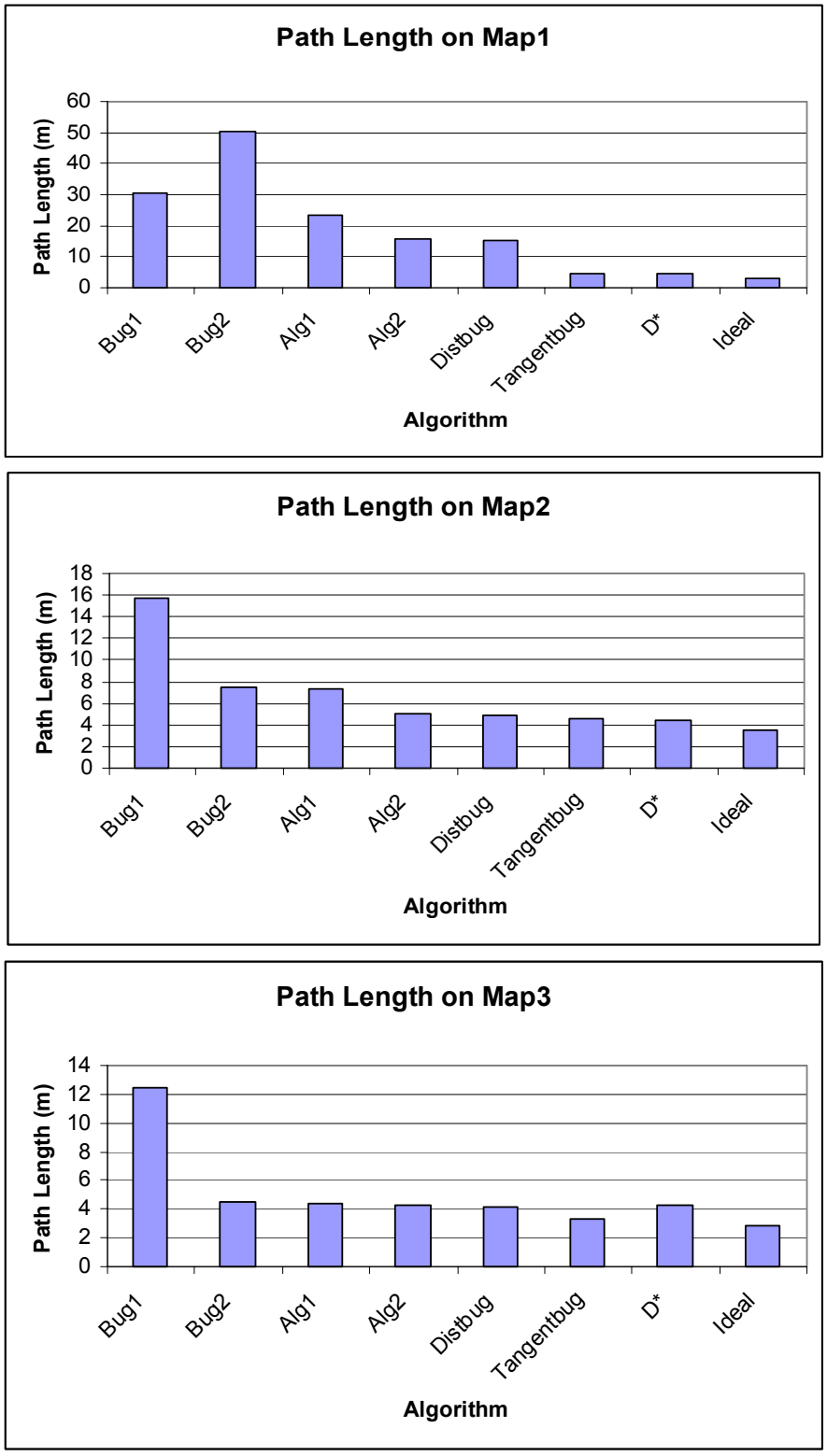


Figure 6-8(a) Top. Path length on Map1. (b) Centre. Path length on Map2. (c) Bottom. Path length on Map3.

6.3 Rotation

Figure 6-9 shows that, in general, rotation is related to path length. That is, the longer an algorithm's generated path, the more rotation is performed. This result is intuitively satisfying.

Alg2 and Distbug stand out as major outliers. Their greatly increased rotation is probably due to the calls to freespace() which are performed whilst following the wall. This call is performed every 40mm and total rotation per call ranges from 0 to 45 degrees. If a servo was mounted which always points in the target's direction or more PSDs are available, then this rotation will decrease.

It may be interesting to remove the hardware-dependant rotation from rotation results. The results will then show rotation which is inherently generated by the algorithm. Figure 6-10 shows that inherent rotation is related to path length. This finding holds for Alg2 and Distbug and shows that Alg2 and Distbug will benefit greatly from a servo or more PSDs.

The only algorithm which does not match this trend is D*. This is due to its segmentation of the map into cells. As a result, the robot can only move in discrete 45 degree turns. Consider figures 6-6(a) and 6-7(a). Tangentbug can maneuver to successive protrusions without rotation, but D* necessarily takes a 45 degree turn each time it travels to successive protrusions.

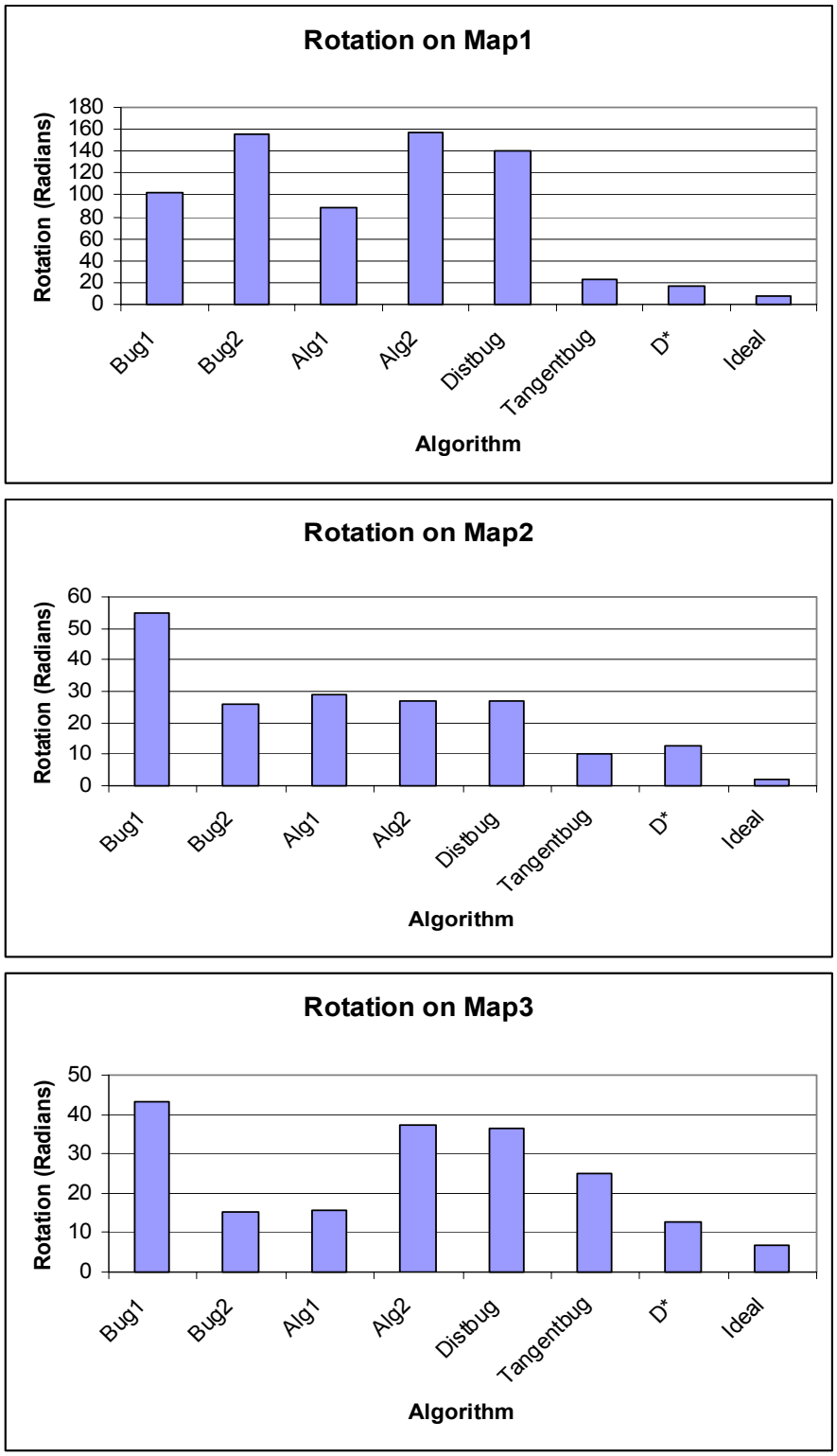


Figure 6-9(a) Top. Rotation on Map1. (b) Centre. Rotation on Map2. (c) Bottom. Rotation on Map3.

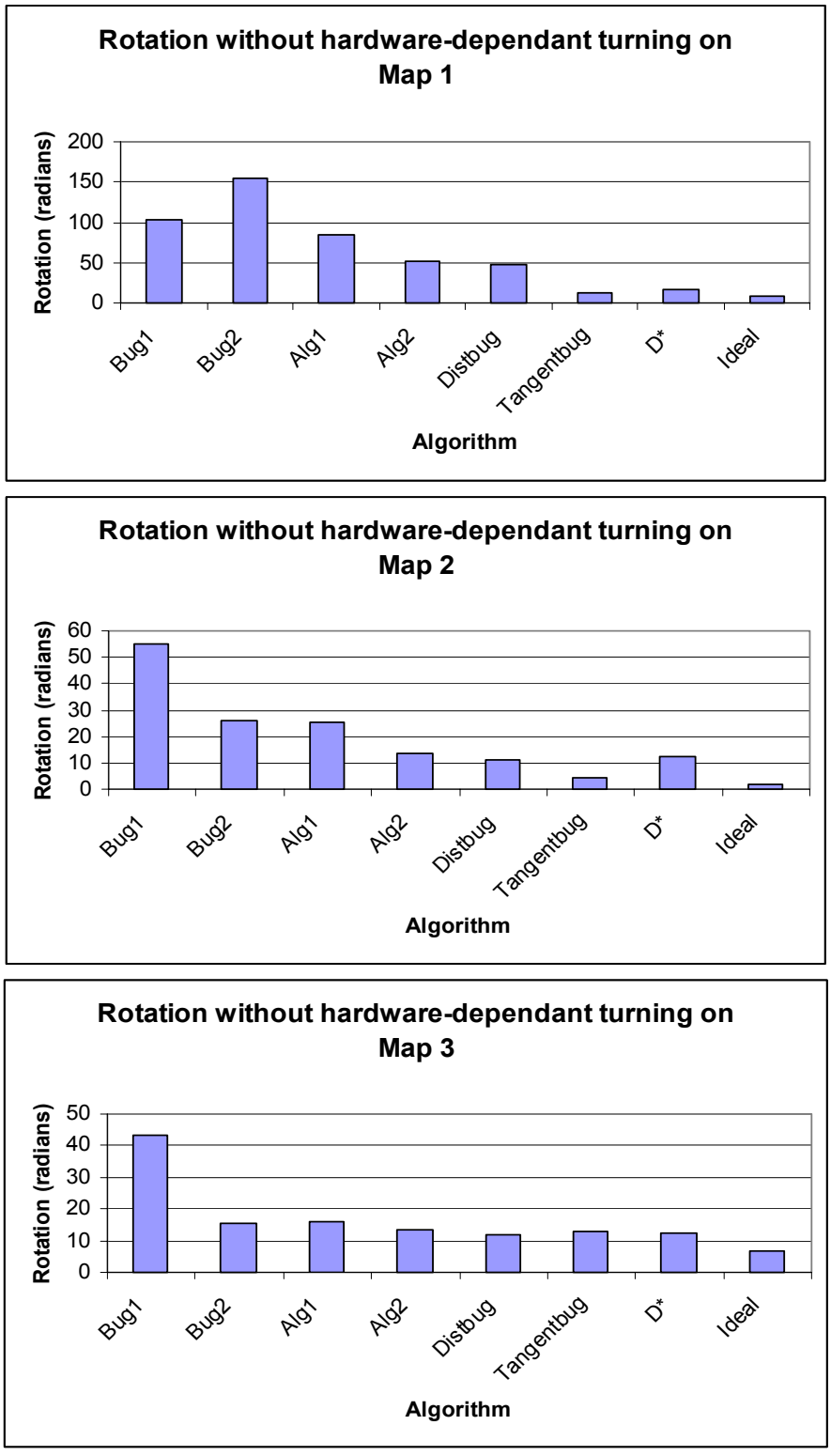


Figure 6-10(a) Top. Rotation on Map1. (b) Centre. Rotation on Map2. (c) Bottom. Rotation on Map3.

6.4 Average computation time

Computation time is an objective measure of the navigation process's CPU usage. Driving time is not as important because it depends on the driving speed of the robot and whilst the robot is driving, the navigation thread can sleep.

Figure 6-11 shows the results of computation time measurements. The results show that increased complexity results in roughly the same or decreased computation time! This is most probably due to less travel resulting from short and near ideal paths. The only major outlier is D*'s performance on Map3 because of its vulnerability to local minimums.

It would be interesting to investigate how much computation is required per metre traveled. This removes the path length advantage of the complex algorithms. This statistic for all 3 maps is shown in figure 6-12.

The trend varies depending on the map. On Map1 and Map2, where are there are no local minimums, D* requires the lowest computation time per metre, even outperforming bug1 and bug2! Then, Bug1 and Bug2 are next because they are relatively simple algorithms. Fairly surprisingly, Tangentbug comes in next followed by distbug, alg2 and then alg1.

D* probably has the least computation per metre due to its map segmentation. This allows D* to abandon reliance on the math library and furthermore, there are no computationally intensive floating point multiplication or division operations. Instead, its reliance on the relatively cheap process-state() method results in low computation times.

Tangentbug is surprisingly competitive. This is most probably due to its proactive nature discussed in section 4.9.1. While it may be expensive to compute the LTG, it performs very little computation whilst driving. Another interesting statistic is that tangentbug calls the math library the least number of times. These two factors may explain tangentbug's relatively low computation time per metre.

Alg1 and Alg2 are the most expensive due to their checking for all previous points at every step. Distbug, which is very similar to Alg2, shows the computation savings of not checking at each step.

However, observe how the trend changes on map3 with a local minimum. Now, bug1 and bug2 are the cheapest, followed by alg1, alg2, distbug, tangentbug, and by far the most expensive, D*. Figure 6-12(c) shows that complexity results in greater computation time per metre.

D*'s computation time is excessively large. It calls process-state() 83855 times in the local minimum resulting in a very large computation time. Unfortunately, this is necessary due to recomputation of the optimal path and is a weakness inherited from its ancestor, the force field technique. Tangentbug's computation time is also very large. It calls the math library 23080 times. This is because it needs to process the visible environment to determine $d_{reach}(T)$.

Tangentbug's large amount of math library calls has not resulted in a proportionately large increase in computation time over other algorithms. This is perhaps due to the fast hardware and parallel processing environment which the simulation was run on. On an actual robot with a single slower processor, the results could be very different.

The parallel processing in tangentbug is only possible due to its proactive nature. Tangentbug gathers all data at one point. The subsequent node processing methods can be processed independently which allows parallel processing to occur. Other algorithms require that robot move a small step before taking measurements. These measurements are inherently sequential and are processed as such.

Unfortunately, D* is unable to exploit parallelism because the optimal backpointer trail must be computed sequentially from the goal as explained in section 3.7. That is, D* is an inherently sequential processing algorithm. A future research topic could focus on whether D* can become a parallel algorithm and if so, changing it.

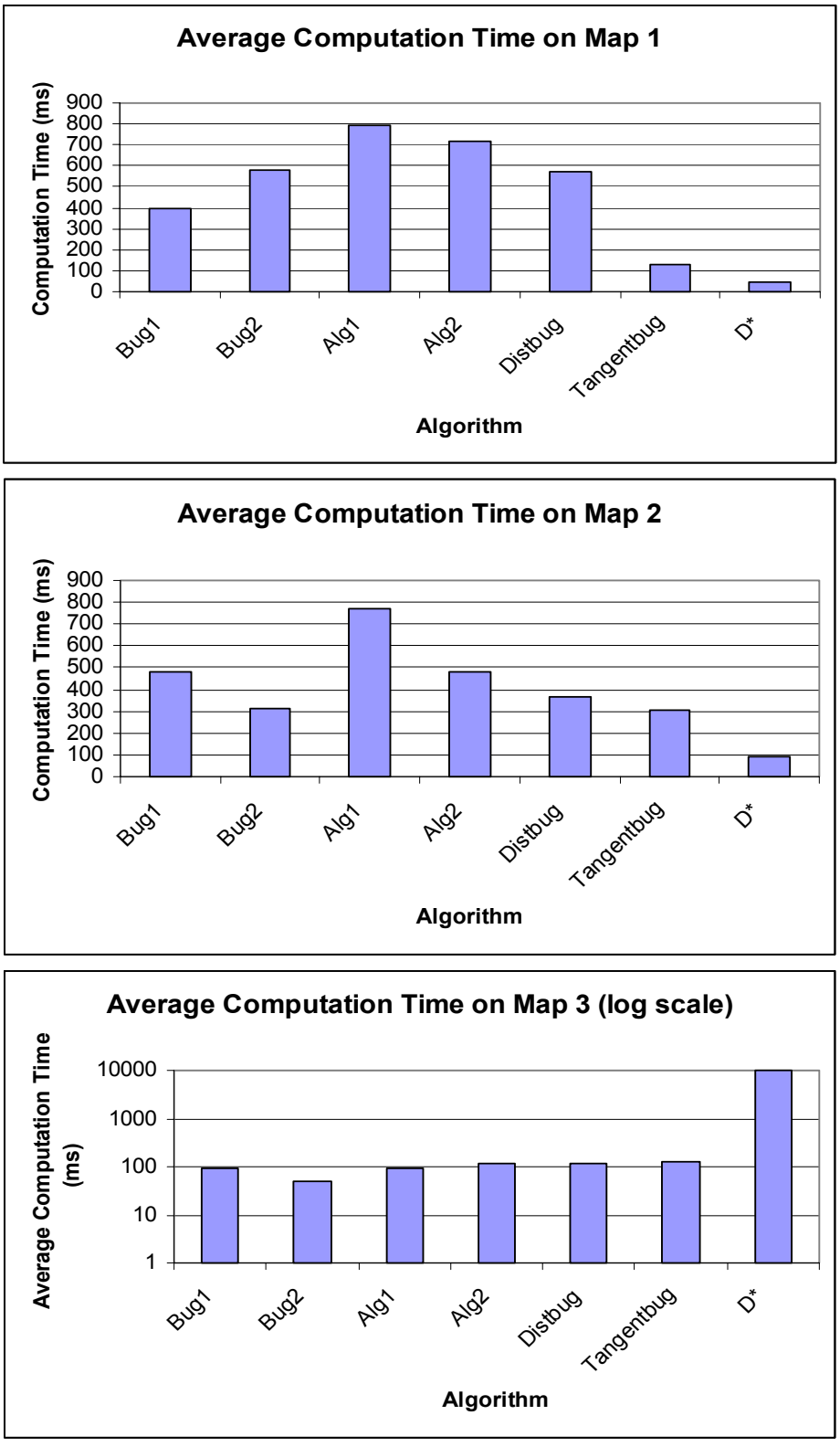


Figure 6-11(a) Top. Computation time on Map1. (b) Centre. Computation time on Map2. (c) Bottom. Computation time on Map3.

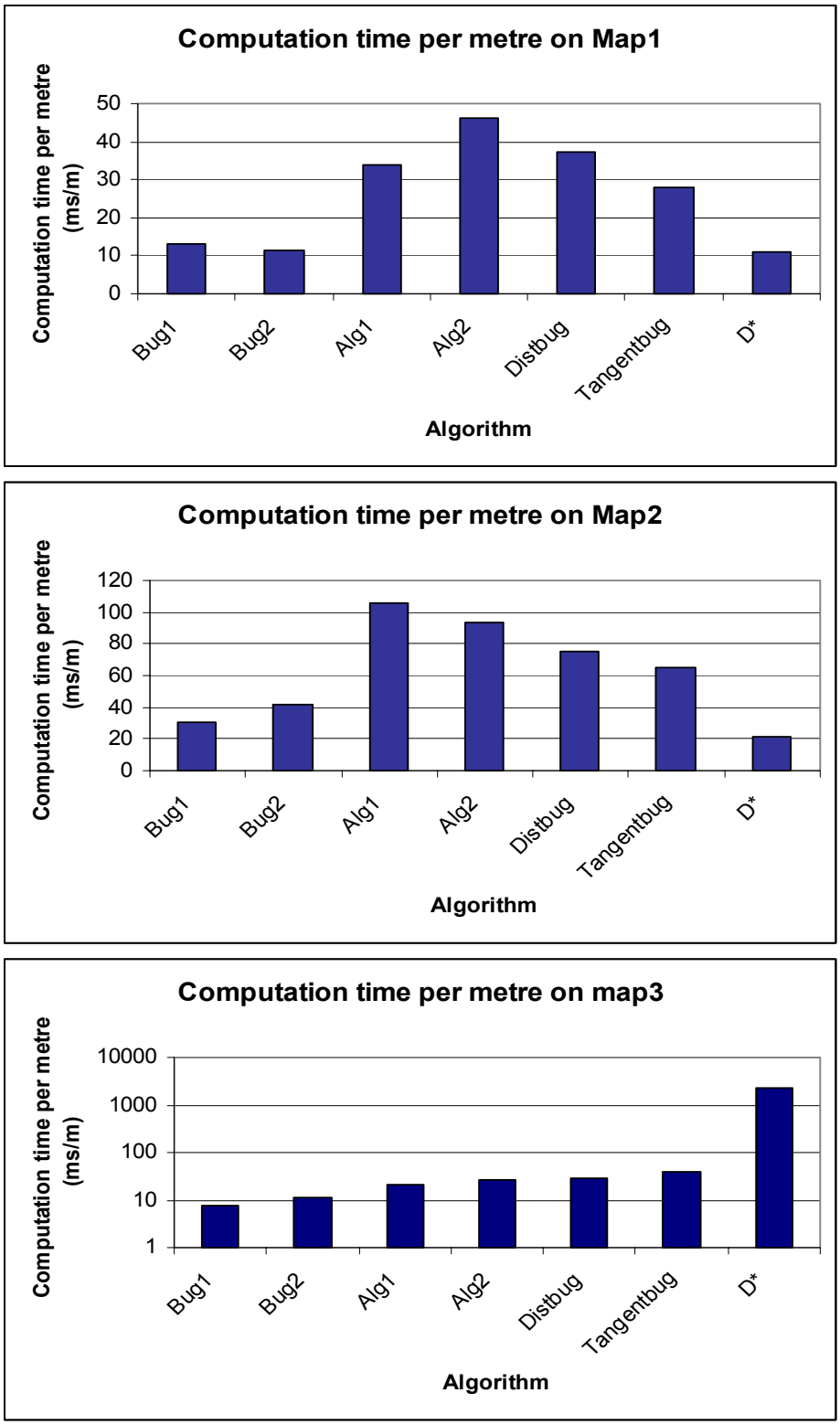


Figure 6-12(a) Top. Computation time per metre on Map1. (b) Centre. Computation time per metre on Map2. (c) Bottom. Computation time per metre on Map3.

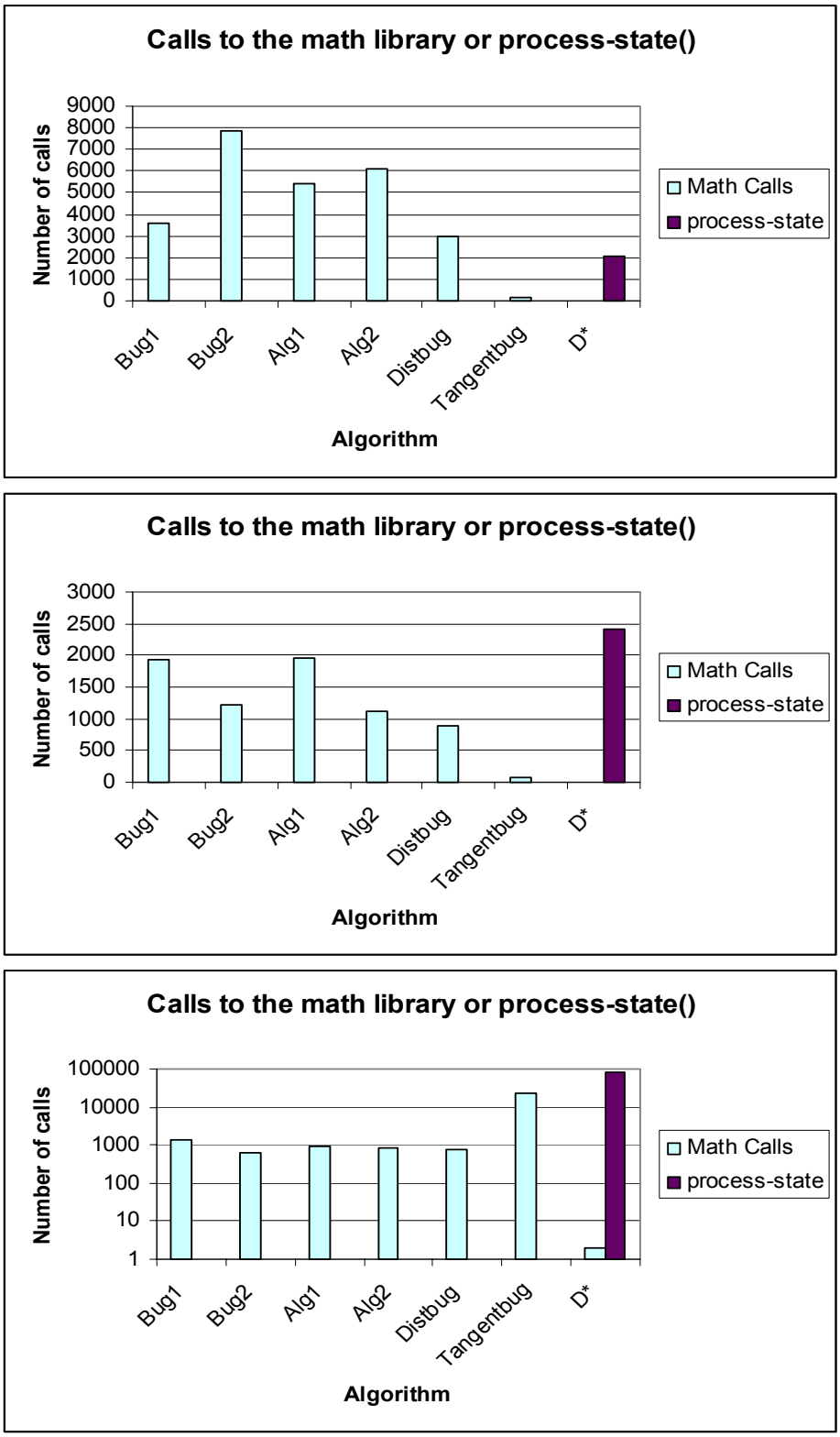


Figure 6-13(a) Top. Calls on Map1. (b) Centre. Calls on Map2. (c) Bottom. Calls on Map3.

6.5 Robustness

Robustness is essential for any practical robot application. No algorithm was designed with robustness in mind and this particular implementation also did not use robustness-enhancing techniques. Therefore, the results will identify algorithms which have inherent robustness and possible reasons for this are discussed.

Figure 6-14 shows that, in general, algorithms were not robust to noise. They performed better against PSD noise than driving related noise. Most algorithms performed poorly against linear driving noise and no algorithm, except for D*, had any robustness against rotational driving noise.

Dead-reckoning leading to false localisation is probably the major contributor to the poor robustness against linear and rotational driving noise. This cause of failure was indicated when algorithms converged too far away from the target. Using purely dead-reckoning, an algorithm cannot correct false localisation. In this sense, the algorithms are not responsible for poor robustness because it is beyond their control. The only way to correct false localisation is landmark recognition or sensor networks.

Robustness is also dependant on map. Comparison of figure 6-14(a) with figures 6-14(b) and 6-14(c) shows that robustness was worse on map1 than on map2 and map3. This implies that algorithms are more robust on open maps than enclosed maps. This is probably due to the greater accuracy needed in navigating tight enclosed spaces. In the open, there is much more room for error before failure occurs.

Also, the algorithms which relied on recognition of previously visited points performed very poorly. Figure 6-14 shows that Bug1 performed very poorly on all 3 maps because it needs to know when the robot has encircled the obstacle. Relying on previous points is difficult even without error because of the reasons discussed in section 4.9.1 and 4.9.2. Combined with driving error, finding previous hit/leave points is near impossible.

Tangentbug was another low achiever in robustness. Once again, this was due to driving errors leading to false localisation because it converged too far away from the goal. Tangentbug performed better against PSD error because the threshold criterion for node identification tolerates small amounts of PSD noise.

D* was the most robust because its segmentation of the grid requires the robot to move in small, discrete steps. Over large distances, the variance of these small steps tends to cancel out, resulting in a more accurate distance traveled. D* is also the most robust against rotational driving errors. This is most probably because of the limited degrees of freedom the robot possesses. Therefore, as long as the rotation is close to one of the eight directions, the algorithm can proceed.

If a robust algorithm is sought, D* is the only candidate. Fault tolerant techniques such as averaging multiple PSD readings or dividing long traveling distances into numerous small distances could be introduced in future.

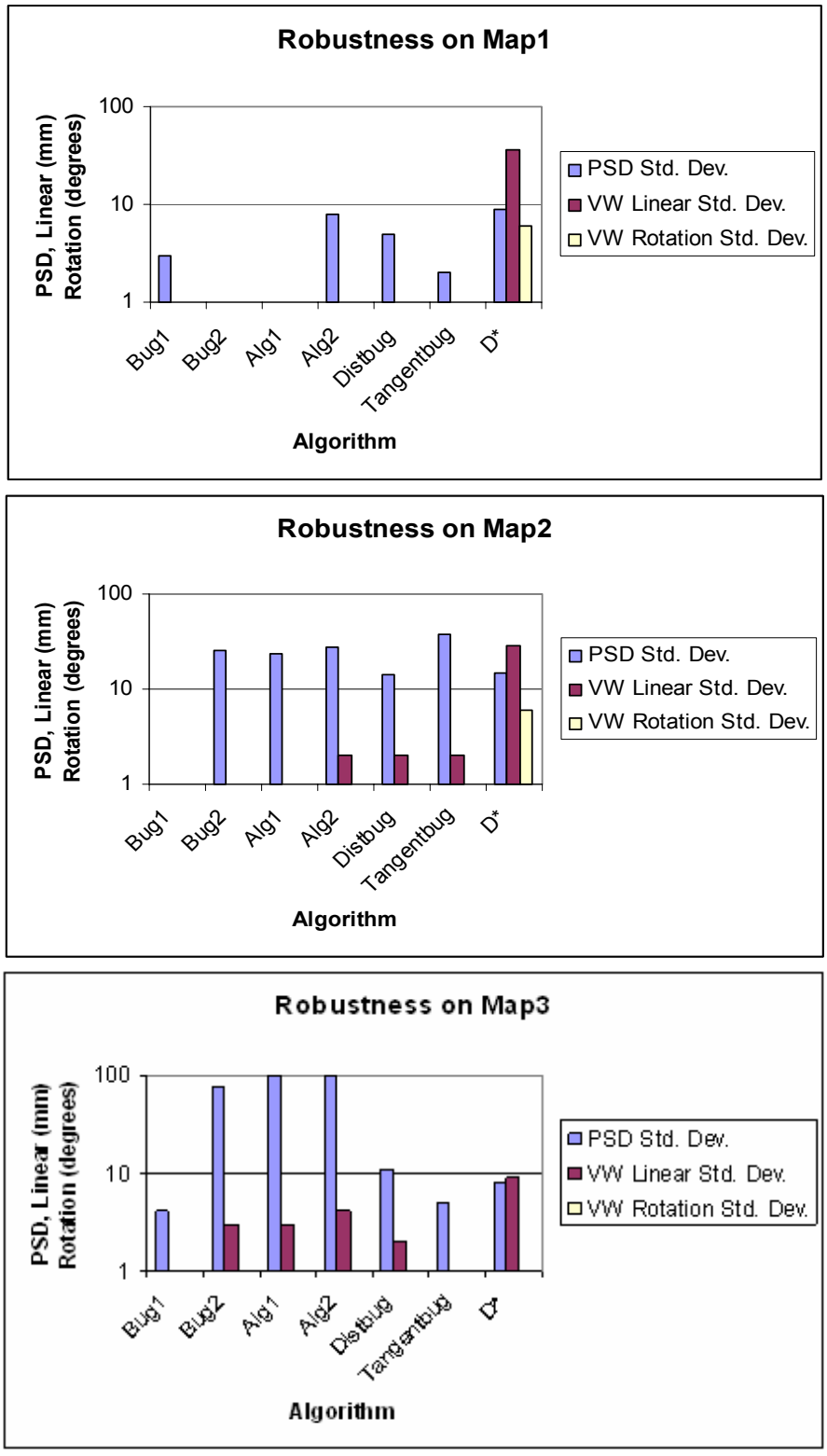


Figure 6-14(a) Top. Robustness on Map1. (b) Centre. Robustness on Map2. (c) Bottom. Robustness on Map3.

6.6 Complexity

From a software development perspective, the less complicated the algorithm the better and more desirable it is. Low complexity reduces software development costs and is less likely to fail.

Complexity is subjective and depends very much on the judgment of the implementer. The most objective measure is lines of code required for the implementation which is shown in figure 6-15. According to the results, the ranking of simplest to most complicated is: Bug2, Bug1, Distbug, Alg1, Alg2, Tangentbug and D*. This ranking is congruent with implementation experience.

Bug2 and Bug1 are the least complicated. Distbug is slightly more complicated due to its leaving condition requiring freespace(). Alg2 is very similar to Distbug but it requires a data-structure module, which makes it slightly more complicated. Alg1 is very similar to Alg2 and hence Alg1's complexity is very similar to Alg2. Tangentbug is moderately complicated due to the data collection and processing of the LTG. Finally, D* is the most complicated due to the classes needed to implement the grid.

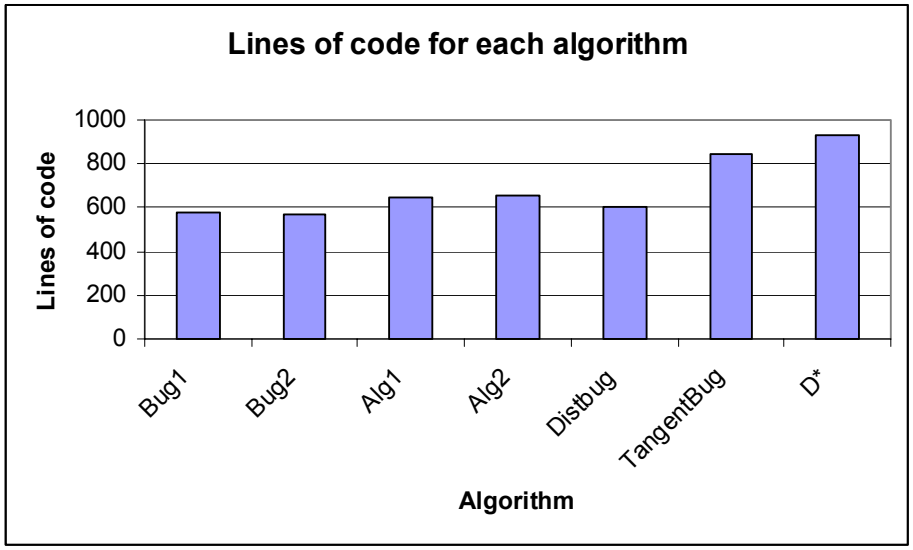


Figure 6-15. Lines of code for each algorithm

6.7 Memory Requirements

The memory requirements should be as low as possible. Figure 6-16 shows the ranking from most expensive to least expensive is: D*, tangentbug, alg2, alg1, distbug, bug2 and bug1. As expected, the simpler algorithms require less memory than the complicated ones.

Bug1, Bug2 and distbug were most inexpensive because they only require the common modules. Next, alg1 requires the data structure module. Alg2 requires an additional 4 bytes to track d(y,T). Tangentbug is next expensive because it needs to store $r(\theta)$, the nodes and the minimum module. Finally, D* requires the most amount of memory because it needs to store a grid full of cells.

It is important to note that only globally memory was measured. Some algorithms call functions which require a lot of memory for a short period of time. Therefore, global memory requirements should be viewed as the minimum memory requirement.

Compared to the amount of memory available on most robots, the memory requirements for the bug algorithms are quite small. The only algorithm which requires a substantial amount of memory is D*.

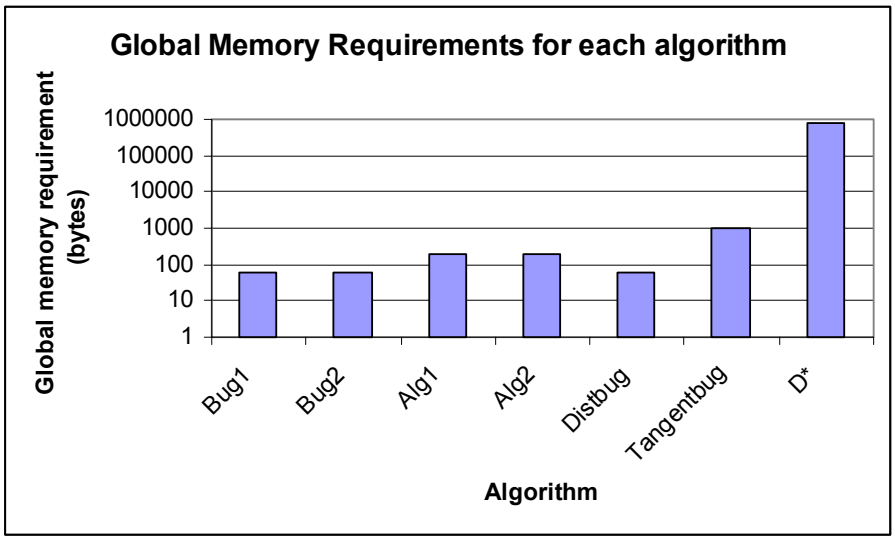


Figure 6-16. Global Memory Requirements for each algorithm

7. Conclusion

This thesis discusses the practical aspects of algorithms designed to solve the robotic navigation problem. Seven algorithms were selected for study: bug1, bug2, alg1, alg2, distbug, tangentbug and D*.

These algorithms were implemented using the Eyesim simulator. The implementation was modular and several common modules were identified and implemented separately. Once the modules were ready, the each navigation algorithm is implemented separately and simply calls the common modules to drive around. The exception to this is D* which is radically different to the others.

Several implementation issues arose. Firstly, there is the issue of navigation condition checking. Tangentbug and D* fare well because of their proactive nature. Secondly, there is the issue of radius around previous points. Tangentbug and D* proved the best for this because they have inbuilt mechanisms which avoid the problem. Thirdly, Tangentbug has a unique problem identifying discontinuities when it is near walls. Fourthly, D* has a problem if its cells are not able to cover the whole map. Finally, in terms of hardware costs, bug1 and bug2 are cheapest because they require only tactile sensors.

Three maps were created to test the implemented algorithms. Each map tested a different aspect of navigation. Map1 tested an algorithm's ability to navigate in an enclosed space. Map2 tested an algorithm's ability to navigate in an outdoor environment. Map3 tested an algorithm's ability to navigate out of a local minimum.

Ten trials were run for each algorithm on each map. Results were generated for path length, rotation, inherent rotation, computation time, computation time per metre, calls to math functions, robustness, simplicity and memory requirements. Table 7-1 shows the relative performance score of each algorithm for each attribute. The performance score was calculated by averaging the algorithm's rank over 3 maps. Therefore, a lower score means better performance.

	Bug1	Bug2	Alg1	Alg2	Distbug	Tangentbug	D*
Path Length	6.67	6.33	5	3.67	2.67	1.67	2
Rotation	6	3.67	4	6	4.67	2.33	1.33
Inherent Rotation	6.67	6.63	4	4.66	2	2	2.67
Total Computation time	4	3	5.33	5	4.33	3.33	3
Computation per metre	2	2.33	5	5.67	5.33	4.67	3
Calls to math functions	4.33	3.67	5.33	4.67	2	2.67	N/A
PSD Error	6	4	4	2	4.33	4	3.67
Linear Driving Error	6	4.83	4.83	3.33	4.33	3.33	1.33
Rotational Driving Error	6	5	2.83	4	5	2.83	2.33
Simplicity	2	1	4	5	3	6	7
Memory requirements	2	2	4	5	2	6	7

Table 7-1 Performance scores for each algorithm on the assessed attributes.

Table 7-1 clearly shows the dominant strengths various algorithms. For path length and rotation, tangentbug and D* are clearly the best. For computation time, bug1 and bug2 are the most inexpensive, but surprisingly, tangentbug and D* are very competitive. For robustness, D* is the only algorithm which inherently possesses this. For simplicity and memory requirements, bug1 and bug2 are clearly the best.

Using these findings, a selection policy can be devised to select the algorithm which best suits a user's needs. Figure 7-1 shows one possible selection policy. Note that this selection policy is aided by knowledge of the type of map. Strictly speaking, the type of map should not be known prior to execution. However, if this knowledge is available, it can assist decision making.

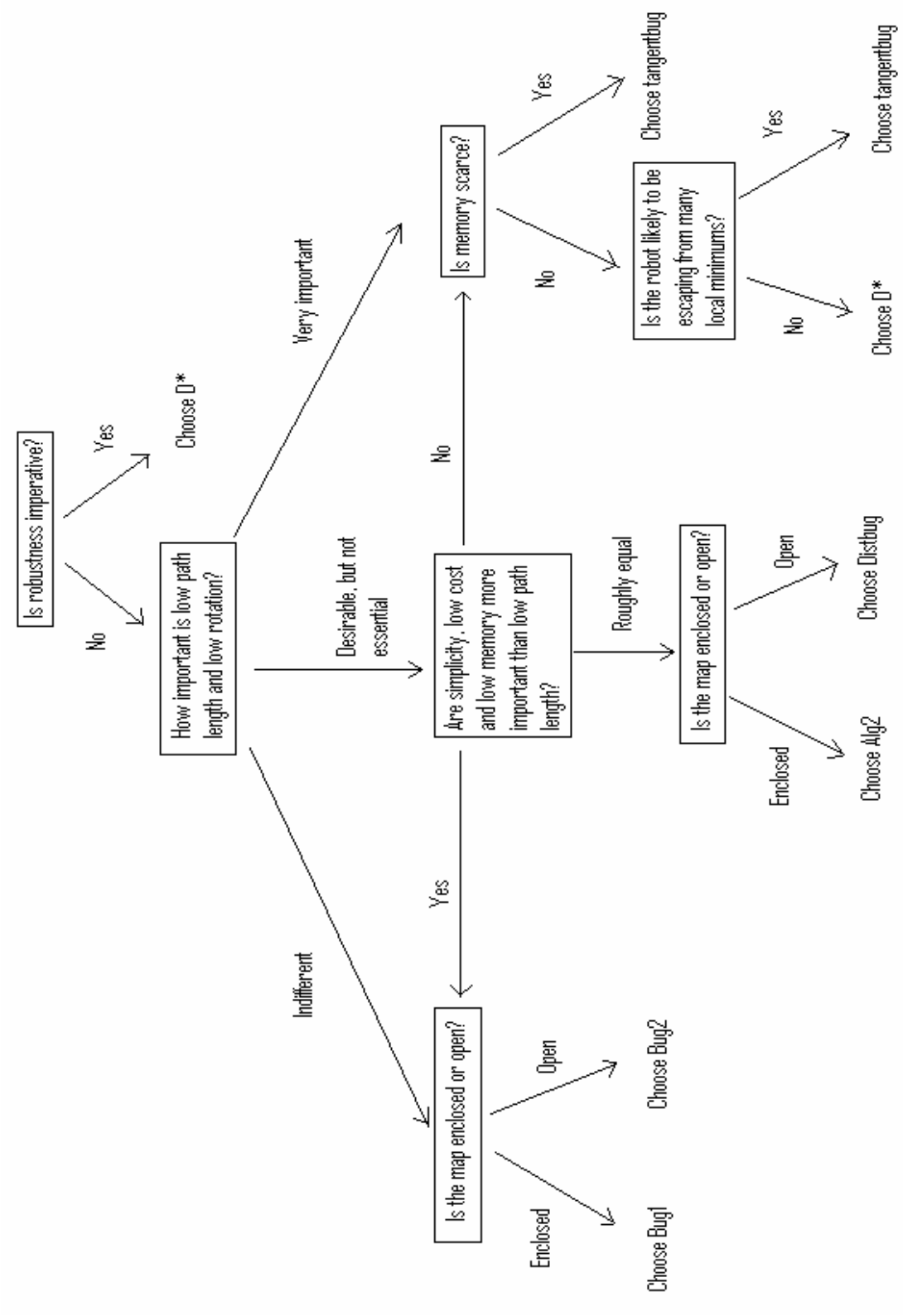


Figure 7-1 A possible selection policy based on table 7-1

8. Future improvements

8.1 Segmentation of the bug algorithms

Currently, the bug algorithms have unlimited degrees of freedom and no grid like structure. It would be interesting to investigate the bug algorithms on a grid like D*. This should give greater robustness at the expense of longer paths and higher memory requirements.

8.2 Localisation

Dead-reckoning causes false localisation leading to non-robustness against driving errors. Although this can be masked by multiple samplings, the best way to overcome error is feedback. One possible form of feedback is landmark recognition from camera images. If this is used, an image processing module needs to be written. Some techniques for this exist in the literature [11] and can be incorporated into future versions.

Another form of feedback could be to use sensor network interaction techniques for localisation [12, 13]. In these models, a sensor network is embedded in the environment and act as signposts for the robot. A recent improvement in radio technology has also made it possible to measure only the distance between beacons and the robot [14]. Line of sight is not required and the costs are significantly lower than conventional sensors.

The drawback of sensor networks that they deprive the robot of true autonomy. A sensor network must be present for navigation. The counter argument is that humans, which are generally regarded as autonomous, also rely on signposts upon entering an unfamiliar environment for the first time. Indeed, most public buildings display room numbers and signposts to the toilets, cafeteria etc… A good improvement could involve a robot which uses the sensor network. However, if there is no sensor network, it reverts to landmark recognition from its camera.

8.3 D improvements*

D* parameters can also be further investigated to yield better performance. The cell size is a very important factor which impacts on computation resources and completion times.

The cell structure can be altered and its effects investigated. Figure 8-1 shows a cell structure composed of triangles instead of squares. D* also needs find a solution to the problem discussed in section 4.3.5 concerning small grids not finding the optimal path. Dynamic cell allocation could be one possible solution

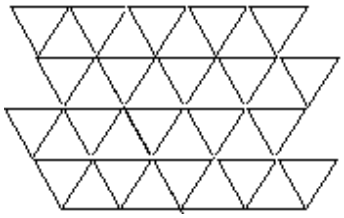


Figure 8-1 Instead of using squares, D* could use triangles

8.4 Robot Learning

Another area which can be improved on is robot learning. If the robot has traversed the trail before, it may be possible to retain some data and improve performance on subsequent traversals. For example, if the robot is using Alg1 or Alg2, it could retain the data-structure and improve subsequent journeys.

8.5 Fault tolerance

Another improvement could be to implement fault tolerance in navigation. This particular implementation was coded without fault-tolerance in mind. In future, fault-tolerant techniques could be incorporated to give a greater degree of assurance. For example, triple modular redundancy with a voter can be used on PSDs to average out error. A compass, gyroscope and GPS, if outdoors, can be used to reduce rotation error.

8.6 Tangentbug improvement

There could also be an improvement made to the tangentbug algorithm. Consider the initial computation performed on map2. Under the current tangentbug algorithm, the robot travels to node1 because node1 is closer to the target than node 2.

However, figure 5-2 shows that a shorter path is in the direction of node 2. The only hint that node2 produces a shorter path than node1 is the PSD reading shown in red in figure

9-2. If that reading can somehow be incorporated into tangentbug to make node2 the optimal node, it would allow tangentbug to more closely follow the ideal path in figure 5-2.

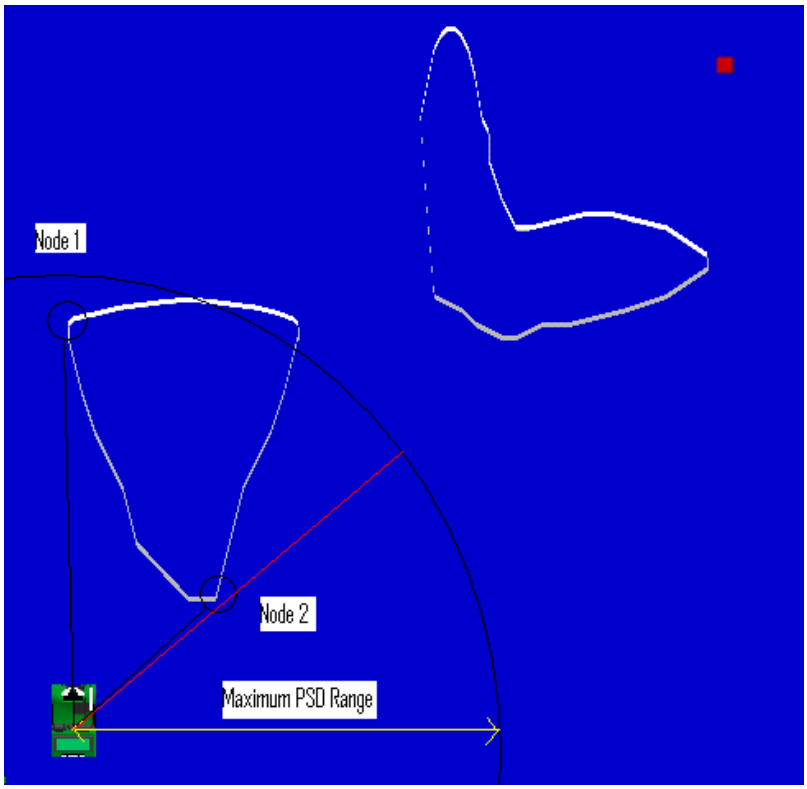


Figure 9-2 Tangentbug can be improved if it chooses node2 over node1

Tangentbug could also be improved to take into account varying terrain costs, similar to D*. This could be very difficult because the current tangentbug algorithm does not have any concept of cost inbuilt.

8.7 Map Classification

Some form of quantitative map classification scheme could also be designed. In this study, the terms enclosed, open and local minimum were loosely used to classify map1, map2 and map3 respectively. However, this was a subjective classification and had no objective basis. One possible technique is the concept of path safety introduced by Kamon and Rivlin [6] where determines its average distance from the nearest obstacle. One would expect that enclosed maps have smaller path safety than open maps.

9. Appendix

9.1 Bug1

Lines of code = 575 lines
Global Memory Requirements:
2 – PositionType = 6 float = 24 bytes
3 – float = 12 bytes
5 – integer = 20 bytes
1 – double = 4 bytes
Total = 60 bytes

9.1.1 Map1

	1	2	3	4	5	6	7	8	9	10
Computation	394	434	407	361	405	357	451	409	328	422
Drive	3763	3801	3859	3732	3909	3986	3799	3872	3828	3703

Average Computation time: 397 ms
Average driving time: 3825 ms
Total Distance: 30.56 m
Total Turning: 102.51 Radians
PSD standard deviation limit = 3mm
Standard deviation of linear VW-Control limit = 0 mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root = 1192, Pow = 2384, Geom = 2

9.1.2 Map2

	1	2	3	4	5	6	7	8	9	10
Computation	532	533	405	436	530	497	530	440	485	423
Drive	3172	3342	3486	3455	3361	3409	3361	3435	3406	3467

Average Computation time: 481 ms
Average Driving time: 3389 ms
Total Distance = 15.72 m
Total Turning = 54.834 Radians
PSD standard deviation limit = 0mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root = 645, Pow = 1290, Geom = 3

9.1.3 Map3

	1	2	3	4	5	6	7	8	9	10
Computation	142	77	47	142	48	106	109	95	63	94
Drive	1280	1126	1156	1061	1155	1082	1094	1092	1156	1125

Average Computation Time: 92 ms
Average Drive Time: 1133 ms
Total Distance: 12.5 m
Total Turning: 43.25 Radians

PSD standard deviation limit = 4mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root = 475, Pow = 950, Geom = 2

9.2 Bug2

Lines of code = 565 lines
Global Memory Requirements:
2 – PositionType = 6 float = 24 bytes
3 – float = 12 bytes
5 – integer = 20 bytes
1 – double = 4 bytes
Total Requirement = 60 bytes

9.2.1 Map1

	1	2	3	4	5	6	7	8	9	10
Computation	641	405	578	673	512	566	658	562	716	468
Drive	5875	6127	5969	5859	6035	5980	5889	5813	5753	6063

Average Computation time: 578 ms
Average Drive time: 5936 ms
Total Distance: 50.3 m
Total Turning: 155.41 Radians
PSD standard deviation limit = 1mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root = 1939, Pow = 5908, Geom = 6

9.2.2 Map2

	1	2	3	4	5	6	7	8	9	10
Computation	282	249	341	250	380	344	346	296	217	408
Drive	1843	1891	1721	1891	1745	1781	1811	1844	1783	1701

Average Computation time= 311 ms
Average driving time= 1801 ms
Total Distance = 7.46 m
Total Turning = 26.12 Radians
PSD standard deviation limit = 26mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 318, Pow = 896, Geom = 3

9.2.3 Map3

	1	2	3	4	5	6	7	8	9	10
Computation	78	32	31	78	16	16	78	47	108	15
Drive	391	452	454	406	468	468	391	438	376	469

Average Computation time = 50 ms
Average driving time = 431 ms

Total Distance = 4.5 m
Total Turning = 15.36 Radians
PSD standard deviation limit = 75mm
Standard deviation of VW-Control limit = 3mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root = 156, Pow = 484, Geom = 2

9.3 Alg1

Lines of code = 647 lines
Square-root =1514, Pow =3888, Geom = 13
Global Memory Requirements:
2 – PositionType (pos, targ) = 6 float = 24 bytes
3 – float = 12 bytes
6 – integer = 24 bytes
1 – double = 4 bytes
10 – PositionType (previous hit and leave points) = 30 float = 120 bytes
Total Requirement = 184 bytes

9.3.1 Map1

	1	2	3	4	5	6	7	8	9	10
Computation	813	767	814	750	690	746	656	959	752	1023
Drive	3375	3405	3342	3438	3497	3441	3531	3229	3420	3133

Average Computation time: 797 ms
Average Driving time: 3381 ms
Total Distance = 23.6 m
Total Turning = 88.66 Radians
Total turning without freespace = 85.551216 Radians
PSD standard deviation limit = 0mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root =1514, Pow =3888, Geom = 13

9.3.2 Map2

	1	2	3	4	5	6	7	8	9	10
Computation	864	766	624	761	721	987	863	728	672	702
Drive	1854	1921	2079	1942	1982	1716	1840	1991	2015	2001

Average Computation time: 769 ms
Average driving time: 1934 ms
Total Distance = 7.3 m
Total Turn = 28.85 Radians
Total turn without freespace = 25.478146 Radians
PSD standard deviation limit = 24mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 565, Pow = 1398, Geom = 5

9.3.3 Map3

	1	2	3	4	5	6	7	8	9	10
Computation	78	63	93	47	62	123	109	78	141	107
Drive	500	515	470	500	516	455	469	484	437	471

Average Computation time = 90 ms
Average driving time = 482 ms
Total Distance = 4.44 m
Total Turning = 15.75 Radians
Total turning without freespace = 11.677344 Radians
PSD standard deviation limit = 100mm (maximum)
Standard deviation of linear VW-Control limit = 3mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 236, Pow = 642, Geom = 3

9.4 Alg2

Lines of code = 653 lines
Global Memory Requirements:
2 – PositionType (pos, targ) = 6 float = 24 bytes
4 – float = 16 bytes
6 – integer = 24 bytes
1 – double = 4 bytes
10 – PositionType (previous hit and leave points) = 30 float = 120 bytes
Total Requirement = 188 bytes

9.4.1 Map1

	1	2	3	4	5	6	7	8	9	10
Computation	676	815	670	887	623	792	625	705	659	699
Drive	4746	4467	4658	4410	4814	4661	4672	4780	4826	4785

Average Computation time: 715 ms
Average driving time: 4682 ms
Total distance = 15.54 m
Total turn = 157.2 Radians
Total turn with freespace not recorded = 50.947201 Radians
PSD standard deviation limit = 8mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 1mm
Square-root = 1943, Pow = 3886, Geom = 291

9.4.2 Map2

	1	2	3	4	5	6	7	8	9	10
Computation	410	484	450	466	544	592	419	529	357	514
Drive	1668	1578	1644	1877	1550	1485	1581	1580	1737	1564

Average Computation time: 477 ms
Average driving time: 1626 ms
Total distance = 5.09 m
Total turn = 26.94 Radians

Total turn without freespace = 13.644241 Radians
PSD standard deviation limit = 28mm
Standard deviation of linear VW-Control limit = 2mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 358, Pow = 716, Geom = 52

9.4.3 Map3

	1	2	3	4	5	6	7	8	9	10
Computation	174	62	110	157	78	124	111	48	157	126
Drive	904	985	968	936	1001	954	982	1014	890	936

Average Computation time = 115 ms
Average driving time = 957 ms
Total distance = 4.22 m
Total turn = 37.46 radians
Total turn without freespace = 13.345954 radians
PSD standard deviation limit = 95mm
Standard deviation of VW-Control limit = 4mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root = 269, Pow = 538, Geom = 73

9.5 Distbug

Lines of code = 604 lines
Global Memory Requirements:
2 – PositionType (pos, targ) = 6 float = 24 bytes
3 – float = 12 bytes
5 – integer = 20 bytes
1 – double = 4 bytes
Total Requirement = 60 bytes

9.5.1 Map1

	1	2	3	4	5	6	7	8	9	10
Computation	545	528	593	551	566	533	686	466	519	721
Drive	4252	4253	4188	4246	4199	4249	4110	4331	4263	4075

Average Computation time: 571 ms
Average Driving time: 4212 ms
Total distance = 15.26 m
Total turn = 140.28 Radians
Total turn without freespace = 47.344215 Radians
PSD standard deviation limit = 5mm
Standard deviation of linear VW-Control limit = 0mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root = 908, Pow = 1816, Geom = 278

9.5.2 Map2

	1	2	3	4	5	6	7	8	9	10
Computation	281	312	373	330	468	421	378	313	359	453

Drive	1953	1656	1596	1639	1501	1564	1670	1656	1610	1531

Average Computation time: 369 ms
Average Driving time: 1638 ms
Total distance = 4.91 m
Total turn = 26.74 Radians
Total turn without freespace = 11.167029 Radians
PSD standard deviation limit = 14mm
Standard deviation of linear VW-Control limit = 2mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 278, Pow = 556, Geom = 53

9.5.3 Map3

	1	2	3	4	5	6	7	8	9	10
Computation	190	125	110	63	107	93	109	95	124	170
Drive	920	984	1015	1062	1002	1048	1001	1030	955	908

Average Computation time = 119 ms
Average driving time = 993 ms
Total Distance = 4.2 m
Total turning = 36.48 Radians
Total turn without freespace = 11.645906 Radians
PSD standard deviation limit = 11mm
Standard deviation of linear VW-Control limit = 2mm
Standard deviation of rotational VW-Control limit = 0 degrees
Square-root =231, Pow = 462, Geom = 66

9.6 Tangentbug

Lines of Code = 843 lines
Global Memory Requirements:
2 – Position Type = 6 float = 24 bytes
4 – float = 16 bytes
8 – int = 64 bytes
3 degrees between samples – 120 samples = 120 ints = 480 bytes
20 nodes – 40 ints and 60 floats = 400 bytes
Total memory requirement: 984 bytes

9.6.1 Map1

	1	2	3	4	5	6	7	8	9	10
Computation	139	94	93	109	157	125	155	156	126	171
Drive	721	750	751	750	703	734	704	688	687	689

Average Computation time: 133 ms
Average driving time: 718 ms
Total distance = 4.75 m
Total turn = 22.50 Radians
Total turn without freespace and $r(\theta)$ = 11.618871 Radians
PSD standard deviation limit = 2mm
Standard deviation of linear VW-Control limit = 1mm

Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 46, Pow =52, Geom = 35

9.6.2 Map2

	1	2	3	4	5	6	7	8	9	10
Computation	392	345	201	296	251	231	297	233	391	391
Drive	764	811	939	844	905	925	859	908	766	766

Average Computation time: 303 ms
Average driving time: 849 ms
Total Distance = 4.63 m
Total turn = 10.37 Radians
Total turn without freespace and $r(\theta)$ = 4.310099 Radians
PSD standard deviation limit = 38mm
Standard deviation of linear VW-Control limit = 2mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 26, Pow = 28, Geom = 21

9.6.3 Map3

	1	2	3	4	5	6	7	8	9	10
Computation	140	109	140	109	171	94	93	142	126	141
Drive	657	688	688	719	658	718	719	686	687	687

Average Computation time = 127 ms
Average driving time = 691 ms
Total Distance = 3.33 m
Total turn = 24.94 Radians
Total turn without freespace and $r(\theta)$ = 12.776836 Radians
PSD standard deviation limit = 5mm
Standard deviation of linear VW-Control limit = 1mm
Standard deviation of rotational VW-Control limit = 1 degree
Square-root = 4736, Pow = 9442, Geom = 8902

*9.7 D**

Lines of Code = 931 lines
ArcEnd structure = 4 + 1 = 5 bytes
Cell structure = 4x4 + 1 + 1 + 5x8 = 58 bytes
Grid class = 36 + 81x81x58 = 380,574 bytes
Open list class = 4x100000 = 400,000 bytes
Driving = 3x4 + 2x4 = 20 bytes
Discrepancy class = 8 bytes
User Interface = 16 bytes
Timer = 16 bytes
Total memory = 780692 bytes

9.7.1 Map1

	1	2	3	4	5	6	7	8	9	10
Computation	61	78	31	31	31	47	62	31	47	46

Drive	564	563	594	609	609	594	547	594	593	594

Average Computation time = 47 ms
Average driving time = 586 ms
Total Distance = 4.33 m
Total Turn = 17.28 Radians
PSD standard deviation limit = 9mm
Standard deviation of linear VW-Control limit = 36mm
Standard deviation of rotational VW-Control limit = 6 degrees
Number of calls to process state: 2034

9.7.2 Map2

	1	2	3	4	5	6	7	8	9	10
Computation	79	126	92	142	79	32	141	94	79	79
Drive	843	843	924	874	921	1000	859	922	937	937

Average Computation time = 94 ms
Average driving time = 906 ms
Total Distance = 4.4 m
Total Turn = 12.57 Radians
PSD standard deviation limit = 15mm
Standard deviation of VW-Control limit = 29mm
Standard deviation of rotational VW-Control limit = 6 degrees
Number of calls to process state: 2424

9.7.3 Map3

	1	2	3	4	5	6	7	8	9	10
Computation	9673	9626	9687	9627	9671	9641	9640	9625	9643	9658
Drive	452	468	391	436	438	453	407	484	436	405

Average Computation time = 9650 ms
Average drive time = 437 ms
Total Distance = 4.24 m
Total Turn = 12.57 Radians
PSD standard deviation limit = 8mm
Standard deviation of VW-Control limit = 9mm
Standard deviation of rotational VW-Control limit = 0 degrees
Number of calls to process state: 83855

10. References

[1] M. Brain, "Robotic Nation", http://marshallbrain.com/robotic-nation.htm

[2] Robocup official site, http://www.robocup.org/

[3] V. Lumelsky and P. Stepanov. "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment". *IEEE Transactions on Automatic Control. Volume 31 No.11*. Pages 1058-1063. Nov 1986

[4] A. Sankaranarayanan and M. Vidyasagar, "A New Path Planning Algorithm For Moving A Point Object Amidst Unknown Obstacles In A Plane", *IEEE Conference on Robotics and Automation*, pages 1930-1936, 1990

[5] A. Sankaranarayanan and M. Vidyasagar, "Path Planning For Moving A Point Object Amidst Unknown Obstacles In A Plane: A New Algorithm And A General Theory For Algorithm Development", *Proc. of the 29th Conference on Decision and Control,* pages 1111-1119, 1991

[6] I. Kamon and E. Rivlin. "Sensory-Based Motion Planning with Global Proofs". *IEEE Transactions on robotics*. *Volume 13(6)*, pages 814-821. 1997

[7] I. Kamon, E. Rivlin and E. Rimon, "A new Range-Sensor Based Globally Convergent Navigation Algorithm for Mobile Robots", *Techion Department of Computer Science*, 1995

[8] S. Laubach and J. Burdick, "An autonomous Sensor-Based Path-Planner for Planetary Microrovers", *Proc. of the 1999 IEEE Conference on Robotics and Automation*, pages 347-354, May 1999.

[9] J. Latombe. "Robot Motion Planning", *Kluwer Academic Publishers*, 1991

[10] A. Stentz, "Optimal and Efficient Path Planning for Partially-Known Environments", *Proc. of IEEE Conference on Robotic Automation*, pages 3311-3317,1994.

[11] J.B. Hayet, F. Lerasle and M. Devy, "A visual landmark framework for indoor mobile robot navigation", *Proc. of the 2002 IEEE International Conference on Robotics and Automation*, pages 3942-3947, 2002

[12] M. Batalin, G. Sukhatme and M. Hattig, "Mobile Robot Navigation using a sensor network", *Proc. of the 2004 IEEE Conference on Robotics and Automation*, 2004

[13] R. Peterson and D. Rus, "Interacting with Sensor Networks", *Proc. of the 2004 IEEE Conference on Robotics and Automation*, 2004

[14] G. Kantor and S. Singh, "Preliminary Results in Range-only Localisation and Mapping", *Proc. of the 2002 IEEE Conference on Robotics and Automation*, 2002