An Analysis of Mobile Robot Navigation Algorithms in Unknown Environments

James Ng

This thesis is presented for the degree of

Doctor of Philosophy of Engineering

School of Electrical, Electronic and Computer Engineering

February 2010

Abstract

This thesis investigates robot navigation algorithms in unknown 2 dimensional environments with the aim of improving performance. The algorithms which perform such navigation are called Bug Algorithms [1,30,62]. Existing algorithms are implemented on a robot simulation system called EyeSim [7] and their performances are measured and analyzed.

Similarities and differences in the Bug Family are explored particularly in relation to the methods used to guarantee termination. Seven methods used to guarantee termination in the existing literature are noted and form the basis of the new Bug algorithms: OneBug, MultiBug, LeaveBug, Bug1+ and SensorBug. A new method is created which restricts the leave points to vertices of convex obstacles.

SensorBug is a new algorithm designed to use range sensors and with three performance criteria in mind: data gathering frequency, amount of scanning and path length. SensorBug reduces the frequency at which data about the visible environment is gathered and the amount of scanning for each time data is gathered. It is shown that despite the reductions, correct termination is still guaranteed for any environment.

Curv1 [19], a robot navigation algorithm, was developed to guide a robot to the target in an unknown environment with a single non-self intersecting guide track. Via an intermediate algorithm Curv2, Curv1 is expanded into a new algorithm, Curv3. Curv3 is capable of pairing multiple start and targets and coping with self-intersecting track.

Acknowledgments

I would like to express my gratitude and thanks to Professor Gary Bundell for his generous support during his time as Head of School. After your term expired, there were times when my resolve to finish wavered but I owe it to you to present what was accomplished during your time.

I would also like to thank Professor Anthony Zaknich and Keith Godfrey for being mentors to me. Your mentoring was very valuable and I would like to take this opportunity to express my gratitude for it.

I also acknowledge Chang-Su Lee, my colleague, for the long and wonderful conversations about our PhD studies, effort in teaching projects and certain supervisors.

I would also like to thank my family for their support.

I would like to acknowledge all the undergraduate students who I have had the pleasure of getting to know during my time as a tutor. I loved the job and I knew that you enjoyed having me as your tutor. Sadly, it could not continue. I wish you all the best.

I would like to acknowledge all the high school students who have put up with my maths tuition and stories of life at university, education, PhD graduations and money. I hope that I have educated you in more ways than one and that you understand what education is all about. Many thanks also to the parents.

Finally, thanks also to Professor Brett Nener and the GRS for their PhD student retention strategies. I understand why it is so successful now.

Table of Contents

Chapter 1

Introduction and Overview

1.1 Background and Motivation for the Bug Algorithms

A 2-dimensional robot driving environment contains a starting point and a target point. A finite number of arbitrarily shaped obstacles, each of finite area, are then placed in the environment. The robot starts at the start point and its objective is to find an obstacle-free, continuous path from start to the target. Figure 1-1 shows sample environments with the green tile marking the start and the red tile marking the target.

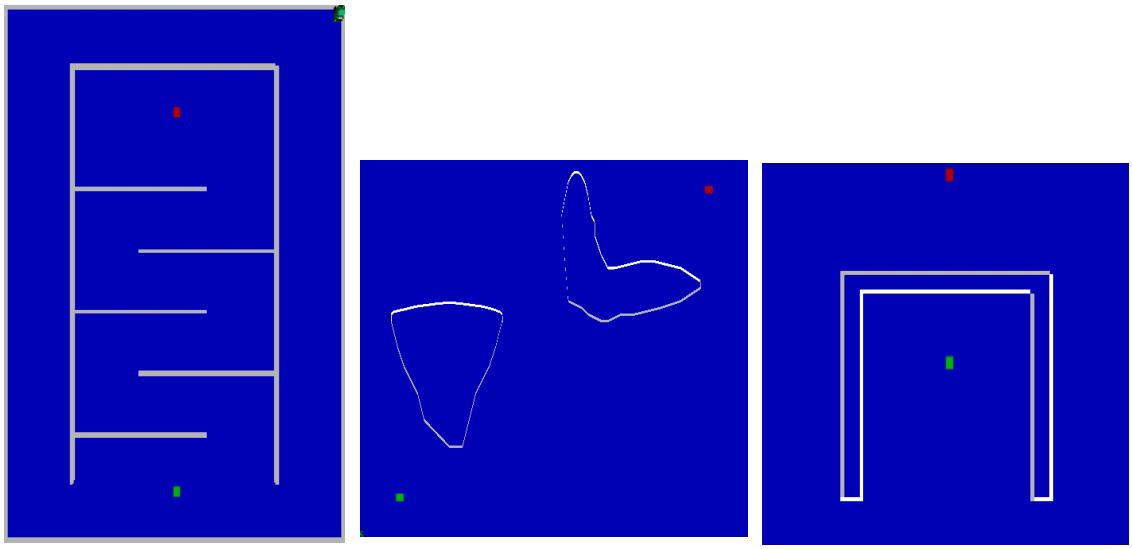

Figure 1-1 Sample navigation environments

The aim of the Bug algorithms is to guide a robot starting at S to the target T given that the robot has no knowledge of the environment. The robot should achieve this goal with as little global information as possible. In practical terms, this means the robot can remember several points of

interest but it cannot, say, perform mapping. If no such path exists, the algorithm is to terminate and report that the target is unreachable. This objective is called termination [1].

The Bug algorithms can be programmed into any robot with tactile or range sensors and a localization method such as odometers, landmark recognition or GPS. Then, the robot is able to autonomously find a path to a desired target. Lumelsky [76,78,79] also has applied the research to robot arms which are attempting to reach a desired pose. In these situations, the movement of the robot arm is very similar to a mobile robot navigating in an unknown environment except that the robot arm is tied to a fixed base.

Another application is close range inspection [52]. This occurs when the robot is surveying a particular area for an item of interest. When it finds such an item it usually needs to get closer to the object to get more details. For example, a robot might be deployed to find radioactive objects in a nuclear reactor. If the robot, from afar, detects a suspicious object, it needs to get closer to determine if that object really is leaking radiation. Thus, it will require a navigation strategy to get close to the suspicious object in an environment where there may be many objects.

Given that the environment may continually change and little information about the environment may be known at any given time, the navigation strategy must reach the target with as little information as possible, preferably only the current position and the target. Some well-known path planning techniques such as A* [39,40,50,53], Dijkstra [54], distance transformation [18,35,55], potential fields [7,14,44,72,73], sampling based [56, 57] and the Piano Movers' problem [59,60,61] require additional information or even a complete map. Others are designed for coverage

path planning [95] which has applications in lawn mowing [96], harvesting [97] and mine hunting [98]. These shortcomings demonstrate a need for point-to-point navigation in unknown environments.

Laubach and Burdick [10] planned to implement WedgeBug on a sojourner rover that is to be sent to Mars if tests are successful. They note that for a motion planner to be useful on Mars, it needs the following characteristics: assume no prior knowledge of the environment, must be sensor-based, robust, complete and correct. WedgeBug satisfies most of the requirements except for a few reported errors on the robustness due to localization errors.

Kim, Russell and Koo designed SensBug for earthwork operations in the construction industry [71]. They note the need for enhanced intelligence for robots in hazardous work environments such as underwater, in chemically or radioactively contaminated areas and in regions with harsh temperatures.

Langer, Coelho and Oliveira [87] note that there is an increasing need for path planning algorithms in unknown environments for manufacturing, transport, goods storage, medicine (remote controlled surgery), military applications, computer games and spatial exploration [88,89,90, 91,92]. They simulated K-Bug in an office like environment and showed that the robot produced competitive paths compared with A*.

Pioneering work on this problem was done by Lumelsky [1,13,16,30,58,62,63,70,74,75,76,77,78,79,80,81,82]. Prior to Lumelsky's work, robot navigation in unknown environments consisted of maze searching algorithms such as the pledge algorithm [21] and Tarry's algorithm [83]. Unfortunately, in the case of the pledge algorithm, a robot

cannot travel to a particular point and the path length performance of both algorithms can be arbitrarily large. There also existed heuristical [84,85,86] methods but these required knowledge of the robot's environment in a limited area around it.

To the best of the author's knowledge, the Bug algorithms were the first non-heuristic algorithms for motion planning in an unknown environment which guaranteed termination. Further, the robot does not need to build a map of the environment, it only needs to store one point for termination to be guaranteed. This makes the Bug algorithms highly suitable for real-time implementation.

Lumelsky and Skewis later extended this work to include range sensors [16]. With range sensors, the robot is able to detect points which are further along the Bug2 path. When the robot can do this, it takes shortcuts and this reduces path length. Later, Kamon designed DistBug [5] which assisted the robot in making better leaving decisions and then TangentBug [6] in which the robot uses the range sensor to gain an omni-directional view of its immediate surroundings.

1.2 Aims of this Thesis

This thesis aims to improve the performance of mobile robots in unknown environments. Several algorithms are simulated and investigated using the EyeSim [7] simulation system. Inferences about performance factors are made and used to improve algorithm performance. Algorithm performance is also investigated when a guide track is available and used to create a new algorithm, Curv3, which is able to perform in environments where there is self-intersecting track, moving obstacles and multiple trails.

1.3 Assumptions of the Bug Model

The Bug model makes three simplifying assumptions about the robot [1, 30]. First, the robot is a point object. This means that the robot has no size and can fit between any arbitrarily small gap. This assumption overcomes the problem that a gap may exist on the map but the robot may be too large.

Second, the robot has perfect localization ability. This means that the robot knows its true position and orientation relative to the origin at any time. This assumption allows the robot to determine the precise distance and bearing to the target and this is very important for guaranteeing termination and for arriving at the target if the target is reachable.

Third, the robot has perfect sensors. In certain algorithms, the robot requires distance sensors to assist navigation. These algorithms rely on the sensor data significantly and imperfect sensors may adversely affect performance.

Obviously, these three assumptions are unrealistic for real robots, and therefore Bug algorithms cannot be directly applied for navigation tasks of real robots, but could be considered as a higher-level supervisory component of a system that incorporates all three assumptions.

In the Bug algorithm publications, some show only theoretical results [1,3,4,13,16,17,19,30,36,62,70,71] and some show theoretical and simulation results [2,8,9,11,12,28,32,34,37,38,64,87]. Several attempts were made at implementing the Bug algorithms on real robots [5,6,41] but frequent problems occurred and the algorithm results and comparisons were based on simulations.

Laubach [10] implemented a modified version of TangentBug [6] on a sojourner rover. In future, it is hoped to be sent to Mars so presumably the algorithm must have worked quite well. However, most of the paper is devoted to theoretical proofs of convergence and other interesting properties of WedgeBug and RoverBug but no practical implementation advice is offered.

Kreichbaum [12] designed Optim-Bug to work with the ideal Bug assumptions and then attempted to account for error in UncertainBug. Dead-reckoning error was compensated by using artificial landmarks and UncertainBug purposely deviates from the Optim-Bug path to use these landmarks for error compensation. Error was introduced in the simulation model and experiments were performed to measure performance. Error compensation was satisfactory but the main drawback was that UncertainBug is unable to guarantee a path to target if such a path exists.

Lumelsky [13] designed Angulus to specifically exclude the reliance on dead-reckoning. Instead, the robot relies on compass readings to determine when to leave the obstacle. However, path length is compromised and may be much higher than a Bug algorithm. Further, there may still be error in a real compass reading when put on a real robot.

Kim, Russell and Koo [71] suggested using the Global Positioning System (GPS) to localize the robot. Although they did not implement the GPS on a real robot themselves, it was noted that GPS is widely used and able to accurately localize objects which are outdoors. Obviously, if used indoors, this approach will not be as successful.

Skewis and Lumelsky [63] implemented Bug2 and VisBug on a

LABMATE robot within a laboratory measuring 8 meters by 6 meters. The robot had the following functionality: mobility, dead-reckoning, obstacle range sensing, landmark registration and motion planning strategies. It was found that the robot's ability to navigate successfully was quite remarkable and included tests on path repeatability, handling local cycles, tests for target reachability and task sequencing.

The results were encouraging but they did find that dead-reckoning alone was not enough to provide sufficient accuracy and they needed to use landmarks to compensate for dead-reckoning error. Landmarks have been classified as feature-based or cell-based. Feature-based landmark recognition [65,66,67] uses natural features of the terrain such as obstacle vertices to localize the robot. The cell-based approach creates a 2D array occupancy cells to estimate a robot's position [68,69].

The purpose of the experiment was not to replicate the landmark recognition techniques but rather to use their outcomes. As such, artificial landmarks were introduced into the environment and the robot was given information about them relative to the starting position. These landmarks were distributed throughout the environment, both on obstacles and on the roof. Once the robot detected a landmark with its IR sensor, it recalibrated its position based on the information given beforehand. Obviously, in a natural setting with no artificial landmarks, such experimental success may be difficult to replicate but this experiment shows that the Bug algorithms are capable of fulfilling its purpose in practice if error is overcome.

Given the Bug algorithm history, it is the norm that algorithms are developed theoretically and then sometimes implemented on real robots. In this thesis, new Bug algorithms are developed and simulated in ideal

environments. The few experiments which have been run on real robots have produced large localization errors which are beyond the scope of this study to rectify. It is left for future research to compensate for this error using existing techniques such as probabilistic localization [20], Kalman Filters [23] and SLAM [24,25].

1.4 Bug Notation

The following notation is used in the Bug algorithms:

- H_i – the i[th] hit point. This is the i[th] time the robot transitions from "moving to target" mode to "boundary following" mode.
- L_i – the i[th] leave point. This is the i[th] time the robot transitions from "boundary following" mode to "moving to target" mode.
- 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 Position Sensitive Device (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.

1.5 The Bug Algorithms

The following section summarizes the existing Bug algorithms.

1.5.1 Bug1

The Bug1 algorithm was the first algorithm in the bug family [1,30,62] created by Lumelsky and Stepanov. Bug1 operates as shown in Figure 1-2 and an example is illustrated in Figure 1-3:

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 $d_{path}(H_i, x)$ of point(s) where $d(x,T)$ is minimal and whether the robot can drive towards the target at x. Label one of these minimal points L_i. When the robot revisits H_i, test whether the target is reachable by checking if the robot can move towards the target at L_i. If the robot cannot then terminate and conclude that the target is unreachable. If the robot can, choose the wall-following direction which minimizes $d_{path}(H_i, L_i)$ and maneuver to L_i. At L_i, proceed to step 1.

Figure 1-2. The Bug1 algorithm

Put simply, the Bug1 algorithm searches each encountered obstacle for the point which is closest to the target. Once that point is determined, the robot evaluates whether it can drive towards the target or not. If it cannot, the

target is unreachable. If it can, the robot knows that by leaving at that point, it will never re-encounter the obstacle.

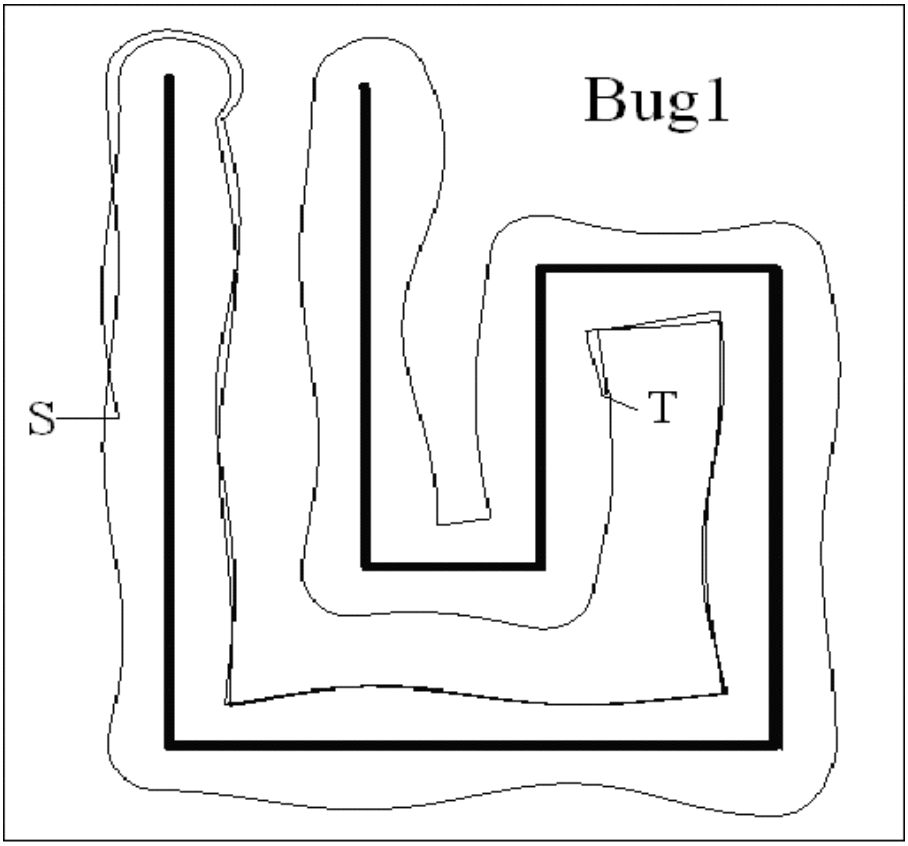

Figure 1-3 The Bug1 algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.5.2 Bug2

The Bug2 algorithm was also created by Lumelsky and Stepanov [1,30,62]. It is less conservative than Bug1 because the robot can leave earlier due to the M-line. Bug2 operates shown in Figure 1-4 with an example illustrated in Figure 1-5:

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 encountered. Label this point H_i. Go to step 2

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

- The target is reached. Stop.
- A point along M is found such that $d(x, T) < d(H_i, T)$. If the robot is able to move towards the target. Label this point L_i. Go to step 1. Otherwise, update $d(H_i, T)$ with $d(x,t)$.
- The robot returns to H_i. The target is unreachable. Stop.

Figure 1-4 The Bug2 Algorithm

There has been some clarification in the literature [3, 8] about the leaving conditions for Bug2. Recently, Antich and Ortiz suggested Bug2+ [36] and this algorithm clarified all doubt, but Sankar [3] and Noborio [8] had already built these clarifications into their respective algorithms which are similar to Bug2. In this thesis, the name Bug2 is used but when simulated or drawn, the Bug2+ algorithm (Figure 1-4) shall be used. This is because the author believes that Lumelsky had originally intended these features to be part of Bug2 but did not explicitly state them. This is justified below.

Lumelsky's original leaving condition states "*b) M-line is met at a distance d from T such that d < d(H, T). Define the leave point Lj. Set j = j + 1. Go to Step 1."* A strict interpretation of this directive allows Bug2 to define a leave point even though the robot will, upon executing step 1, define a hit point again without moving. However, it does not make sense that a robot is allowed to leave if it does not move towards the target immediately after leaving. Hence, the robot is only allowed to leave if it can drive towards the target.

Also, if the robot is denied leaving because it cannot move toward the target, then it should update $d(H_i, T)$ with $d(x, T)$. Obviously, if a robot denied leaving because it cannot move toward the target then there must exist a point on the same obstacle and on the M-line which is closer to the target. In any case, if Lumelsky's original algorithm was strictly followed the actual path is the same as in Bug2+ since the robot will update $d(H_i, T)$ when executing step1.

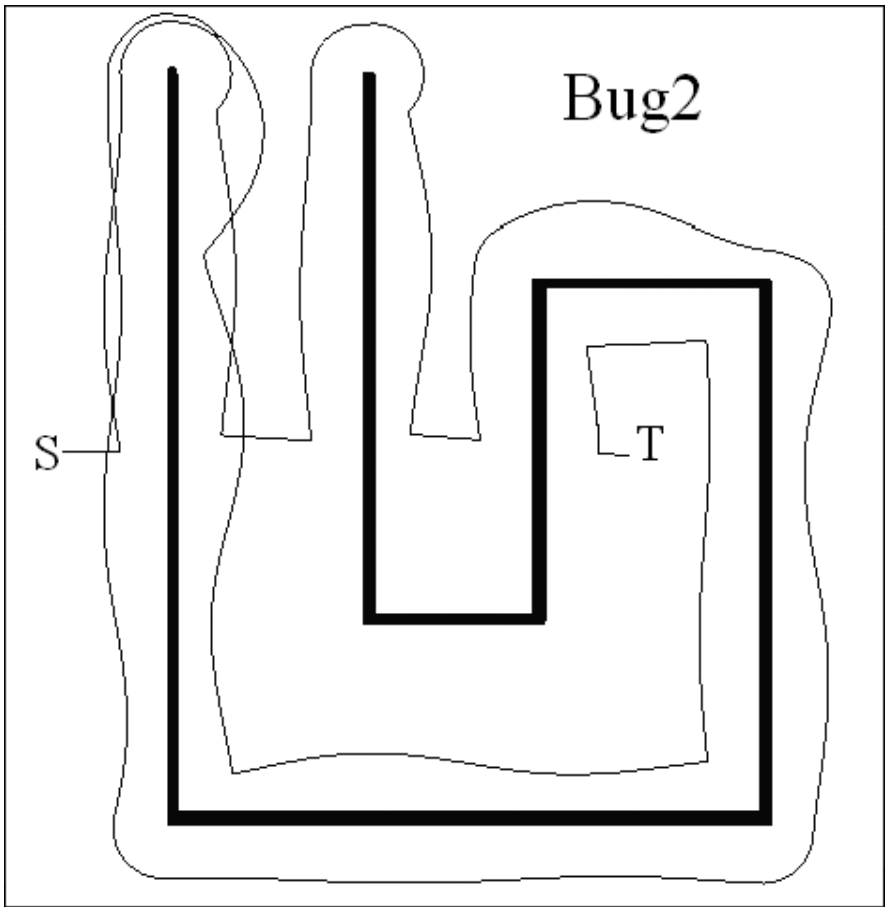

Figure 1-5 The Bug2 Algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.5.3 Alg1

The Alg1 algorithm is an extension of Bug2 invented by Sankaranarayanan and Vidyasagar [3]. Bug2's vulnerability is that it can trace the same path twice and create long paths. To rectify this, Alg1 remembers previous hit and leave points and uses them to generate shorter paths. Alg1 operates as shown in Figure 1-6 with an example in Figure 1-7:

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 1-6 The Alg1 Algorithm

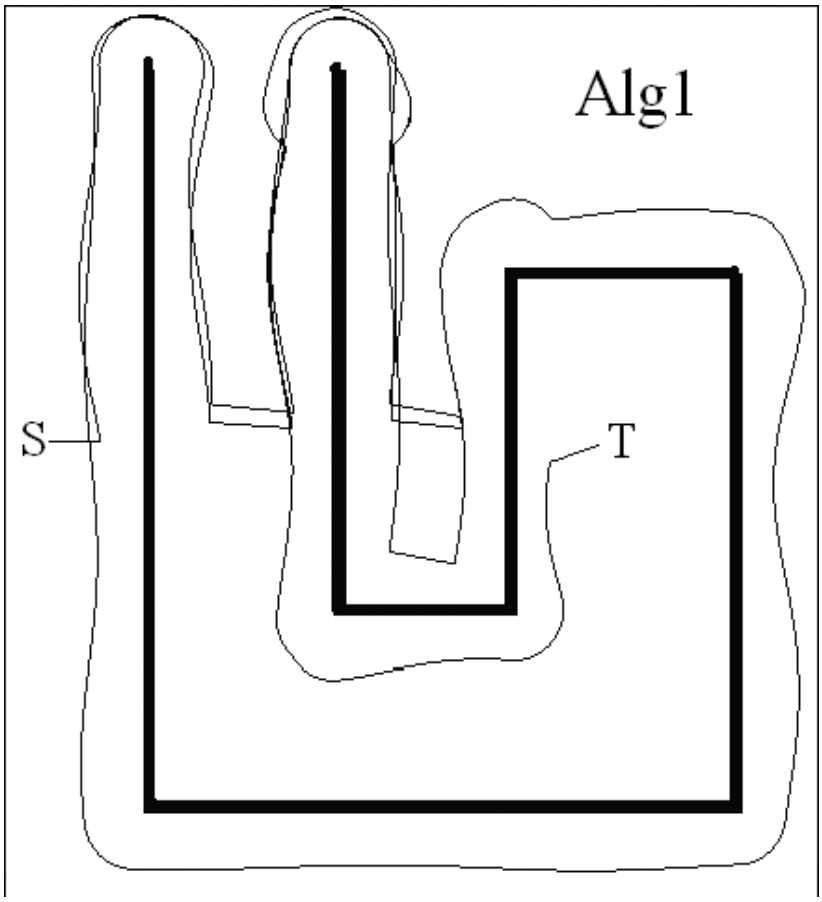

Figure 1-7 The Alg1 algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.5.4 Alg2

The Alg2 algorithm is an improvement from the Alg1 algorithm invented by Sankaranarayanan and Vidyasagar [4]. The robot abandons the M-line concept and a new leaving condition is introduced. Alg2 operates as shown in Figure 1-8 with an example in Figure 1-9:

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 point 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 $d(y,T) < d(Q,T)$ and
  - 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 1-8 The Alg2 Algorithm

Alg2's leaving condition is a great improvement since the robot does not need to be on the M-line to leave the obstacle. It will be shown in the later chapters that this improves performance and it is more computationally efficient. However, such improvements require a method to prevent the Class1 scenario and this will be discussed in Chapter 3.

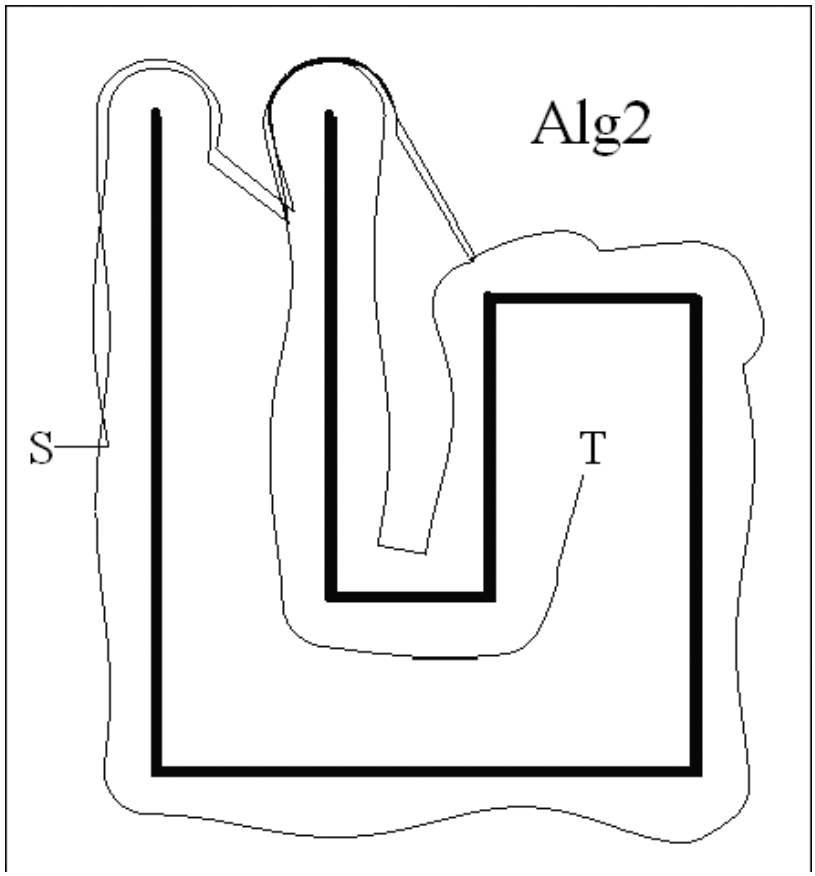

Figure 1-9 The Alg2 algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.5.5 DistBug

The DistBug algorithm was invented by Kamon and Rivlin in [5]. DistBug uses a distance sensor to detect F and uses it in its leaving condition. The algorithm is shown in Figure 1-10 and an example is shown in Figure 1-11.

0) Initialise i=0 and *Step* to the wall thickness (this is the minimum thickness of an obstacle in the environment. It must be entered by the user and is a drawback of this algorithm).

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

Figure 1-10 The DistBug Algorithm

DistBug will also be shown to improve path length performance in chapter 2 and the reasons for this are investigated in depth in chapter 4. In short, it is because each time DistBug checks its ranged based leaving condition, it is actually testing two things. First, whether the robot can use its range sensor to detect a point which is closer to the target than any previously

visited. Second, the STEP criteria is used to prevent the Class1 scenario (refer to page 40 for a description of this scenario and page 78 for the STEP criteria).

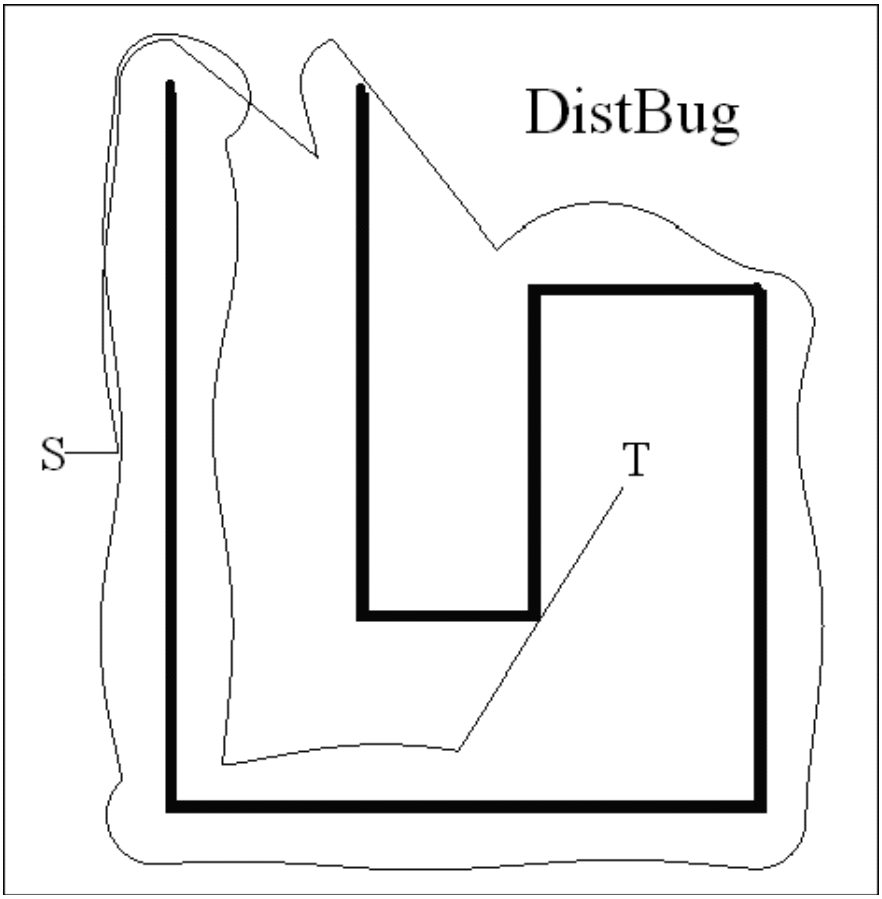

Figure 1-11 The DistBug Algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.5.6 TangentBug

The TangentBug algorithm was developed by Kamon, Rivlin and Rimon [6]. TangentBug uses distance sensors to build a graph of the robot's immediate surroundings and uses this to minimize path length. To understand how the algorithm works, a few path planning concepts are presented as background.

1.5.6.1 The Global Tangent Graph

Consider the environment depicted in figure 1-12(a). Next, consider the convex vertices of all the obstacles which are circled orange in figure 1-12(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 1-12(c). It has been shown that the global tangent graph always contains the optimal path from start to finish [14,18,35]. As expected, figure 1-12(d) shows the optimal path for this particular map.

1.5.6.2 The Local Tangent Graph

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 1-13. 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 1-13.

- 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 1-13.
- 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 1-13. Similarly, if $r(\theta) \neq r$, and $r(\theta)$ subsequently increases such that $r(\theta) = r$, create a node in θ's direction.

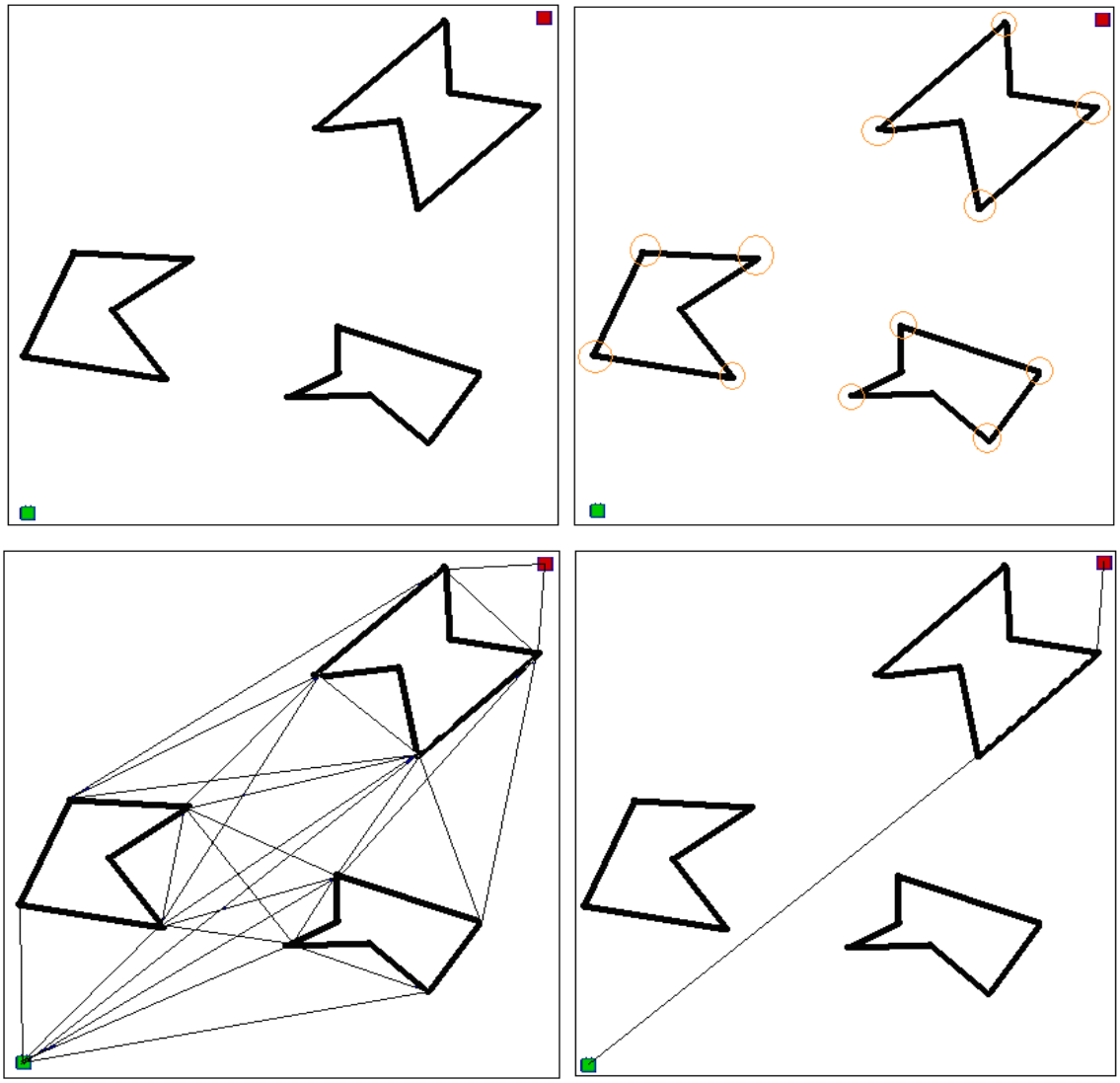

Figure 1-12 (a) Top Left. The environment. (b) Top Right. All convex vertices are circled. (c) Bottom Left. The global tangent graph. (d) Bottom Right. The optimal path.

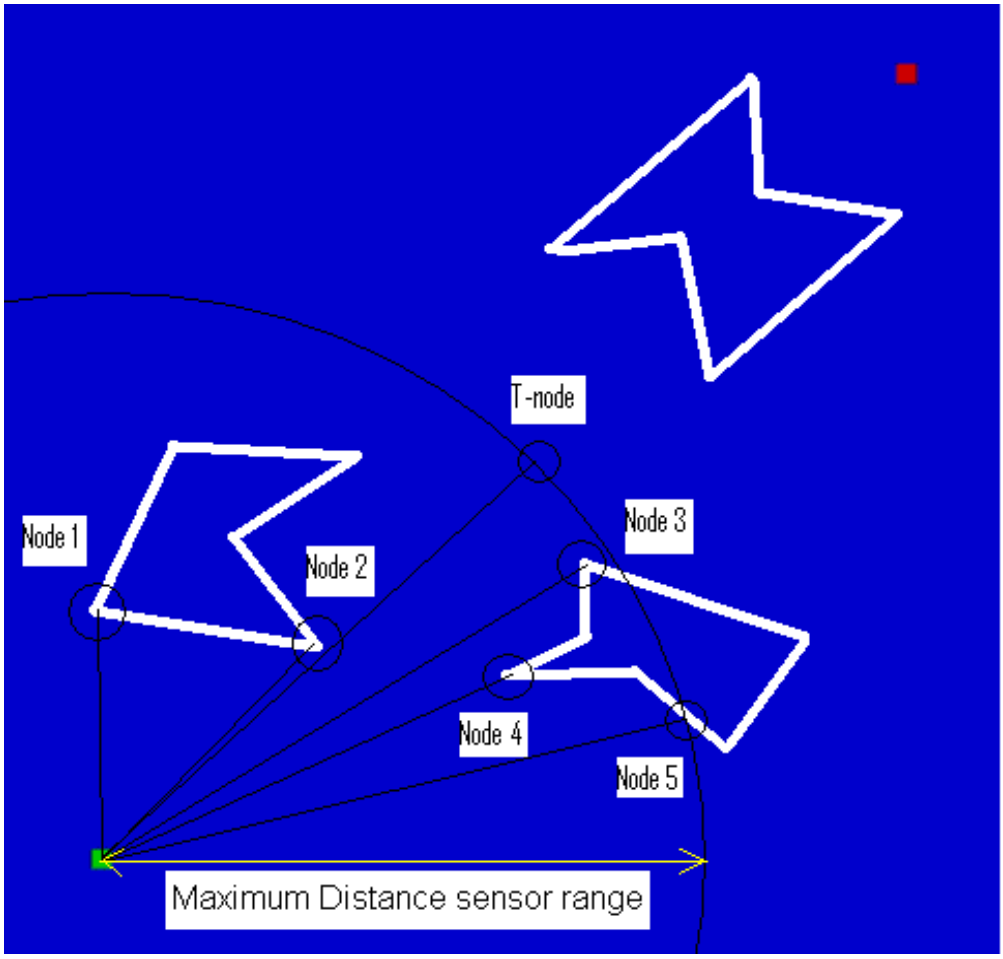

Figure 1-13 The local tangent graph

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 1-13, N^* is the T-node since the T-node is closest to the target.

1.5.6.3 Local Minima

Figure 1-14 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 visible 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.

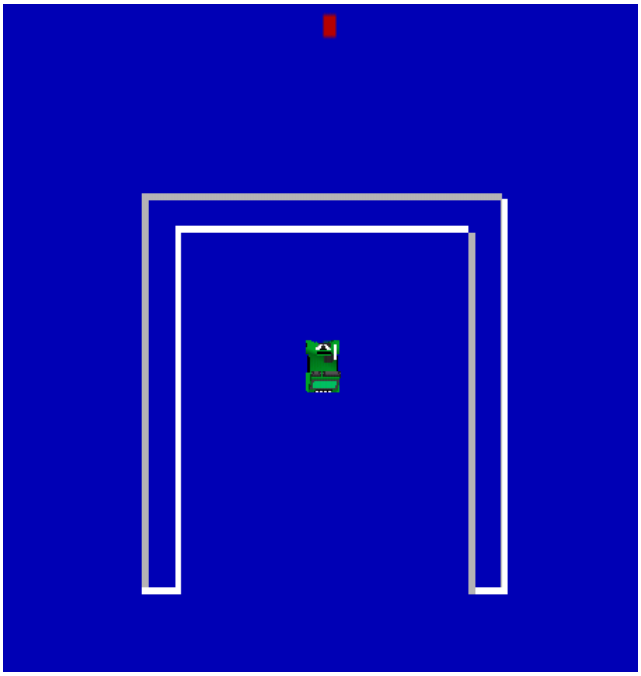

Figure 1-14 The robot in a local minimum

The TangentBug algorithm is illustrated in Figure 1-15.

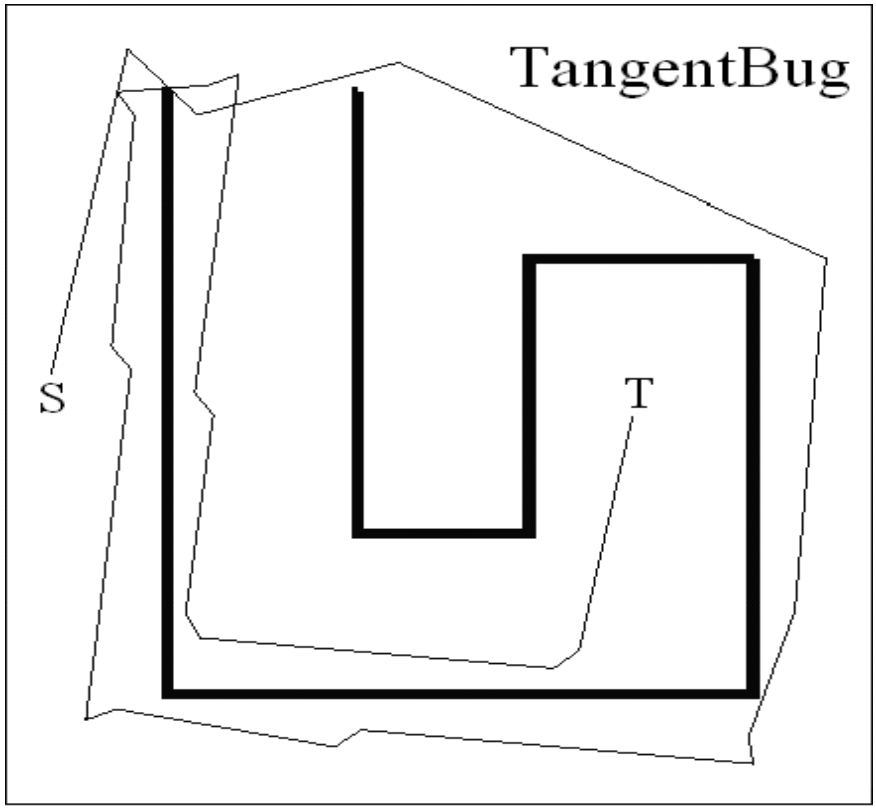

Figure 1-15 The TangentBug Algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.5.7 D*

The D* algorithm was invented by Stentz [15] and is very different from the bug algorithms because it uses mapping. Mapping is prohibited in the Bug Family but this makes an interesting aside. D* 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 [15]. A more abstract, higher-level example is presented in the following sections.

1.5.7.1 Generating an Optimal Path

D* is best explained by example. Let the target be cell (5,3) and the robot's initial position at (1,3) as depicted in figure 1-16(a). Let the traveling cost be 1 when traveling horizontally or vertically and $\sqrt{2}$ when traveling diagonally.

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

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

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

Table 1-1 shows that cells (5,4), (5,2) and (4,3) have the lowest total cost. Those cells set their backpointers towards the target as depicted in figure

1-16(b). Then, the neighbours of T, (5,4), (5,2) and (4,3) are considered for the total minimum cost to target in table 1-2.

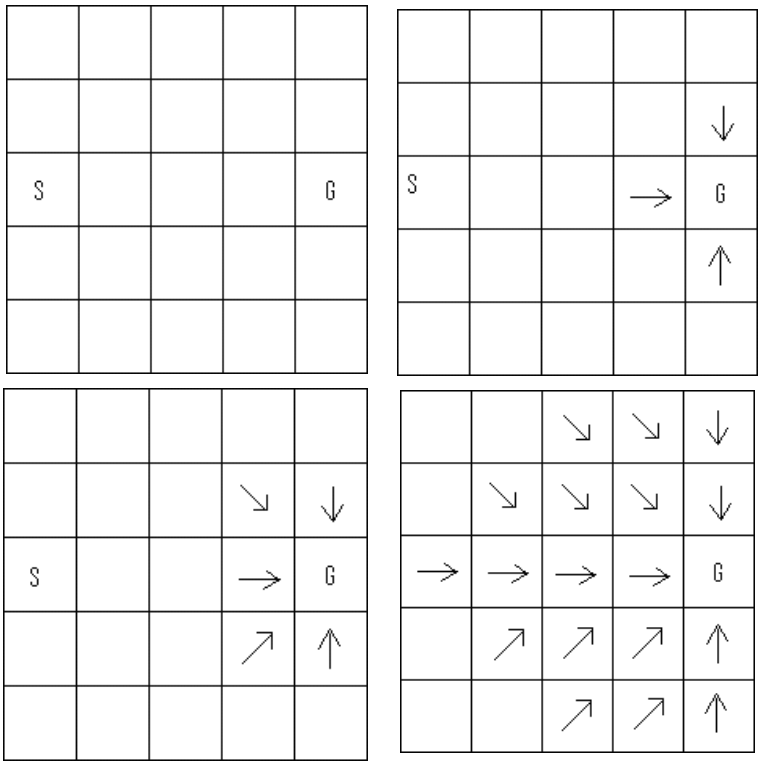

Figure 1-16. (a) Top left. The initial grid. (b) Top right. The grid after data from table 1-5 is entered. (c) Bottom left. The grid after data from table 1-1 is entered. (d) Bottom right. The final grid.

Position (1)	Nearest cell with backpointer or target (2)	Cost from (1) to (2)	Cost from (2) to T	Total Cost
(4,4)	T	1.414	0	1.414
(4,2)	T	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 1-2 The second table generated by D*

Table 1-2 shows that cells (4,4) and (4,2) have the lowest total cost. Those cells set their backpointers towards the target and the grid is depicted in figure 1-16(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 target. Figure 1-16(d) shows the 5x5 grid with T and backpointers leading to T. As can be verified, following any given backpointer trail will produce a path of least cost. This process is how D* generates optimal paths.

1.5.7.2 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 1-17(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 1-3.

Position (1)	Nearest cell with backpointer or target (2)	Cost from (1) to (2)	Cost from (2) to T	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 1-3. The first table drawn after an obstacle was detected at (3,3).

Table 1-3 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 1-17(b). D* repeats this process again and generates table 1-4.

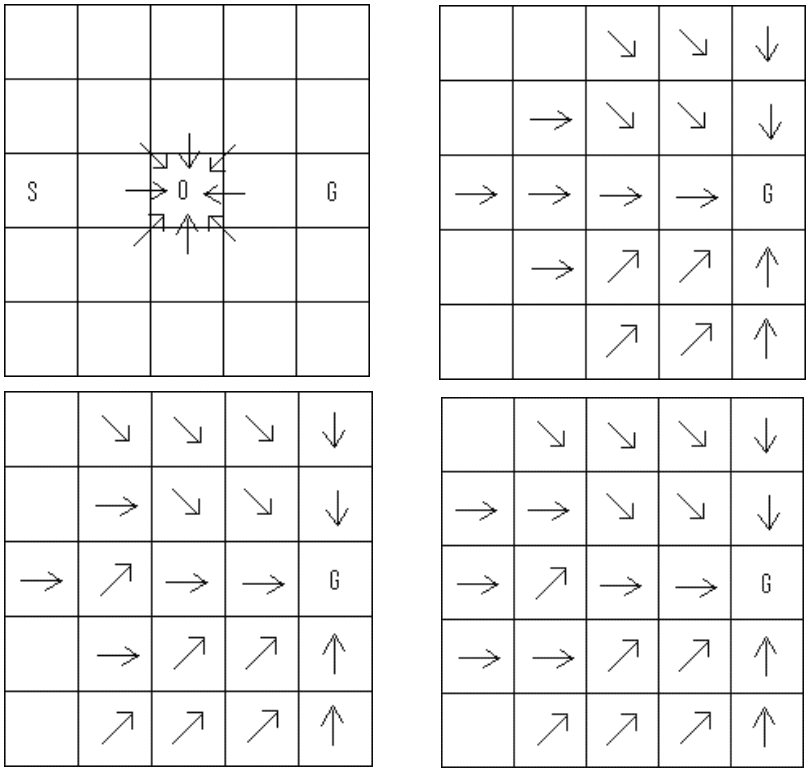

Figure 1-17. (a) Top left. An obstacle cell is identified in position (3,3). (b) Top right. The grid after data from table 1-3 is entered. (c) Bottom left. The grid after data from table 1-4 is entered. (d) Bottom right. The grid after data from table 1-6 is entered.

Position (1)	Nearest cell with backpointer or target (2)	Cost from (1) to (2)	Cost from (2) to T	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 1-4. The second table drawn after an obstacle was detected at (3,3)

Table 1-4 shows that cells (2,3), (2,1) and (2,5) change their backpointers so that their costs to target are minimised. Hence, the updated grid is shown in figure 1-17(c).

D* repeats this process until the minimum total cost in the generated table is greater or equal to the robot's cost to target 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 1-5 is computed:

Position (1)	Nearest cell with backpointer or target (2)	Cost from (1) to (2)	Cost from (2) to T	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 1-5. The third table drawn after an obstacle was detected at (3,3)

The terminating condition holds in table 1-6, and figure 1-17(d) shows the final grid.

Position (1)	Nearest cell with backpointer or Goal (2)	Cost from (1) to (2)	Cost from (2) to T	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 1-6. The forth table drawn after an obstacle was detected at (3,3)

Note that cell (2,3) does not point backwards towards the start. D* maintains optimality and avoids getting stuck in local minimums which have troubled similar techniques [14,18,35]. 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.

1.5.7.3 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 1-18 illustrates D* on an Environment.

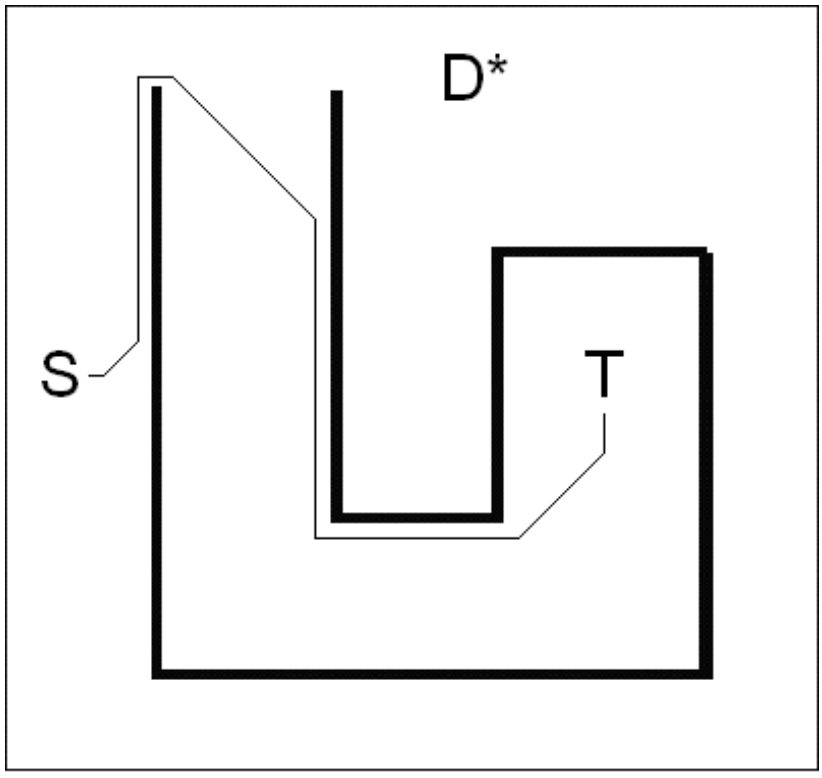

Figure 1-18 D* Algorithm in environment A

1.5.8 Com

The Com [4] algorithm is not an official Bug algorithm and does not guarantee termination. Instead, it is used to illustrate what happens when the robot is allowed to leave for the target whenever it is able to do so. Com is used to develop the Bug algorithms and justify why special leaving rules must exist. It operates as shown in Figure 1-19. In Figure 1-20, the Com algorithm is depicted on an environment. Note that it will never reach the target and instead encircle the obstacle indefinitely.

1) Move toward the target until one of the following occurs:
   - The target is reached. Stop
   - An obstacle is encountered. Follow the obstacle boundary. Go to step 2.

2) Leave if the robot can drive to the target. Go to step 1.

Figure 1-19 The Com Algorithm

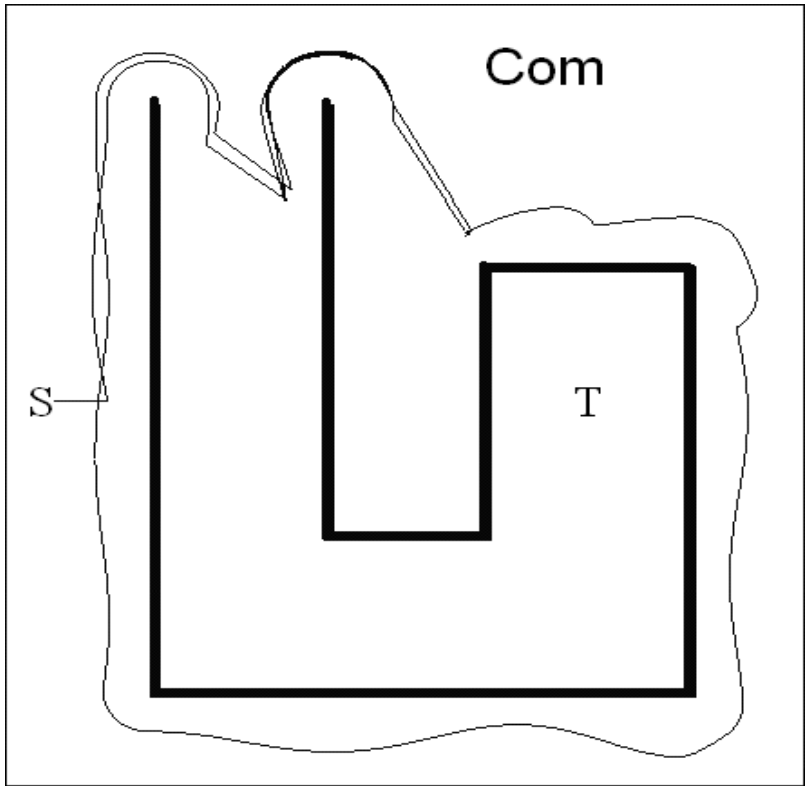

Figure 1-20 Com algorithm in environment A

1.5.9 Class1

The Class1 algorithm [8] is not an official Bug algorithm and does not guarantee finite termination. It is used to illustrate what happens if the robot is allowed to leave if it is closer to the target than any point previously visited and it can travel towards the target. Class1 is used to develop the Bug algorithms and justify why special leaving rules must be applied. It is shown in Figure 1-21 and an example in Figure 1-22:

1) Move toward the target until one of the following occurs:

- The target is reached. Stop
- An obstacle is encountered. Follow the obstacle boundary. Go to step 2.

2) Leave if the robot can drive towards the target and the robot is closer to the target than any point previously visited. Go to step 1.

Figure 1-21 The Class1 Algorithm

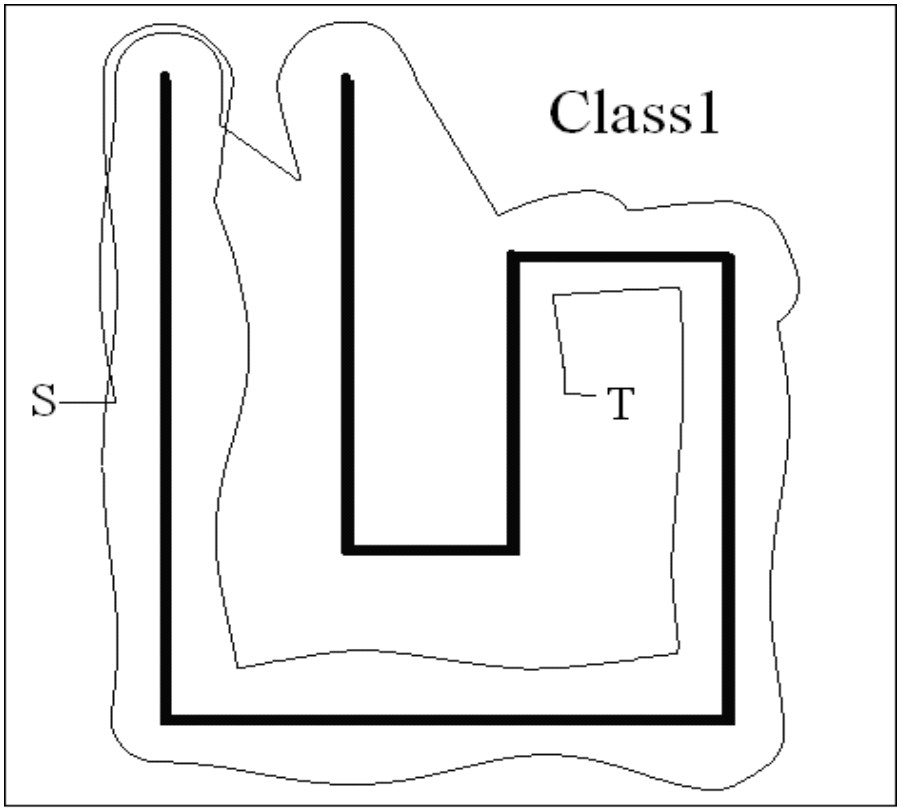

Figure 1-22 Class1 algorithm in environment A

1.5.10 Rev1

The Rev1 algorithm was invented by Horiuchi and Noborio [8]. It operates as shown in Figure 1-23 and an example is illustrated in Figure 1-24.

1. Move towards the target until the following occurs:

(1a) If a robot arrives at T, exit with success.

(1b) If the robot encounters an uncertain obstacle, set a hit point H_i and register the details into the Hlist and Plist.

2. The direction is checked at H_i and Hlist, and if both directions were already checked at H_i, it is immediately eliminated in Hlist. Then, a robot faithfully traces an obstacle by the direction Dir until the following occurs:

(2a) If a robot R arrives at T, exit with success.

(2b) If the distance to target is shorter than any distance previously encountered (the metric condition), the robot is on the M-line (the segment condition) and the robot can go straight to target (the physical condition), then record the leaving details in the Plist, change Dir and go back to step 1.

(2c) If a robot R returns to the last hit point H_i exit with failure. In this case, T is completely enclosed by obstacle boundary.

(2d) If a robot returns to a past hit point, the former point H_k is memorized as the same later point Q_l into Hlist. Then, the robot returns to the hit point H_i using the shortest path as determined by the Hlist and Plist. Once the robot has returned to H_i follow the wall in the opposite direction than previously.

(2e) If the robot returns to a past leave point, the former point is memorized as the same later point into Hlist. Then, the robot returns to the hit point H_i using the shortest path as determined by the Hlist and Plist. Once the robot has returned to H_i follow the wall in the opposite direction than previously.

Figure 1-23 The Rev1 Algorithm

As acknowledged by the authors, Rev1 is very similar to Alg1 except that the robot alternates wall following direction every time it encounters an obstacle and it has the Hlist and Plist mechanisms for better record keeping

purposes.

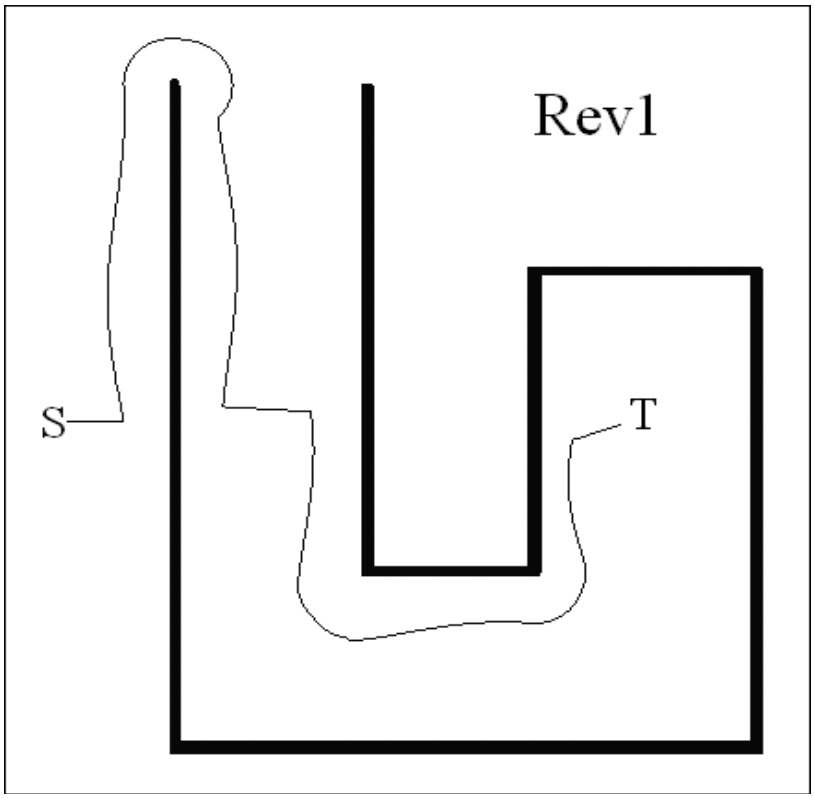

Figure 1-24 Rev1 algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.5.11 Rev2

The Rev2 algorithm was invented by Horiuchi and Noborio [8]. It operates as shown in Figure 1-25 and an example is illustrated in Figure 1-26:

1. Move towards the target until the following occurs:

(1a) If a robot arrives at T, exit with success.

(1b) If the robot encounters an uncertain obstacle, set a hit point H_i and register the details into the Hlist and Plist.

2. The direction is checked at H_i and Hlist, and if both directions were already checked at H_i, it is immediately eliminated in Hlist. Then, a robot faithfully traces an obstacle by the direction Dir until the following occurs:

(2a) If a robot R arrives at T, exit with success.

(2b) If the distance to target is shorter than any distance previously encountered (the metric condition) and the robot can go straight to target (the physical condition), then record the leaving details in the Plist, change Dir and go back to step 1.

(2c) If a robot R returns to the last hit point H_i exit with failure. In this case, T is completely enclosed by obstacle boundary.

(2d) If a robot returns to a past hit point, the former point H_k is memorized as the same later point Q_l into Hlist. Then, the robot returns to the hit point H_i using the shortest path as determined by the Hlist and Plist. Once the robot has returned to H_i follow the wall in the opposite direction than previously.

(2e) If the robot returns to a past leave point, the former point is memorized as the same later point into Hlist. Then, the robot returns to the hit point H_i using the shortest path as determined by the Hlist and Plist. Once the robot has returned to H_i follow the wall in the opposite direction than previously.

Figure 1-25 The Rev2 Algorithm

As acknowledged by the authors, Rev2 is very similar to Alg2 except that it alternates wall following direction. Therefore, the only difference between Rev2 and Rev1 is that the segment condition has been removed from (2b).

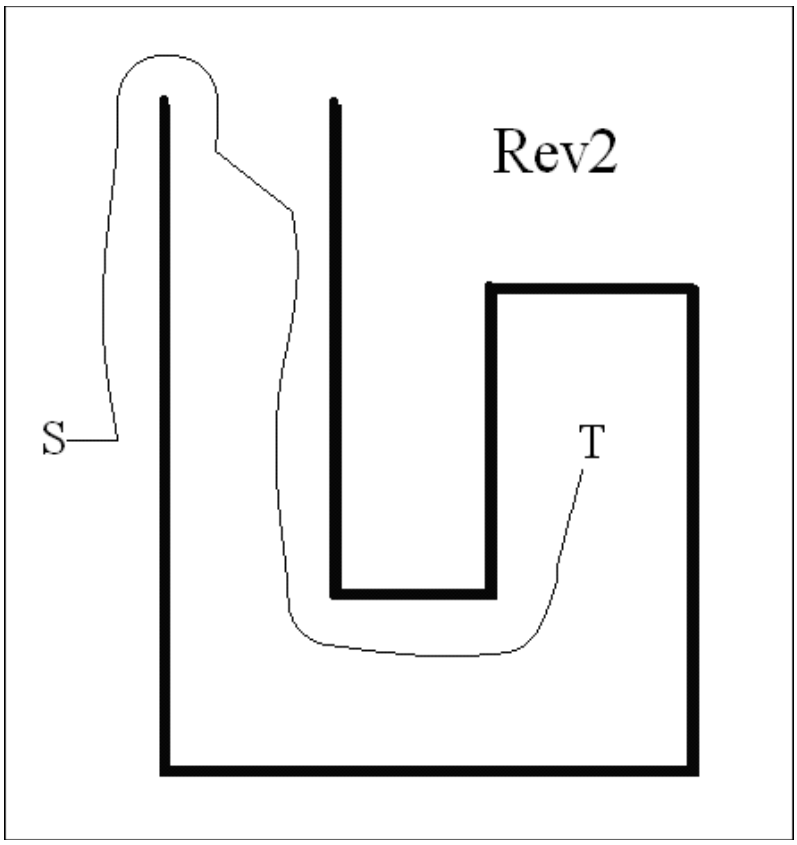

Figure 1-26 Rev2 algorithm in environment A

For more examples, refer to figure 3-4, 3-6, 3-8 and 3-10.

1.6 Other Bug Algorithms

Other Bug algorithms are listed in this section. These algorithms are either very similar to an algorithm which has been simulated or had not been published at the time when the simulations were run but have since become available.

1.6.1 HD-I

Algorithm HD-I is identical to Rev1 except for the selection of wall following direction which is based on perceived distance to target, target direction and past following directions. It has been shown [8] that the path length is reduced on the average.

1.6.2 Ave

The Ave algorithm was invented by Noborio, Nogami and Hirao [8] is an improvement of HD-I. The algorithm is quite lengthy and can be found in [8]. The main improvement of Ave was a better mechanism for determining which wall following direction to take. The decision is based on all past node data instead of just the current node and perceived distance to target.

1.6.3 VisBug-21

The VisBug-21 [16] algorithm allows a robot to follow the Bug2 path using range sensors. The robot uses the range sensors to find shortcuts and takes them to reduce length.

1.6.4 VisBug-22

The VisBug-22 [16] algorithm takes more risk in respect to finding

shortcuts. The path length can be much shorter if the shortcut proves fruitful, however the path length can also be a lot longer if the shortcut does not.

1.6.5 WedgeBug

The WedgeBug algorithm was developed by Laubach and Burdick [10]. It involves scanning in wedges. The first wedge is one which contains the direction towards the target and if there are no obstacles then the robot moves towards the target. Otherwise, the robot performs "virtual boundary following", in which more wedges are scanned and the robot follows the wall via its use of range sensors. The advantage of WedgeBug was that it does not need to generate the full LTG as in TangentBug. It scans in wedges only when is required, thus saving resources.

1.6.6 CautiousBug

The CautiousBug algorithm was developed by Magid and Rivlin [11]. It involves spiral searching in which the robot repeatedly changes wall following direction during the boundary following mode. As such, the robot is not dependent on a favourable choice of wall following direction but the trade-off is that a longer path is produced on average.

1.6.7 3DBug

3DBug was developed by Kamon, Rimon and Rivlin [64]. It is an extension of the TangentBug algorithm and operates in 3 dimensions instead of the typical 2. There were several problems encountered the largest being surface exploration of the obstacle instead of simply following the boundary of an obstacle.

1.6.8 Angulus

The Angulus algorithm [13] was developed by Lumelsky and Tiwari. Based on ideas originally found in the Pledge algorithm [21], the angulus algorithm bases leaving decisions on two variables: a, the angle between the lines x (the robot's current position) to T and S to T and b, the angle between the line S to T and the robot's current velocity vector.

Typically, Bug algorithms require something which measures the robot's current distance to the target. This is often based on dead-reckoning from the original start to target positions. However, Angulus does not require any range measurements and can perform with only a compass. This makes it more resilient to error.

1.6.9 Optim-Bug

The Optim-Bug algorithm [12] was developed by Kriechbaum. Unlike other Bug algorithms, Optim-Bug builds a map of its environment as it senses its surroundings with infrared sensors. With all prior knowledge of its surroundings, it is able to eliminate the need for a boundary following mode. Instead, in each navigation cycle, the robot calculates the shortest path to target taking into account the currently known surroundings and the map. It follows this path for one cycle and then recomputes the shortest path to target. The algorithm terminates as soon as the target is reached or the shortest path is known to cross an obstacle.

1.6.10 UncertainBug

UncertainBug [12] is similar to Optim-Bug except that it takes into account uncertainty in the robot's path. It always computes the optimal path to target for each step given the currently available information but it also

aims to minimize the uncertainty of the robot's final pose with respect to the target. To reduce error, UncertainBug directs the robot away from the optimal path and towards known landmarks which reduce localization uncertainty. The trade-off is, as Kriechbaum noted, that the robot may not always reach the target even if such a path exists.

1.6.11 SensBug

SensBug [71] is specially designed for construction environments. In such environments the obstacles are assumed to be “simple” in that they are all curves of finite length. Therefore, the leaving requirements are relaxed and the robot is permitted to leave as in the Com algorithm. In these environments, obstacles are also able to move and it was shown that SensBug is still able to navigate successfully. Also, the robot is equipped with a wireless communication device to assist it in determining which direction to following the obstacle so that path length can be shortened.

1.6.12 K-Bug

K-Bug [87] was designed by Langer, Coelho and Oliveira. If the path to target is obstructed, K-Bug evaluates the vertices which surround the robot and directs the robot to travel to the nearest one. Then, K-Bug's behaviour is described assuming complete knowledge of the environment and how the optimal path can be found. This section appears to contradict one of the fundamental assumptions of the Bug algorithms in that the environment must be unknown. Otherwise, there are plenty of algorithms which can be used to find a path in a completely known environment.

1.7 Anytime algorithms

These are a special sub-class of the Bug algorithms in which the environment is known before starting. As a result, they can provide the optimal path, but they can also produce intermediate, sub-optimal paths as soon as they are available.

1.7.1 ABUG

The ABUG algorithm was developed by Antich, Ortiz and Minguez [37, 38]. It then divides the environment into discrete cells and combines the A* [39, 40] algorithm with Bug2 to solve the Bug problem. Each time the robot encounters an obstacle, the left and right routes are considered parts of a binary tree. Once the tree is built, the A* algorithm is used to find the shortest path.

1.7.2 T^2

The T^2 algorithm was developed by Antich and Ortiz [41]. It is based upon the principles of the potential field approaches [42, 43, 44]. The problems with the potential field approaches was that the robot could become stuck in an environment such as Environment B in chapter 2. T^2 overcomes this by combining it with the Bug boundary following mode. By doing this, T^2 forces the robot to move away from the target in an attempt to find other routes to the target.

T^2 can be thought of as an extension of Bug2 with two key differences. Firstly, the robot is allowed to leave when it notices that it can drive towards the target. Leaving at such points is only permitted once in the entire journey and if leaving occurs, the robot redefines the M-line to begin at the leave point. Secondly, the robot is also permitted to leave under

Bug2 rules with redefined M-lines.

1.8 Structure of this Thesis

This thesis aims to improve the bug algorithms in respect to some desired performance measure. Traditionally, this has been path length but this can also include reduced computation, processing and scanning.

In the second chapter, a comparison of the simulation results and exploring implementation issues is presented. EyeSim simulation results are also presented and the foundation for further performance investigation is laid.

The third chapter presents an analysis of leaving conditions. It is argued that these conditions are fundamentally very similar and vary only with respect to how they ensure that the number of leave points or hit points was kept finite.

The fourth chapter presents results of the Bug algorithms on an environment with a single semi-convex obstacle. This subclass of obstacle produces some interesting performance results and these are investigated mathematically.

The fifth chapter presents Curv2, an improved algorithm for following a trail. Curv3 is also presented and it is suited to pairing start and targets when there are multiple trails.

The sixth chapter presents SensorBug. This algorithm uses the Q method developed in chapter 4 along with range sensors. The range sensor use is kept to a minimum.

The conclusion summarizes the thesis with its key findings. Also, areas of future research are presented and these are the areas where the most promising results lie.

In the appendix, implementation details on the EyeSim simulation system are provided. It will consist of a high level overview of the code structure and organization as well as some sample code.

Chapter 2

Performance Comparison of Bug Navigation Algorithms

2.1 Introduction

Eleven variations of Bug algorithm have been implemented and compared against each other on the EyeSim simulation platform [26]. This chapter discusses their relative performance for a number of different environment types as well as practical implementation issues.

The robot has to either reach the target position – or terminate if the target is unreachable – it must not map its environment. Therefore, a particular navigation algorithm can have a statistically better performance than another, but may not be better for any possible environment setting. For example, Alg1 is supposed to improve on Bug2 but is shown later that this is not always the case.

Since every algorithm in the Bug family has to have the termination property, subsequently published Bug algorithms try to improve the algorithm performance, e.g. the path length or time required to either reach or to detect that the target is unreachable. The aim is to identify the navigation techniques that work best by conducting unbiased performance comparisons of various Bug algorithms, based on empirical data from experiments in different environments.

Section 2.2 introduces two new Bug algorithms which have been

implemented. These are new to the Bug family and the introduction here is necessary as they have been implemented for simulation. The full rationale is discussed in section 3. Section 2.3 discusses theoretical differences between Bug algorithms, as well as practical implementation issues. Section 2.4 presents simulation results from eleven Bug algorithms in four different environments and also discusses algorithm implementation complexity. Section 2.5 presents conclusions and also touches on fault tolerance issues in noisy environments.

2.2 LeaveBug and OneBug

LeaveBug is similar to Bug1, except that instead of circumnavigating the entire obstacle before evaluating the line segment $[Q_m,T]$, the robot evaluates this condition after completing each path segment that does not prevent movement towards the target. Full pseudo code is presented in chapter 3.

OneBug is similar to Alg2, except that no stored points are used. Instead the robot completely explores a segment along the blocking obstacle that prevents movement towards the target. Full pseudo code is presented in chapter 3.

2.3 From Theory to Implementation

Since Bug algorithms are usually published as pseudo code, they do leave some room for interpretation. Therefore, it is important to specify all adaptations required to transform them into proper executable algorithms. The RoBIOS application programmer interface has been used [7], which is compatible with real SoccorBot mobile robots as well as EyeSim simulation system. Below is a discussion of some of the issues

encountered during the implementation phase:

2.3.1 Update Frequency

In theory, Bug algorithms continuously update their position data and will automatically detect that any of the navigation conditions are satisfied. For example, in Bug2 as soon as the robot lies on the M-line, the algorithm will detect this and act accordingly.

In practice, this will require the robot's position data to be updated and the navigation conditions to be checked. The robot's position is based on dead reckoning and for every update wheel encoders must be read and calculations must be performed. Clearly, this requires computation resources and updates cannot occur too frequently.

Initially, updating robot position and checking was done as a background thread. However, with that approach came inherent unpredictability especially if the robot was moving at high speeds or on an irregular wall following path. Furthermore, interfacing the thread with the main program required much programming effort.

It was found through experiments that a distance of 40 mm between updates achieved the optimal balance between updating too frequently and too infrequently on the EyeSim simulator. Thus, the robot drives 40mm either driving towards the target or following the wall and then updates its position. This implies that any position of significance must include some margin for error and this is discussed next.

2.3.2 Recognition of Stored Positions

In theory, Bug algorithms use infinitesimally small points to represent the start, target, latest hit point and other significant positions in Alg1, Alg2, Rev1 and Rev2. This does not work in practice, because of the limited update frequency and subsequent deviations during wall-following. Hence, in the implementation, each significant position is represented by a square of side length 50 mm. A square was chosen because it is computationally efficient to check if the robot is inside. The size of the square was chosen such that it is slightly larger than the robot's driving distance during an update cycle, but not so large that it would lead to frequent false positives.

2.3.3 Robot Sensor Equipment

Some algorithms only require tactile sensors, for example Bug1 and Bug2. In these algorithms, range sensors are used as substitute tactile sensors. The range sensors assist only for wall-following and wall-detecting purposes.

2.3.4 Moving Towards Target

In all Bug algorithms, the ability to check if the robot can move towards the target at its current location is essential. For instance, Bug1 requires Q_mT to be checked in its test of target reachability. Also, Bug2, Alg1, Alg2, Rev1 and Rev2 all require this check to be made on any prospective leave points.

In theory, the robot is able to use its tactile sensor to evaluate this check. In our implementation, this check is performed by obtaining the free-space in the target's direction and comparing it to a predefined value. Through experiments, it has been found that a value of 270mm works adequately. This value allows the robot to rotate on the spot and align itself parallel

with the wall. Once parallel, the robot can follow the wall at a safe distance. Further, 270mm allows the robot to stop in plenty of time in case the check is delayed.

To obtain the free-space in the target's direction, the robot points one of its eight range sensors in the target's direction such that rotation is minimized.

2.3.5 Wall Following

Lumelsky [1] notes that special algorithms beyond the scope of the Bug algorithms are required to follow a wall [94]. In our implementation, the robot uses a simple proportional-derivative (PD) controller to follow the wall. The code is given in Figure 2-1. The current error is the difference between the distance to the wall and the desired distance. The derivative is the difference between the new error and the previous error. The amount of curvature is a proportion of the error and the derivative.

```
void follow_wall_straight(bool is_on_right){
     double dist;
     double derivative;
     double new_error;

     if(is_on_right){
          dist = PSDGet(psd_right);
          new_error = dist-WALL_DISTANCE;
          derivative = new_error - old_error;
          curve(STEP, - new_error/KP - derivative/KD);
          old_error = new_error;
     }
     else{
          dist = PSDGet(psd_left);
          new_error = dist-WALL_DISTANCE;
          derivative = new_error - old_error;
          curve(STEP, new_error/KP + derivative/KD);
          old_error = new_error;
     }
}
```

Figure 2-1 The code for following the wall

This works well if the robot is following a “gentle” curve, but an obstacle's perimeter can be arbitrary. Hence, there are situations where the robot has to perform special movements. For instance, if there is a wall directly ahead of the robot and to its right, the robot will rotate counter-clockwise on the spot until it can drive forwards again. This situation is illustrated in Figure 2-2, left. If there are no walls surrounding the robot, the robot drives in a circular pattern until it detects a wall on the right or it detects a wall ahead of it. This is illustrated in Figure 2-2, right.

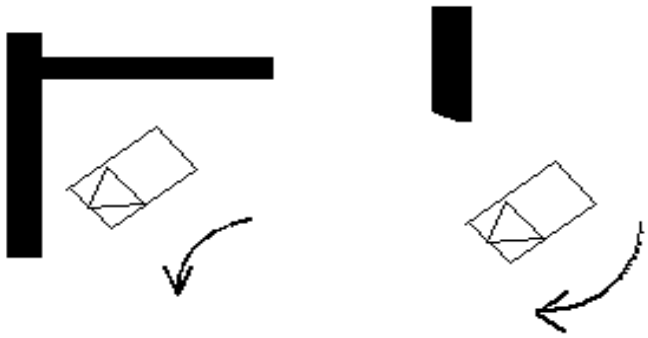

Figure 2-2 Left: Robot rotates on the spot Right: Robot drives in a circle

Recently, Charifa and Bikdash [45] proposed a Boundary Following Algorithm to specifically address this problem. Their algorithm is based a local-minimum-free potential field. When following the wall, the robot must keep a safe distance from the obstacles to avoid collision. To achieve this, elements of generalized Voronoi diagrams [46, 47, 48, 49] are used since Voronoi diagrams find points which are furthest from all obstacles. However, the robot must also follow obstacles closely. Hence, elements of reduced visibility graphs [50, 51] are used since visibility graphs achieve the shortest path with no clearance. A balance is struck and the result is a safe Boundary Following Algorithm that can be implemented in future Bug algorithms.

In addition, Lee [99] proposed a Rough-Fuzzy Controller for wall-following navigation. This controller uses fuzzy logic [102] rough-membership functions [100,101] to improve its uncertainty reasoning. Lee tested the boundary following controller on a real robot and the results show that it exhibited show that it outperformed a bang-bang controller, PID controller, a conventionally fuzzy controller and an adaptive fuzzy controller using Genetic Algorithms [103,104]. However, the performance used to evaluate the controllers does not include path safety so no direct comparison can be made with Charifa.

2.3.6 Limited Angular Resolution for the LTG

In theory, the Local-Tangent-Graph (LTG) should be continuous. In practice, range sensors have a finite angular resolution. For our robot model this resolution is 1 degree between each sample. To identify nodes, successive values are compared against a discontinuity threshold. If the difference is larger, a node is identified. This can lead to an error where a node is mistakenly identified, as illustrated in Figure 2-3. To reduce the possibility of these errors, the sensor range is restricted and the robot is programmed to move away from an obstacle if it comes too close.

2.3.7 M-line Identification

Bug2, Alg1 and Rev1 use the concept of the M-line that links start and target positions. Checking if the robot is positioned on the M-line is essential. Consider the situation where “S” is at the origin, “T” is a vector to the target, “P” is a vector to the robot's current location and “a” is a scalar such that the vectors “P-aT” and “T” are perpendicular. Figure 2-4 illustrates this situation.

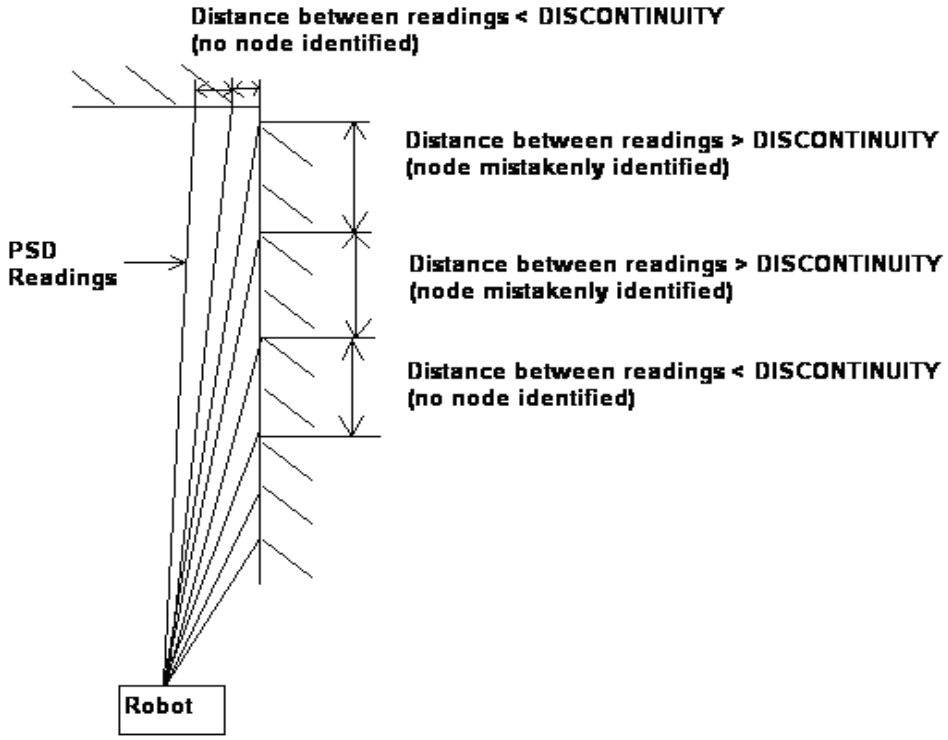

Figure 2-3: Finite angular resolution causes incorrect node identifications

It follows from the dot product that:

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

Rearranging for "a" gives:

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

If $0 \leq a \leq 1$ and the Euclidean distance between "P" and "aT" is smaller than a threshold value, the robot is on the M-line.

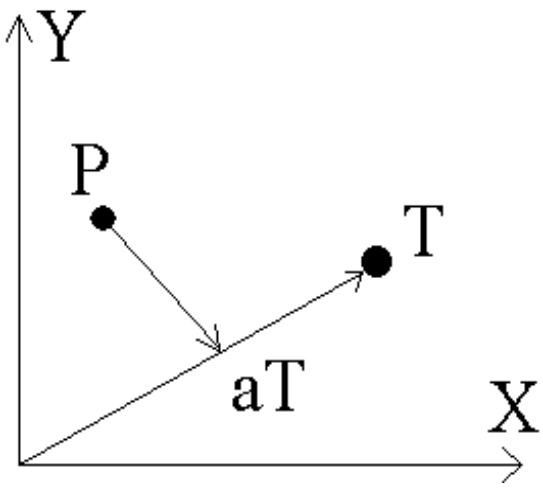

Figure 2-4: The vectors “P-aT” and “T” are perpendicular

2.4 Experiments and Results

For these experiments, a simulation setting without sensor or actuator noise has been selected. Figure 2-5 illustrates the Bug algorithms on environment B, featuring a local minimum. Several early navigation techniques such as the potential field method [14] had difficulties overcoming local minimums. In theory, no Bug algorithm should have difficulty overcoming a local minimum and this is verified in our implementation.

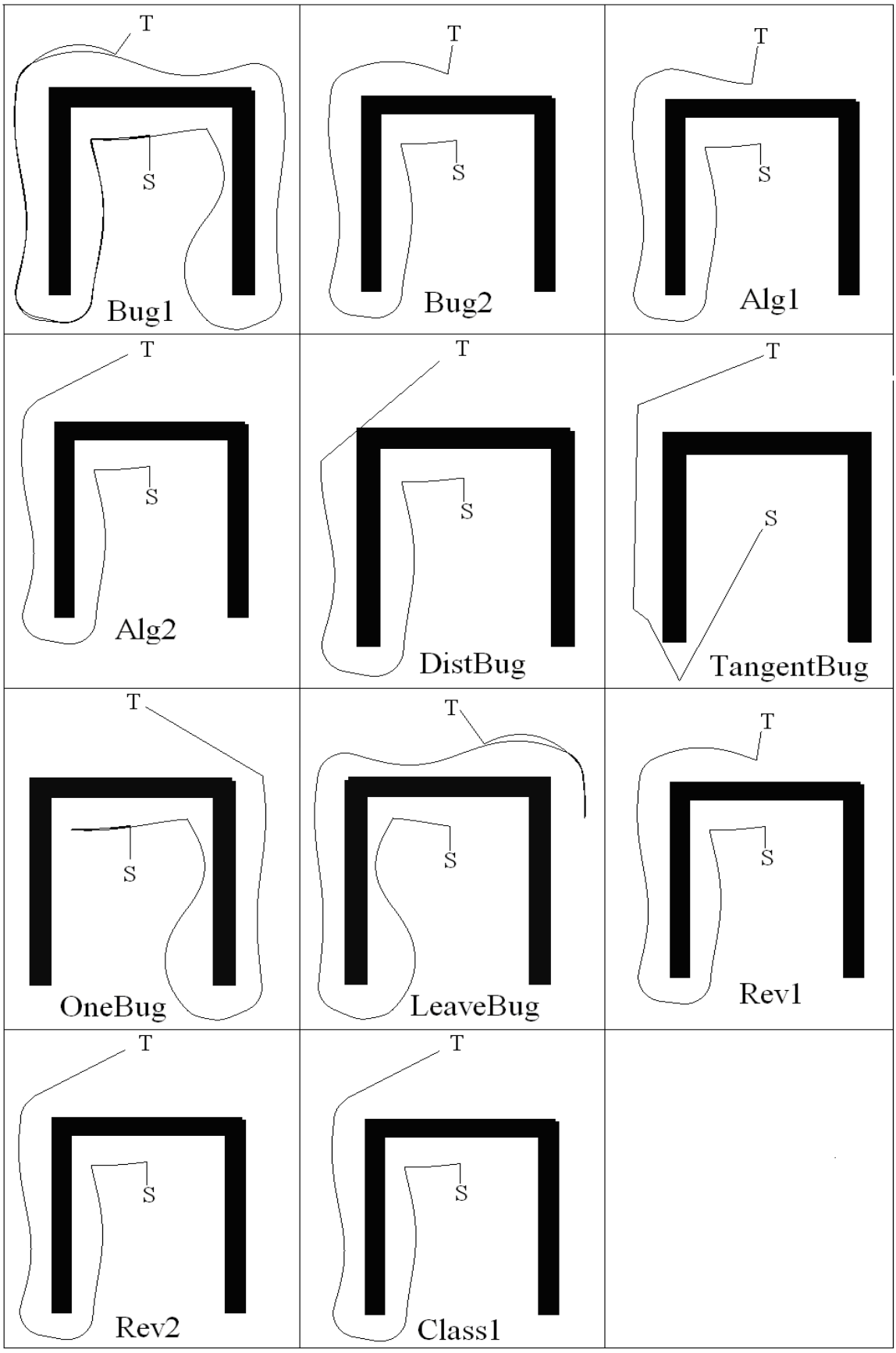

Figure 2-5: Paths for Bug algorithms in environment B

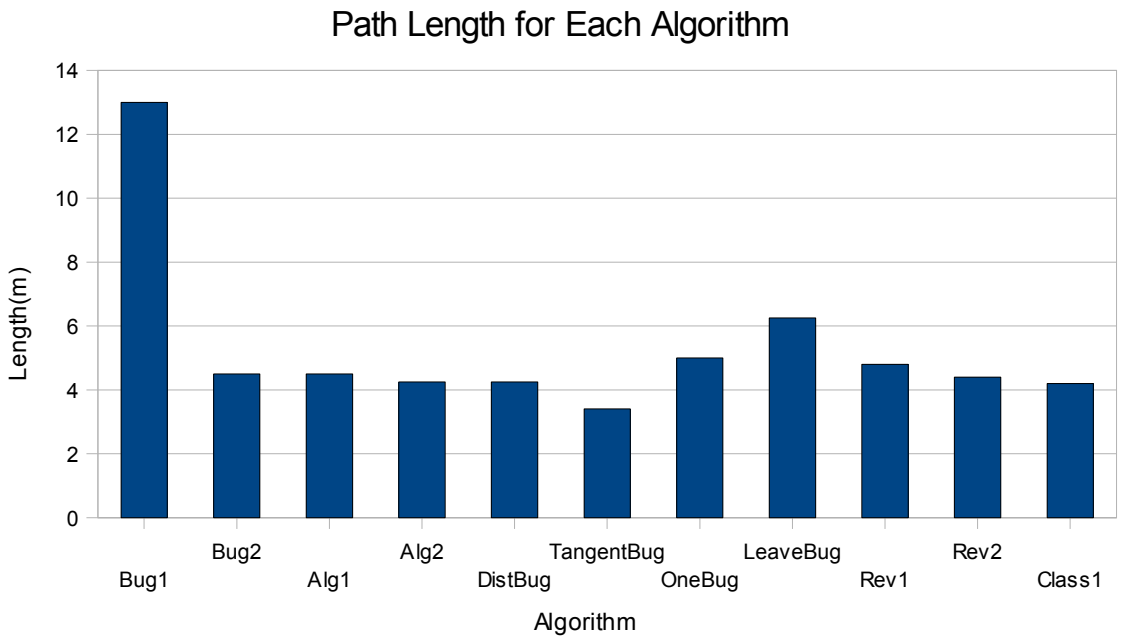

Figure 2-6: Path lengths for environment B

Bug1 has the longest path length, followed by LeaveBug, while all other algorithms have a similar small path length. This is due to the fact that Bug1 does not check for target reachability until it re-encounters the first hit point (collision point) with the U-shaped obstacle. This unnecessarily makes the robot circumnavigate the complete obstacle, while other algorithms depart towards the target much earlier. (Figure 2-6)

Figure 2-7 illustrates the algorithms on a terrain originally created by Sankar [3]. The beauty of this terrain is that it is complicated enough to reveal the unique characteristics of each algorithm but not so complicated as to be overwhelming. For instance, the M-line is clearly visible in Bug2, Alg1 and Rev1 as is the stored points concept in Alg1 and Alg2. DistBug's leaving condition allows it to leave slightly earlier than Alg2 and Rev2, resulting in a shorter path length. The overall shortest path was reached by Rev2 followed by Rev1. This can be attributed to the alternative wall following.

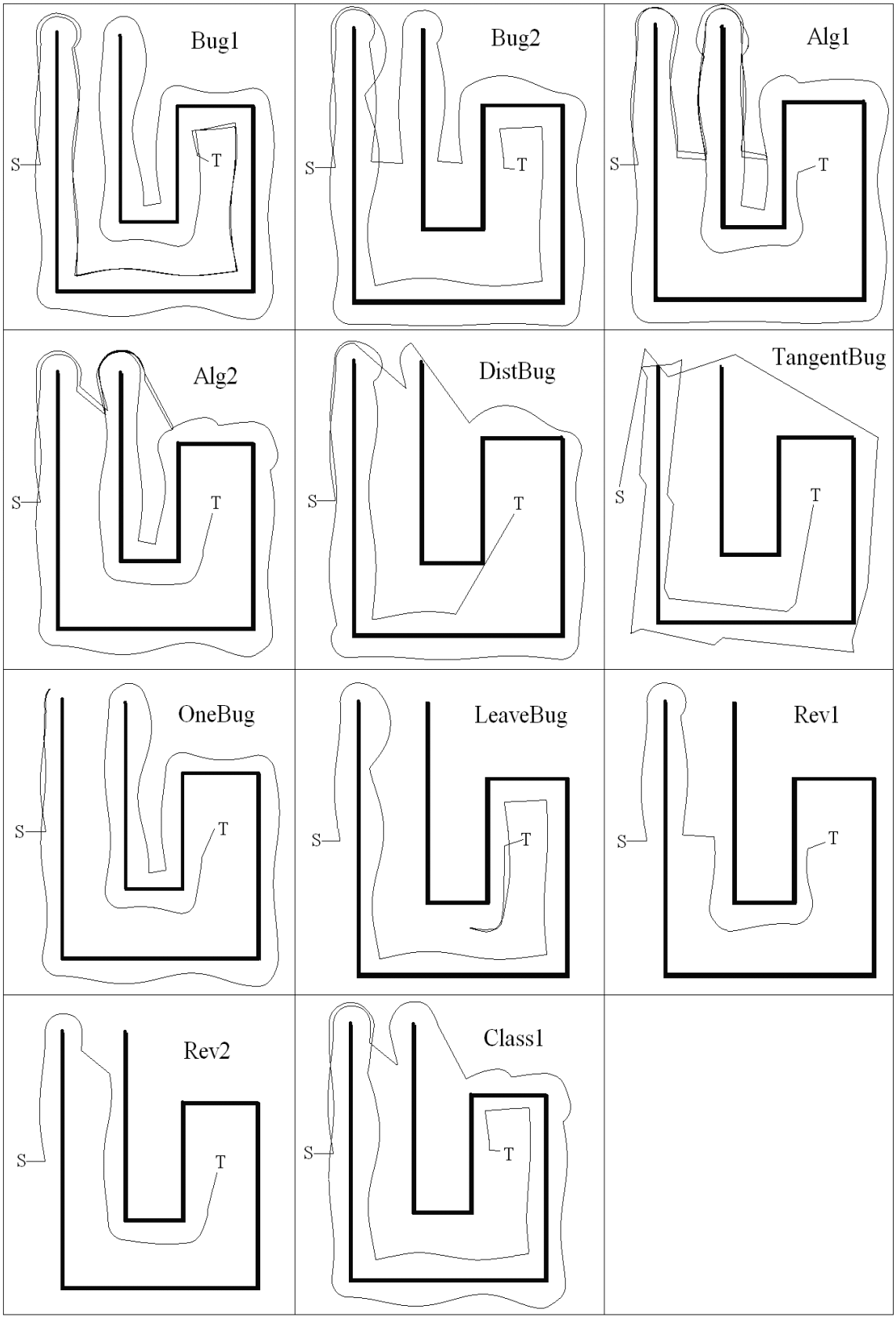

Figure 2-7: Paths for Bug algorithms in environment A

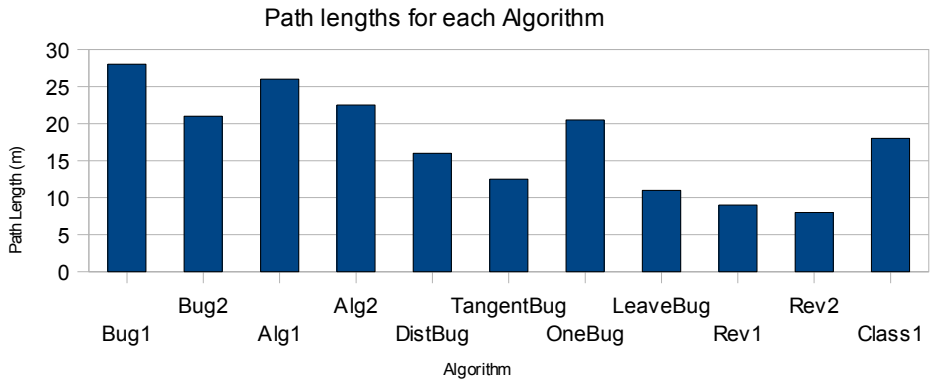

Figure 2-8: Path lengths for environment A

Some interesting issues arise when comparing Alg1 against Bug2 and Alg2 against DistBug. Recall that Alg1 is very similar to Bug2 except that Alg1 uses stored points and Alg2 is very similar to DistBug except for stored points and the inclusion of the range-based leaving condition. Interestingly, Bug2 and DistBug produce shorter paths than their stored point counterparts. (Figure 2-8)

Figure 2-9 illustrates the algorithms on a terrain featuring only a single semi-convex obstacle. Consider the convex hull associated with any obstacle. If all differences between the obstacle and the convex hull are convex, then the obstacle is called semi-convex.

In this environment the shortest path is produced by TangentBug. This is because TangentBug can use the LTG (local tangent graph) to sweep all areas of a discrepancy between the convex hull and the semi-convex obstacle, since the discrepancy must be convex. Hence, it can travel along the convex hull. Second best after TangentBug is DistBug. Clearly, its use of range sensors allow it to leave the obstacle earlier than Alg2 and this results in a shorter path. Rev1 and Rev2 did not perform well for this environment, since they conducted unnecessary circumnavigation due to a boundary following direction decision.

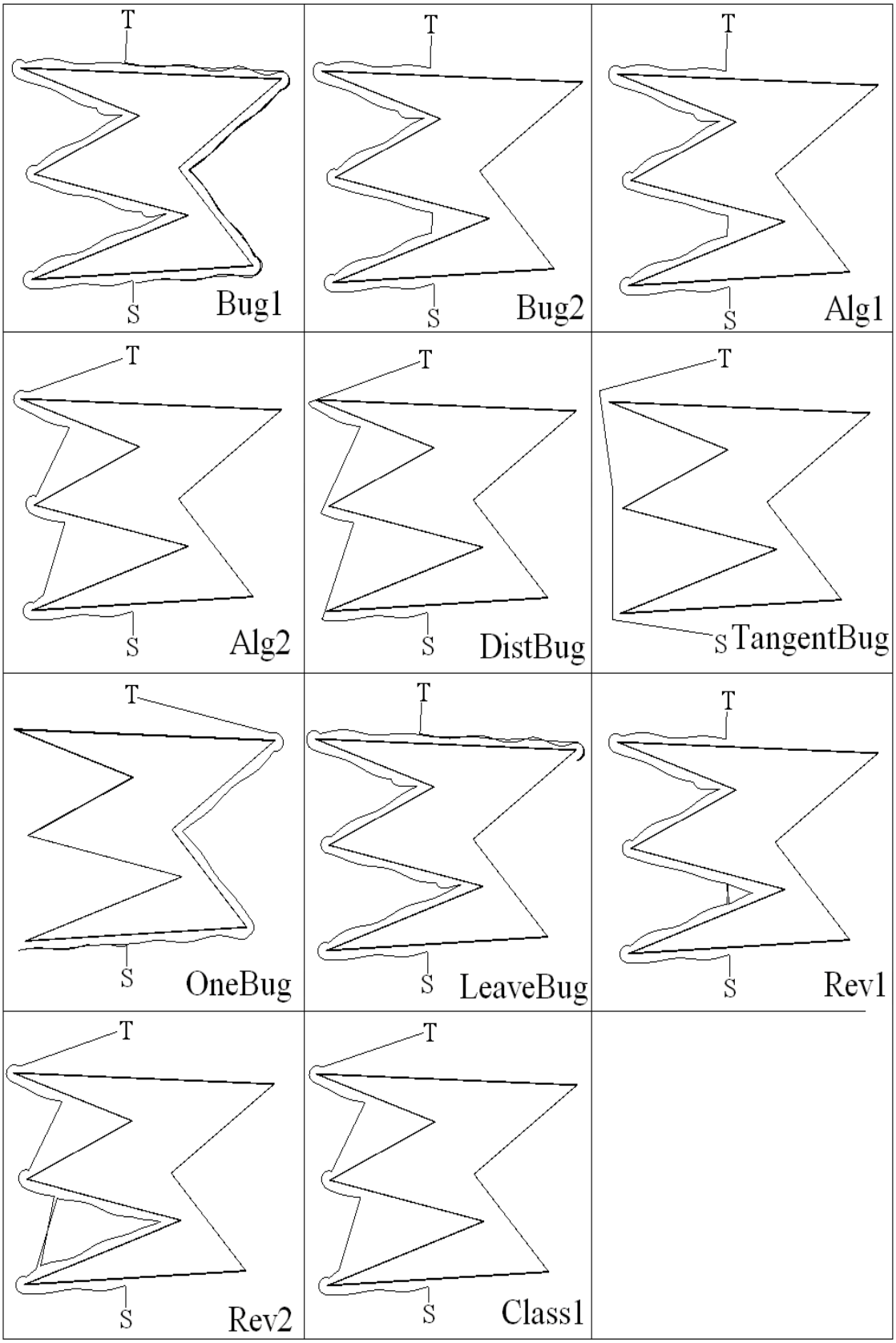

Figure 2-9: Paths for Bug algorithms in environment C

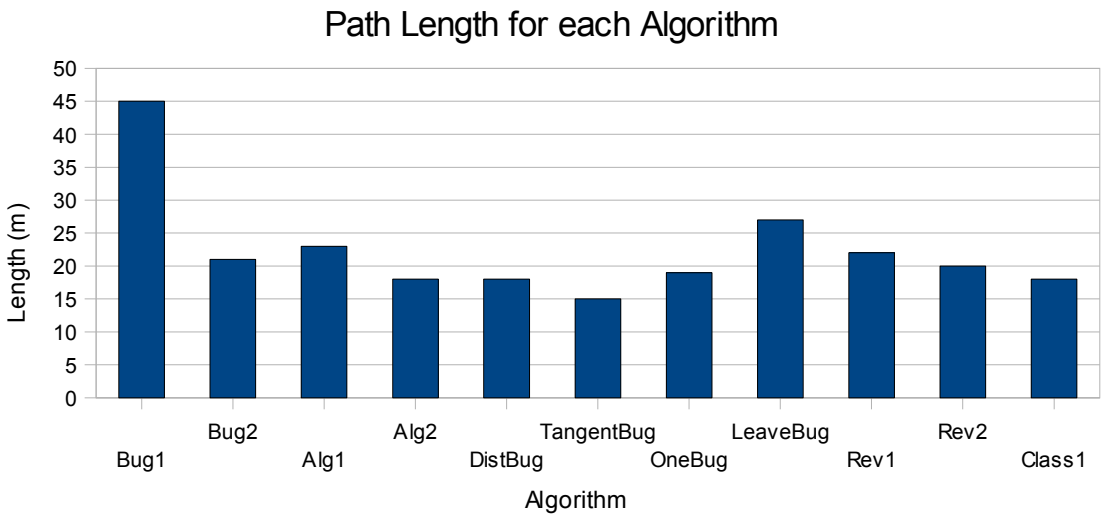

Figure 2-10: Path lengths for environment C

Some interesting questions arise as to whether particular algorithms will always perform better than others on semi-convex obstacles. For instance, it is foreseeable that TangentBug will always produce the shortest path since it always travels on the convex hull. Also, does DistBug always produce the next shortest path because of its range-based leaving condition? (Figure 2-10)

Figure 2-11 illustrates the Bug algorithms on an environment where the target is unreachable (here the target is inside the obstacle). An unreachable target implies that at least one obstacle must be fully circumnavigated. TangentBug produces the shortest path because its range sensors allow it to scan along the surface of the obstacle without the robot needing to actually travel there. Amongst the tactile sensor algorithms, Bug1 and LeaveBug are tied for shortest path because Bug1 requires the entire obstacle to be circumnavigated before leaving and LeaveBug requires the entire enabling segment to be explored. On this particular environment LeaveBug's path is identical to Bug1. As expected, stored points, alternative wall-following methods and other path-shortening measures are useless on an obstacle which renders the target unreachable.

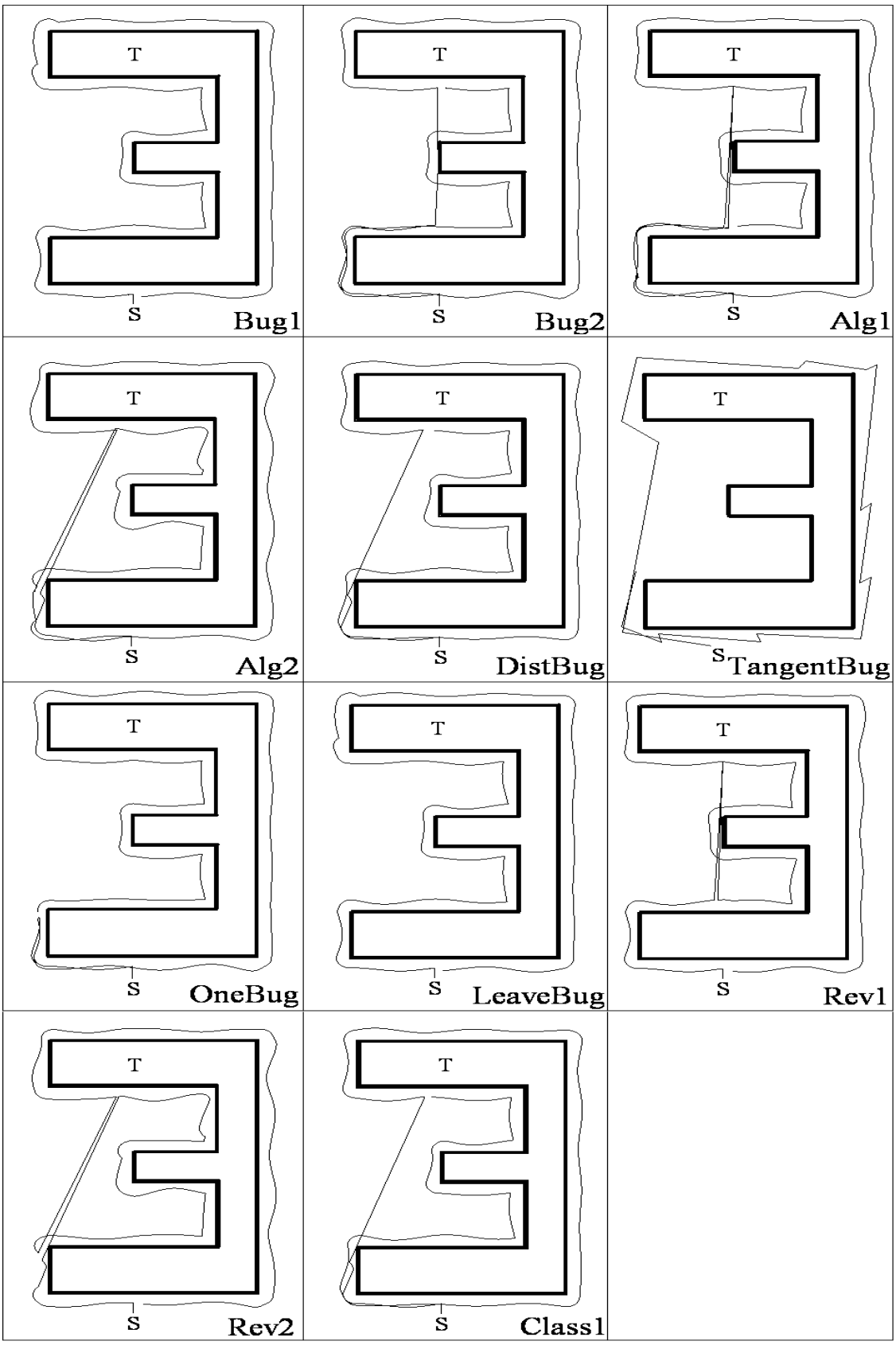

Figure 2-11: Paths for Bug Algorithms in environment D

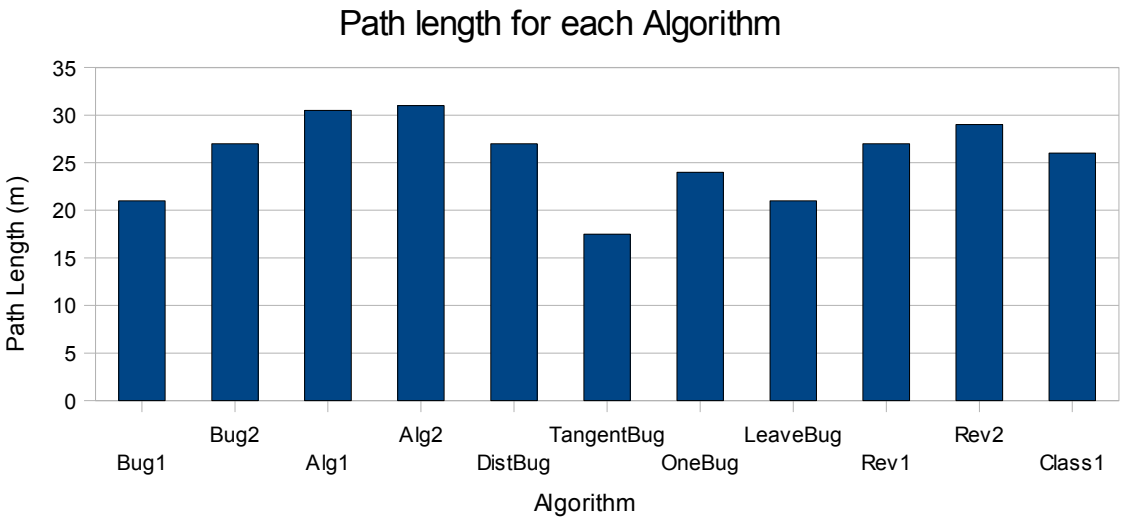

Figure 2-12: Path lengths for environment D

Algorithms with the fewest states were the easiest to implement, i.e. simple leaving condition and storing only the current hit point for circumnavigation detection. These include Bug2 and Class1, which require the least amount of code for implementation. They did not fail in any of the test environments and used the least amount of memory.

Then, there are slightly harder to implement algorithms such as Bug1, which requires Q_m to be continuously updated as well as information on whether or not the robot can drive towards the target. OneBug and LeaveBug require several states to operate and DistBug requires a range-based leaving condition. Neither of these algorithms failed in any of the test environments, but they required slightly more memory and some more lines of code for implementation.

Algorithms that require stored points are more difficult to implement. Alg1, Alg2, Rev1 and Rev2 all require management of multiple stored points and therefore also more memory. Generally, these algorithms are reliable but a few failures were recorded due to the robot falsely classifying

a protruding obstacle point as a previously encountered stored point. Also, additional debugging and testing was needed for the stored points scheme to ensure correct functionality.

Finally, TangentBug was the most difficult to implement. The requirements of detecting local minima, "corner smoothing" with finite angular resolution and processing nodes were quite complicated in comparison to other Bug algorithms. These requirements also required a lot of computation resources and were a frequent source of failure. Also, the robot has to drive slightly away from the focus node to avoid a collision. One advantage of TangentBug is that there is no need for wall-following since the robot only turns on the spot and travels in straight lines. TangentBug shows what can be theoretically achieved using an omni-direction, always up-to-date LTG.

2.5 Results Achieved by other Researchers

This comparison of the Bug algorithms was the first which compared 11 algorithms. Since then there have been other comparisons by other researchers in the area. Yufka and Parlaktuna [28] used the MobileSim [29] simulation system to simulate Bug1, Bug2 and DistBug. Their findings concur with the author's results. According to them:

"The results show that Bug1 is the worst one whose path length is approximately 3.37 times longer than the path length of the Bird-eye's view and DistBug is the best one whose path length 1.49 times longer than the path length of the Bird-eye's view."

Chiang, Liu and Chou [32] compared Bug1 and Bug2 against the Fast

Marching Method [33, 93] and their refined version called Boundary Following Fast Marching Method [34]. The Fast Marching Method relies on a completely known environment and hence can produce the shortest path. However, the Boundary Following Fast Marching Method partially explores the environment before plotting a path to the target. Their findings concur with the author's results for Bug1 and Bug2. It was also noted that the BFFMM produces a path length between Bug1 and Bug2. According to them:

"BFFMM builds the partial map by contouring the boundary curves of obstacles, so the path length in exploring mode would approach the limit of Bug1.The total path length of BFFMM is close to that of Bug1, but the path length of the path planning phase is even shorter than that of Bug2."

Noborio, Maeda and Urakawa compared Class1, Bug2, Alg1, Alg2 and HD-I on two large mazes [31] with four random start and target points. These mazes are quite large and do not resemble any "normal" environment. Not considering Class1, the results show that Bug2 always produces a longer path than Alg1, Alg2 generally outperforms Alg1 and HD-I almost always outperforms Alg2. This concurs with the author's findings concerning Bug2, Alg1 and Alg2. HD-I is able to perform well because it "learns" which direction is likely to produce short paths based on previous encounters and the direction of the target. However, these results have only been shown to work on large mazes.

2.6 Summary of Results

Table 2-1 summarizes the algorithms against path length and different criteria enumerated through sections 2.3.1 to 2.3.7.

Algorithm Name	Total Path Length (m)	Wall Following Algorithm	Sensors	Check if robot can move to T	LTG	M-line detection required?
Bug1	107	PD	Tactile	Yes	No	No
Bug2	74.3	PD	Tactile	Yes	No	Yes
Alg1	83.3	PD	Tactile	Yes	No	Yes
Alg2	76	PD	Tactile	Yes	No	No
DistBug	64.5	PD	Infrared	No	No	No
TangentBug	46.5	N/A	Infrared	No	Yes	No
OneBug	68.5	PD	Tactile	No	No	No
LeaveBug	65	PD	Tactile	No	No	No
Rev1	64.8	PD	Tactile	Yes	No	Yes
Rev2	62	PD	Tactile	Yes	No	No
Class1	67.5	PD	Tactile	No	No	No

Table 2-1 Summary of the Bug algorithms. Note that Total path length is the sum of path lengths over 4 Environments.

Chapter 3

An Analysis of Bug Algorithm Termination

3.1 Introduction

In this chapter, similarities and differences in the Bug Algorithms are explored particularly in relation to the methods used to guarantee termination. All the Bug algorithms are very similar fundamentally but differ in how they are modified to guarantee termination. The author has observed that all published Bug algorithms possess one of five distinct methods which allow it to guarantee termination. Two new methods are created and form the basis for the new Bug algorithms OneBug, LeaveBug, MultiBug and SensorBug.

The aim of examining the methods used to guarantee termination is in line with the goals of the thesis which is to reduce resources consumed in guaranteeing termination. In this case, the more specific aim is to determine which of the methods can be used to reduce path length or algorithm complexity whilst guaranteeing termination.

3.2 Bug Algorithm Analysis

There are two striking similarities which occur in all the examined algorithms. Firstly, all Bug algorithms use at least two modes of operation – "moving to target" and "boundary following". Secondly, the leaving rules almost always compare the potential leave point (this may be different from the robot's actual position in the case of range sensors) with the closest point ever visited (or scanned by range sensors) by the robot or they

seek to prohibit the robot from leaving at points it has left from before. It should be asked if these are fundamental and essential properties for any algorithm which attempts to solve the Bug problem or if there is another way.

The "moving to target" mode where the robot drives directly to the target is the only logical course of action given that the robot knows nothing of the environment. The path length of these segments must be finite since the distance from any point on the 2D plane to target is finite. However, obstacles may be encountered during "moving to target" and hence the necessity for "boundary following" mode.

The maximum path length of any "boundary following" segment is the perimeter of the obstacle which is being followed. Once the robot has fully followed that perimeter and its leaving condition has not held, then it should conclude that the target is unreachable and terminate since it is only going to get old data by more circumnavigation. Given that the lengths of both the "moving to target" and "boundary following" modes are finite then the only thing which could cause path length to go to infinite is if there is an infinite amount of "moving to target" and "boundary following" segments. When the robot is "moving to target", the transition to "boundary following" mode is straightforward. Hence, the only thing which can cause infinite length is the leaving condition and thus their importance in the Bug algorithms.

Leaving rules are best explained by Sankar [4] in his development of Alg2. A brief synopsis is given here. For termination to be guaranteed, the set of obstacles which the robot can possibly encounter in its journey from S to T must be finite and never allow new obstacles to be added. It is necessary to

ensure that the set is finite and non-admitting because if this does not hold then the robot can encounter an infinite number of obstacles and never terminate. An example of such a set illustrated in Figure 3-1. As the robot performs "moving to target" the radius shrinks and no new obstacles can join. As the robot performs "boundary following" it must ensure that no new obstacles can join when it leaves – hence the necessity to leave only when the robot is closer to the target than any point previously visited P.

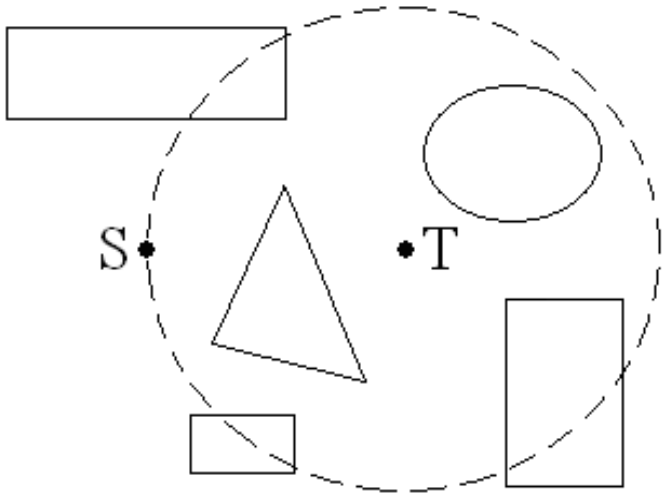

Figure 3-1: Number of obstacles inside the disc is finite.

To satisfy this requirement, the only method is to compare d(x,T) against d(x,T). If d(x,T) < d(P,T), then the robot can leave and the set remains non-admitting. This effectively checks if the robot is inside the disc centered at T with radius d(P,T). Clearly, there is no other enclosure which can satisfy this requirement. Any enclosure which is larger may allow the robot to admit obstacles into the set. Any enclosure which is smaller may result in the robot circumnavigating an obstacle and incorrectly concluding that the target is unreachable.

The algorithms Class1 and Com (which are not a Bug algorithms but something used in the development of workable Bug algorithms [4]) were

used to illustrate the above arguments. Some interesting properties emerge from the Class1 algorithm:

Theorem 1: *A robot can never leave twice from the same position.*

Proof: The robot updates P to X and if X is revisited then d(X,T) < d(P,T) is false. ■

Theorem 2: *A robot can never register a hit point an obstacle along a previously visited boundary segment.*

Proof: The robot leaves at a point that is closer to the target than any previously visited. During "moving to target" that distance monotonically decreases. If an obstacle is encountered then the hit point will be closer to the target than any point previously visited and therefore cannot be along a previously visited boundary segment. ■

Class1 has the disadvantage that the path length can be made arbitrarily long. Consider Class1 in the environment depicted on Figure 3-2. Decreasing W can make the path length arbitrarily long. Fundamentally, the problem is that there are an infinite number of points within the "shrinking-disc" and an environment can be constructed such that the robot always leaves at a point which only very marginally shrinks the disc.

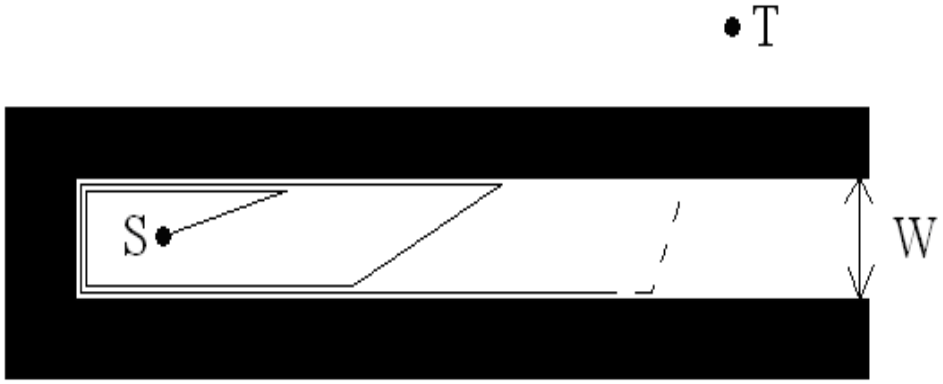

Figure 3-2: Decreasing W produces an arbitrarily long path under Class1.

3.3 The Methods

To overcome this problem, the "shrinking-disc" needs to be supplemented with another rule that guarantees the number of leave points is finite. The author has observed five methods are currently used to solve this problem.

3.3.1 The Closest Points Method

Each obstacle has a minimum distance to the target and all points which satisfy that are in set C. The robot can only leave at points in C.

This method works by noting that if the robot leaves at a point in C, the robot can never encounter that obstacle again in the remainder of its journey. Hence, there can only be at most one leave point associated with all obstacles in the finite set of obstacles intersecting the "shrinking-disc". The downside is that to identify C for a particular obstacle O, the robot must fully circumnavigate O and this can lead to unnecessarily long path lengths. Bug1 is the only algorithm to utilize the closest points method.

3.3.2 The M-line Method

Create a line from S to T called the M-line. The robot leaves on the M-line.

This method creates a line within the "shrinking-disc" since all points on the M-line are also points with the "shrinking-disc". The intersection of the line with an obstacle produces a distinct and unique point. Given the assumption that there can only be a finite number of intersections with the M-line, then there can only be a finite number of leave points in the entire journey. The downside is that many cycles can be created if there are many intersections with the M-line. VisBug-21, VisBug-22, Bug2, Alg1, Rev1, HD-1/w, Ave and ABUG use the M-line method.

3.3.3 The Disabling Segments Method

Keep the number of hit points that occur along disabling segments finite.

For any given obstacle, it has been shown [6] that there are a finite and non-zero number of disabling segments. Similarly, there will be a finite number, possibly zero, of enabling segments. A disabling segment occurs when the robot cannot travel towards the target for all points in the segment. An enabling segment occurs when the robot can travel towards the target for all points in the segment. Figure 3-3 top illustrates disabling segments and Figure 3-3 bottom illustrates enabling segments. The robot can only encounter an obstacle on a disabling segment and if these encounters can be restricted using Theorem 2 then the number of hit points can be kept finite.

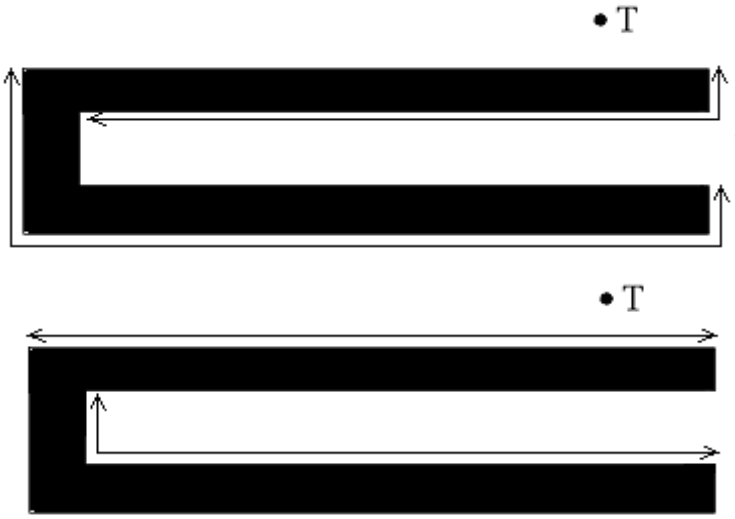

Figure 3-3 Top: Disabling segments. Bottom: Enabling segments

There are many different ways to implement this method. For example, Alg2 implements it by forcing the robot to return to the last defined hit point and exploring the remaining counter-clockwise wall following direction when a previously stored point is met. Similarly, Rev2 forces the robot to return to the closest hit point on H-list and to explore the alternate

direction. These rules ensure that there can only be a finite number of hit points per disabling segment.

Obviously, there can be many other ways of implementing this method. Consider a new and simplistic Bug algorithm, OneBug, in which the robot must explore the entire disabling segment before it can begin the exploration of a neighboring enabling segment. Figure 3-4 shows the pseudo-code of OneBug:

1) Drive directly to the target until one of the following occurs:
    a) Target is reached. OneBug stops.
    b) An obstacle is encountered. Go to step 2.
2) Perform clockwise circumnavigation until one of the following occurs:
    a) Target is reached. OneBug stops.
    b) The robot is able to drive towards the target. Go to step 3.
    c) The robot completes circumnavigation around the blocking obstacle. The target is unreachable and OneBug stops.
3) Perform counter-clockwise circumnavigation until one of the following occurs:
    a) Target is reached. OneBug stops.
    b) The robot is at a point which is closer to the target than any previously visited and it is able to drive towards the target. Go to step 1.
    c) The robot completes circumnavigation around the blocking obstacle. The target is unreachable and OneBug stops.

Figure 3-4 Pseudo code of OneBug

OneBug is so named because there can be at the most one hit point per disabling segment. OneBug can be described as Alg2 without the stored

points and Class1 without the arbitrarily long path length. Figure 3-5 shows OneBug's simulation [7] results on two environments.

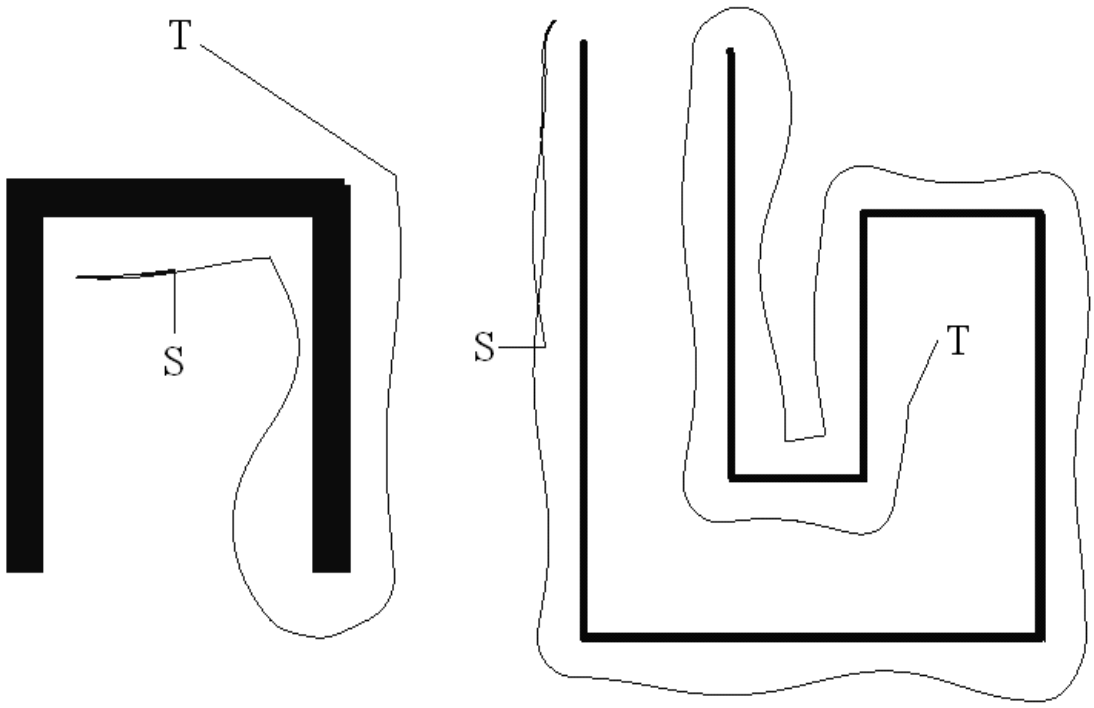

Figure 3-5: OneBug algorithm in two different environments

The stored points used to implement this method on Alg2 also serve another purpose, to reduce path length. However, chapter 2 illustrated a situation where the path length of Alg2 was greater than DistBug despite Alg2 using stored points (Figure 2-7).

The reason for this is that Alg2 does not allow the robot to resume searching from the pathwise closest unexplored region but chooses the last defined hit point. Now consider a new Bug algorithm MultiBug which allows the robot to choose clockwise or counter-clockwise circumnavigation to explore the closest unexplored disabling region. MultiBug is described in Figure 3-6.

The number of hit points per disabling segment is kept finite because upon returning to a previously stored point H in step 3, the robot must choose to follow the wall in a counter clockwise direction. This necessarily means that the disabling segment associated with H must be fully explored using at most two encounters. Figure 3-7 illustrates MultiBug.

1) Drive directly to the target until one of the following occurs:
    a) Target is reached. MultiBug stops.
    b) An obstacle is encountered. Record a hit point H_i. Go to step 2.
2) Perform clockwise circumnavigation until one of the following occurs:
    a) The robot is at a point closer to the target than any previously visited and is able to move towards the target. Go to step 1.
    b) The robot encounters a previously recorded hit point. Go to step 3.
    c) The robot encounters H_i. Terminate with failure.
3) Decide upon which wall following direction would take the robot to the nearest hit point H which only has the clockwise direction explored. Return to H. At H, perform counterclockwise wall following until one of the following occurs:
    a) The robot is at a point closer to the target than any previously visited and is able to move towards the target. Go to step 1.
    b) The robot returns to H. Terminate with failure.

Figure 3-6 Pseudo code for MultiBug

3.3.4 The Step Method

Leave only after the robot is a predefined distance, STEP, inside the "shrinking-disc".

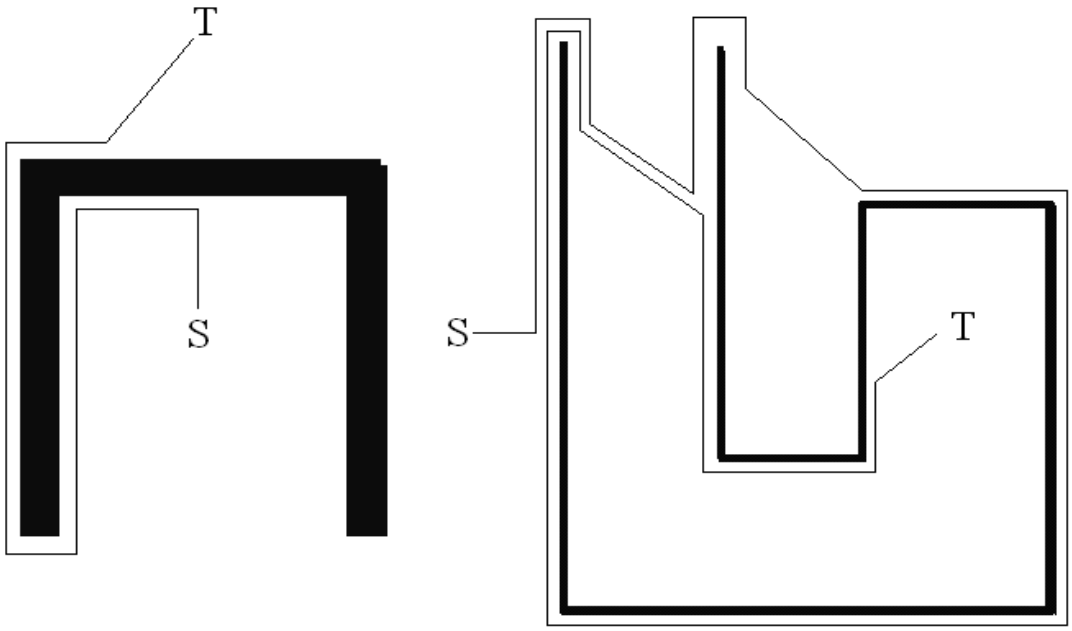

Figure 3-7 MultiBug algorithm in two different environments

Using this method, the robot can leave at most $\lfloor d(S,T)/STEP \rfloor$ times in its entire journey. Unfortunately, this method requires knowledge of the environment since STEP must be chosen such that during circumnavigation, a point on every encountered obstacle will fall in an enabling segment that is STEP closer to the target. If this doesn't hold then it would be possible for the robot to circumnavigate an obstacle and incorrectly conclude that the target was unreachable. The step method is used in DistBug for both tactile and range sensors.

3.3.5 The Local Minimum Method

Restrict hit points to local minimums that occur on encountered obstacles.

Using the local minimum method, the robot can only define hit points if it detects that it is in a local minimum on the encountered obstacle with respect to the target. It has been shown [6] that the number of local

minimums on any obstacle is finite. Figure 3-8 shows the local minimums on two environments with respect to the target.

TangentBug, CautiousBug, RoverBug, 3DBug and WedgeBug all use the local minimums method.

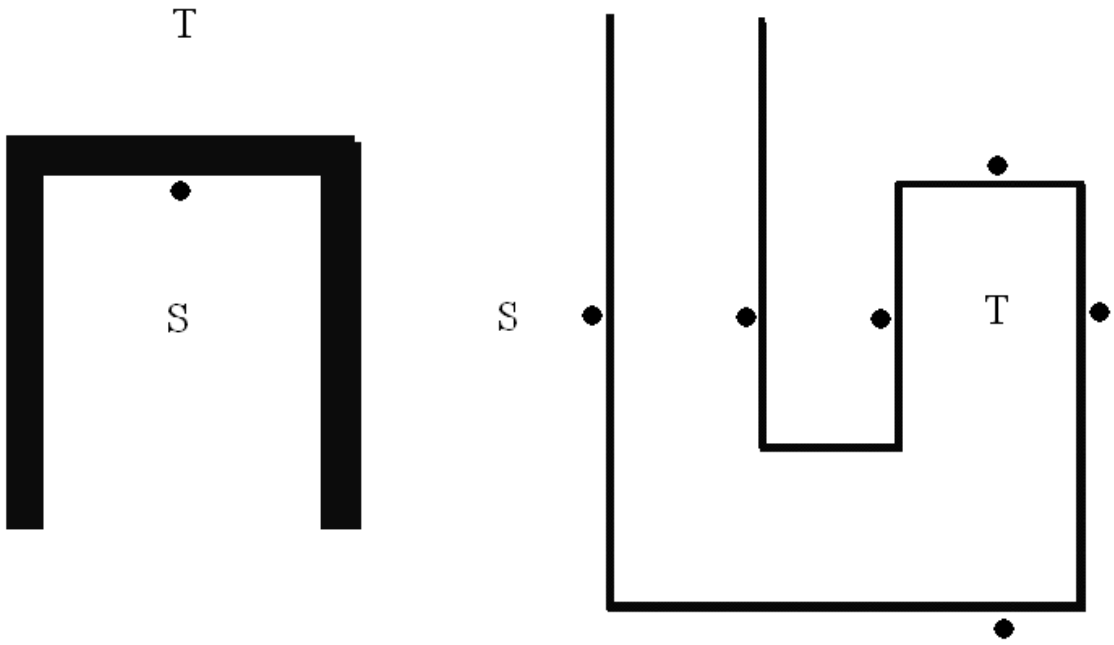

Figure 3-8: Two environments with local minima

3.3.6 The Enabling Segments Method

Keep the number of leave points that can occur along enabling segments an obstacle finite.

The enabling segments method uses the fact that there are only a finite number of leaving segments for any given obstacle. Consider a new Bug algorithm LeaveBug which implements the enabling segments method. LeaveBug pseudo code is illustrated in Figure 3-9. Examples of LeaveBug are presented in Figure 3-10.

1) Drive directly to the target until one of the following occurs.

a) Target is reached. LeaveBug stops.

b) An obstacle is encountered. Go to step 2.

2) Perform clockwise circumnavigation until one of the following occurs:

a) Target is reached. LeaveBug stops.

b) The robot is able to move towards the target. Go to step 3.

c) The robot completes circumnavigation around the obstacle. The target is unreachable and LeaveBug stops.

3) Perform clockwise circumnavigation whilst updating a point P to X if the robot is currently closer to the target than any point previously visited. Do this until one of the following occurs:

a) Target is reached. LeaveBug stops.

b) The robot is unable to move towards the target. If P contains a position, go to step 4. Else, go to step 2.

c) The robot completes circumnavigation around the obstacle. The target is unreachable and LeaveBug stops.

4) If $d_{path}(X,P)$ is 0, reset P to null and go to step 2. Else, perform counter-clockwise circumnavigation until the robot returns to P. At P, reset P to null and go to step 1.

Figure 3-9 Pseudo code for LeaveBug

The T^2 algorithm [41] utilizes the enabling segments method by permitting only two leave points per enabling segment. The first one is at the transition from a disabling to enabling segment and the second one is from the current M-line. There are only a finite amount of type one leaving points and the robot is only permitted to leave at each of them once. Therefore, there can only be a finite number of M-line redefinitions and each of the redefinitions can only pass through an enabling segment once.

Hence, the number of leave points is finite. The robot will always find the target, if reachable, because even in the absence of type one leaving points it will always have an M-line and that has been shown to guarantee termination previously.

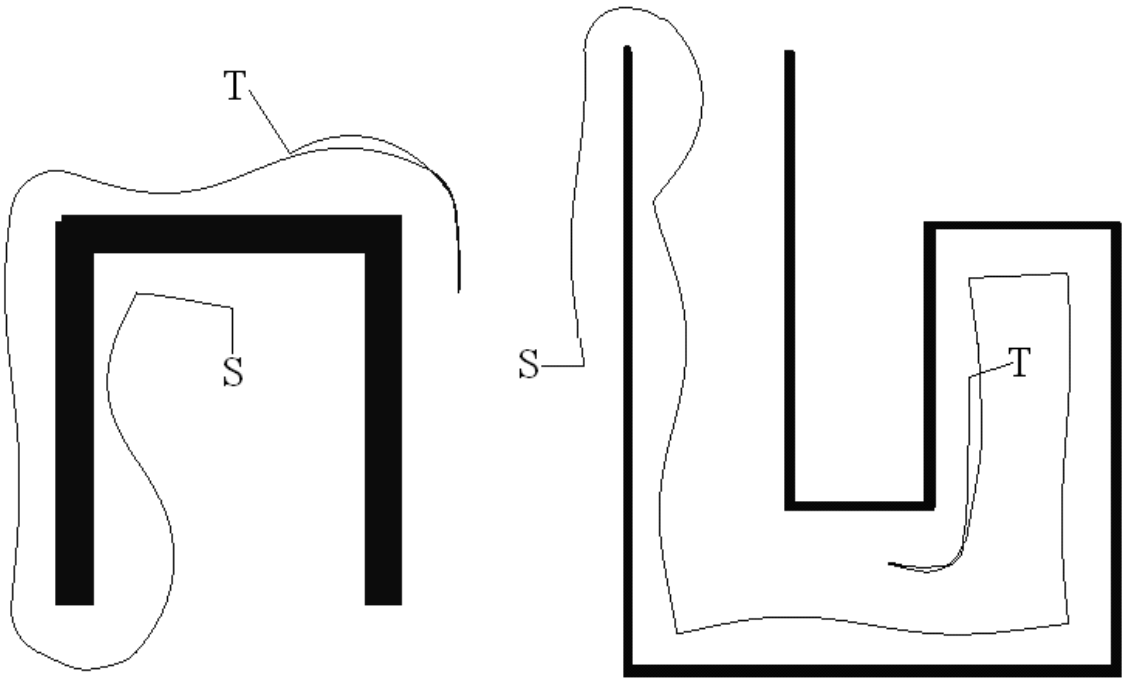

Figure 3-10: LeaveBug algorithm in two different environments

Angulus [13] allows the robot to leave when "it is physically possible to walk straight toward T, the current velocity vector v points towards T and both "a" (the robot's position vector with respect to the ST line) and "b" (the robot's velocity vector with respect to the ST line) are in the range of 0 to 360 degrees, proceed straight to T". Such a position corresponds to a transition between disabling segments and enabling segments. Technically, the leave point would be on the edge of the enabling segment and the number of these points is finite. The second leaving condition of Angulus where "if a = b = -360 leave towards T" is vague because that can only occur in rare situations and does not fit any of the methods.

SensBug [71] and K-Bug [87] direct the robot to leave at positions where the robot can travel towards the target. This is in effect directing the robot to leave at transition points between disabling and enabling segments.

3.3.7 The Q Method

There is a unique set of points Q which is finite for any polygonal obstacle O. Q is the set of points at which the robot must stop to gather data to continue circumnavigating O both in the clockwise and counter-clockwise directions. Clockwise and counter-clockwise circumnavigation must begin from a vertex on O for the purposes of determining members of O.

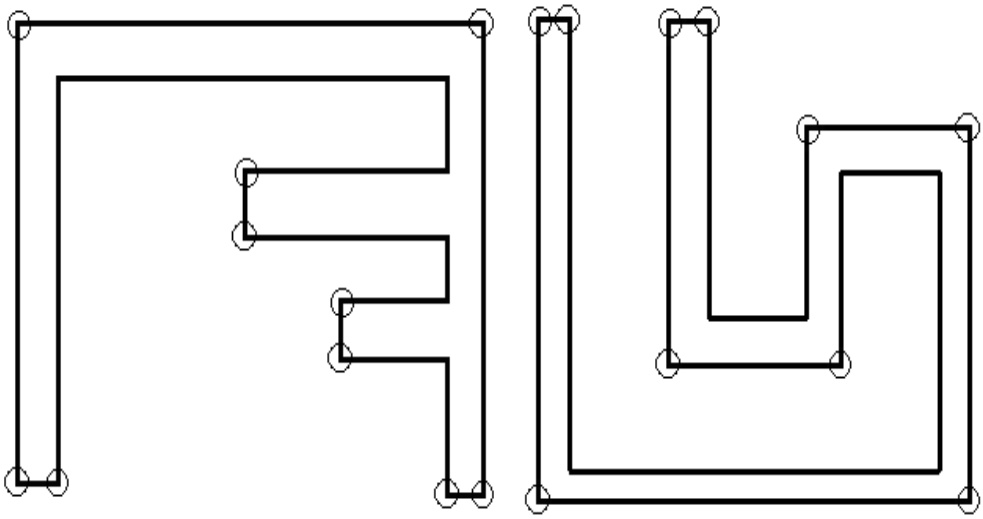

Figure 3-11: Points in Q are circled

In Figure 3-11, the circled points represent members of Q assuming a robot has infinite sensor range.

The Q method is to be used in the development of SensorBug in Chapter 6. Some of the issues which arise are how to implement it. For instance, how to ensure that the robot will always leave at a point in Q? What about the case where Q is empty, for instance if the robot starts enclosed in a box?

How to guarantee that the robot will always leave an obstacle which does not encircle the target?

The Q method has a significant drawback in that it requires a polygonal obstacle so that the number of vertices is finite. Otherwise, Q could be infinite on a curved section of O. Also, the robot must have range sensors. If tactile sensors are used then Q will be infinite.

There are two main foreseeable advantages of the Q method. Firstly, it should be easy to implement for robots using range sensors compared to TangentBug and VisBug. Secondly, the robot does not have to continuously poll the external environment whilst traveling between members of Q. Instead, the robot is free to devote computing resources to other tasks.

3.4 Other Methods to keep hit or leave points finite

There are infinite ways in which methods can be combined with each other to produce new Bug algorithms. Further, any new Bug algorithm can incorporate special features of other algorithms. For instance, it would be easy to add the reverse wall following procedure from Ave, or the cautious wall-following procedure in CautiousBug, or the LTG from TangentBug or the stored points concept from Alg1. But for any new Bug algorithm to be truly unique, it needs to find a new way of analyzing obstacles to determine the positions of leave or hit points in such a way that any obstacle will have a finite number. From there, a leaving condition can be created to implement the analysis. In other words, a new method needs to be created for any future Bug algorithm to be considered truly unique. The creation of new methods is beyond the scope of this thesis and is left for future research. However, the task is going to require much imagination and creativity.

3.5 Completely Exploring the Blocking Obstacle

When Lumelsky published Bug1 and Bug2 [1], Lucas commented on them shortly afterwards [58]. Lucas acknowledged that one of Bug1's purposes was to completely explore the blocking obstacle. However, Lucas recommended that Bug1 and Bug2 be combined so that the benefits of both algorithms can be used simultaneously to reduce path length whilst preserving the complete exploration of the blocking obstacle. He named his algorithm (2) and it is quoted in Figure 3-12.

Initially, j = 1; *Lo* = starting point.

1) From point L_{j-1} move toward the target along the M-line until one of the following occurs:

a) Target is reached. The procedure stops.

b) An obstacle is encountered and a hit point H_j is defined. Go to step 2.

2) Using the accepted local direction (left), follow the obstacle boundary. If the target is reached, stop. Use R1 to store the coordinates of the intersection with the M-line Q_m having the least distance from the target point, R2 to integrate the length of the boundary starting at H_j and R3 to integrate the length of the boundary starting at Q_m. (In case of many choices for Q_m take any.) After having traversed the whole boundary and having returned at H_j define a new leave point as $L_j = Q_m$. Go to step 3.

3) Using the contents of R2 and R3 determine the shorter way along the boundary to L_j and use it to go to L_j. If the straight line from L, to target crosses the obstacle, then the target cannot be reached. Otherwise, set j = j +1. Go to step 1.

Figure 3-12 Lucas's (2) algorithm which combines Bug1 and Bug2

Lumelsky's reply [58] was that this defeats the motivation behind Bug1 which was to allow the robot to deviate as far as one wishes from the desired path (the M-line). By this, the author assumes that Lumelsky means Bug1's purpose was to attempt to find other routes to the target and that this is necessary to fulfilling its higher level objectives. If this assumption is correct then Lumelsky is correct in rejecting the modification.

However, if the objective of the Bug algorithms is to minimize path length to the target then the modification holds weight. Consider the environment shown in Figure 3-13. In this environment, (2) clearly outperforms Bug1. Further, (2) has the advantage of reverting to the Bug1 leave point if the M-line leave point proves disadvantageous. Therefore, it will always have a shorter path than Bug1.

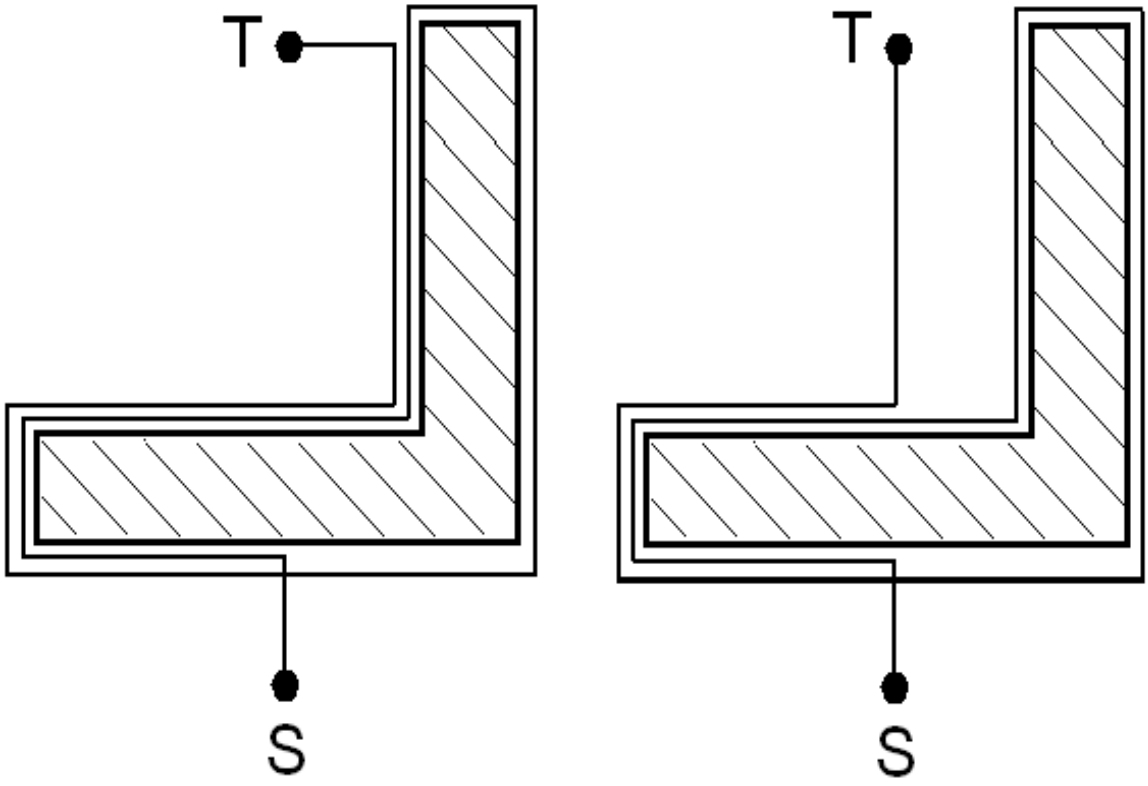

Figure 3-13 Left. The Bug1 Algorithm Right. Lucas's (2) algorithm

It is true that Bug1 can choose either clockwise or counter-clockwise boundary following when it returns to the original hit point depending on which requires less travel. However, the right side of the obstacle can be arbitrarily modified so that the counter-clockwise route requires extensive travel.

If the objective is to minimize path length whilst exploring all of the blocking obstacle then improvements can be made. Consider an algorithm, Bug1+, which directs the robot to return to either end of the enabling segment which contains Q_m depending on which required less travel. Once there, the robot leaves. Formally, Bug1+ can be written as shown in Figure 3-14. Bug1+ is shown in Figure 3-15 and it can clearly be seen that its path length is shorter than Bug1 and (2).

Furthermore, Bug1+ allows the robot to deviate from the M-line and find alternate routes to the target. Clearly, this satisfies Lumelsky's motivation for Bug1 and there should now be no reason for rejection.

Bug1+ raises an important question. Suppose that as soon as the robot encounters an obstacle that it somehow knows the entire geometry of the obstacle, when is it best for the robot to leave? Obviously, the Bug1+ algorithm does not cover all possibilities. For example, there may be other transition points which are not associated with Q_m. These other transition points require less travel and allow the robot to travel directly to the target.

Initially, j = 1; *Lo* = starting point.

1) From point L_{j-1} move toward the target along the M-line until one of the following occurs:

- Target is reached. The procedure stops.
- An obstacle is encountered and a hit point H_j is defined. Go to step 2.

2) Following the obstacle boundary in a clockwise manner. Record the following data until the robot returns to H_j. Go to step 3.

- A transition point t, where t is such that the robot transitions from a disabling to enabling segment or from an enabling to disabling segment. Record $d_{path}(t, H_j)$ for all such points.
- A point Q_m, where Q_m is a point which is closer to the target than any previously visited. Record $d_{path}(Q_m, H_j)$ and whether or not it is on a disabling segment.

3) When the robot returns to H_j, determine the following:

- If Q_m lies on a disabling segment, the target is unreachable. Stop.
- The two transition points associated with Q_m in the clockwise and counter-clockwise directions.
- The path lengths to travel to the two transition points from H_j. Go to step 4.

4) Following the boundary to the transition point which requires the least amount of travel. Leave at that point. Go to step 1.

Figure 3-14: The Bug1+ Algorithm

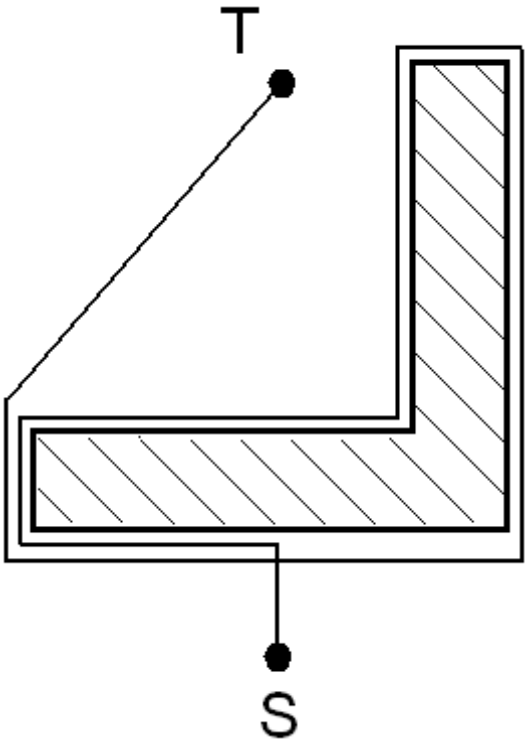

Figure 3-15: The Bug1+ Algorithm on an Environment

Figure 3-16 Left illustrates one environment as an example. Suppose that the robot completes circumnavigation and leaves at point A. If it does so, then it will produce a shorter path than at point B, since at point A it moves directly to the target without encountering the obstacle again. This would be the best result possible.

Secondly, consider Figure 3-16 right where leaving at A would cause the robot to re-encounter the obstacle. Clearly, this is not desirable since the algorithm would need to be more complicated to take the scenario into account. The ideal solution would be for the robot to travel from the original hit point, to A, directly to B and then directly to the target. However, designing an algorithm to do that will be challenging, but possible since the obstacle is completely known. This is left for future research.

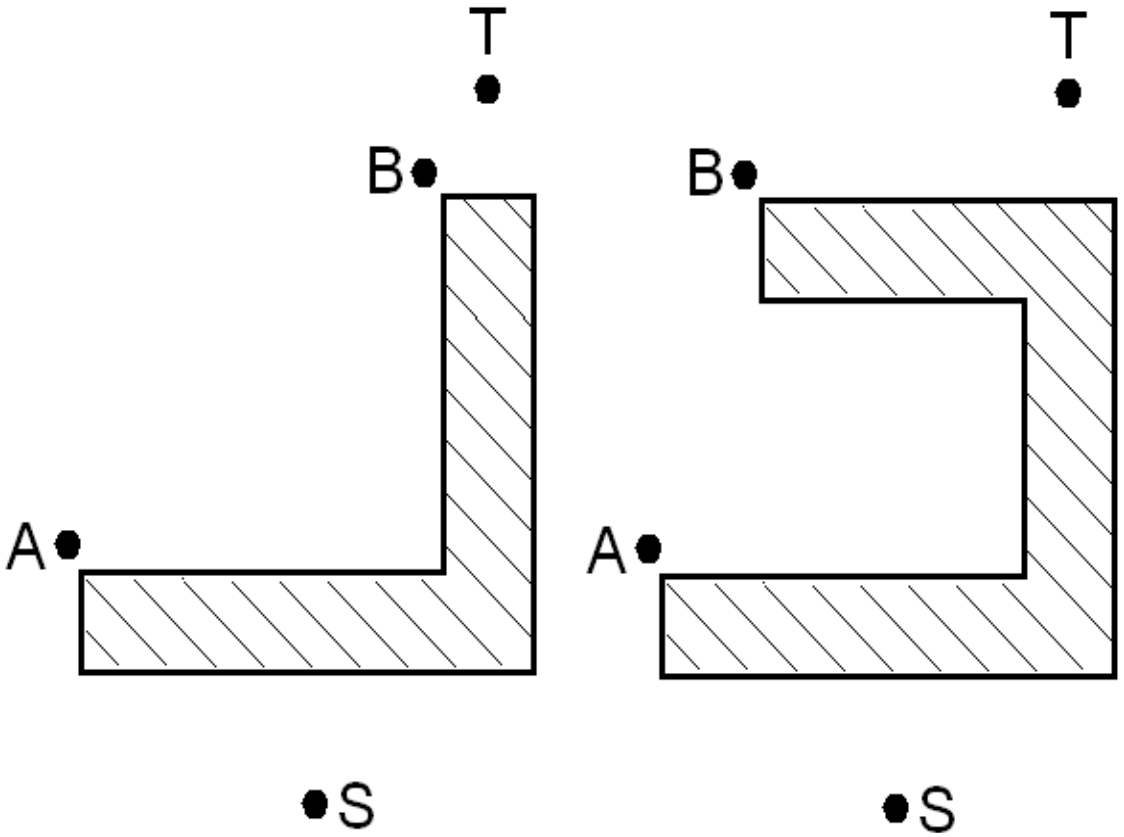

Figure 3-16 Left: Leaving at A produces the shortest path
Right: The optimal path is requires the robot to travel from A to B to T

A possible criticism that could be made against Bug1+ on Figure 3-15 is that once Bug1+ leaves the obstacle it has a longer path to target than Bug1. Since it has a longer path, it is more likely to encounter another obstacle and be forced to circumnavigate it thereby increasing path length. However, if the simplest and most logical assumption holds that there are no more obstacles then Bug1+ will always produce a shorter path than Bug1.

Chapter 4

Bug Algorithm Performance on Environments with a Single Semi-Convex Obstacle

4.1 Introduction

Lumelsky [1] noted two classes of environment in his original work. The *in-obstacle* class were those environments which had either S or T inside the convex hull of an obstacle. The *out-obstacle* class were those environments which were not *in-obstacle*. These were important because the performance of Bug2 was significantly better on out-obstacles since no local cycles could be made. In this chapter, this idea of obstacle classes is extended and investigated with more algorithms and a new semi-convex obstacle.

The performance of Bug algorithms Bug1, Bug2, Alg2, DistBug and TangentBug are investigated on environments which can only contain a single semi-convex obstacle with a reachable target and are *out-obstacle*. Semi-convex obstacles are a new class of obstacle defined in section 4.3. Proofs are developed based on the algorithm and obstacle properties and it is shown that for any such environment TangentBug will always produce the shortest path followed by DistBug, Alg2, Bug2 and then Bug1. Theoretical proofs are reinforced by simulations results.

Given that the environment is unknown prior to commencement it is obvious that path length minimizing navigation decisions will depend on luck. For instance, when the robot is driving straight towards the target and

encounters an obstacle, the decision to turn left or right is based on an arbitrary guess. However, if some restrictions are placed on the obstacles in the environment it will be shown that some algorithms will always outperform other algorithms. Firstly, the algorithms are compared on environments that can only contain one convex obstacle with a reachable target. Then, the results are generalized to environments that can only contain one semi-convex obstacle with a reachable target. Finally, simulations are conducted using EyeSim to reinforce theoretical findings.

4.2 Examined Bug Algorithms

Bug [1], Bug2 [1], Alg2 [4], DistBug [5] and TangentBug [6] are selected for examination. Other algorithms were not included in the examination mainly because they are fundamentally similar to each other with respect to their leaving rules. For example, CautiousBug is similar to TangentBug except that wall-following direction changes whilst following a given obstacle. Also, Rev1 is similar to Bug2 except that Rev1 introduces alternate wall-following and the reverse procedure. Alg1 is similar to Bug2 except that it uses multiple stored points. Rev2 is similar to Alg2 except that it has alternate wall-following and the reverse procedure. HD-1 is also similar to Alg2 except that that the robot can alter boundary following direction during a “boundary following” segment without needing to find a previous stored point.

Some algorithms were also implementations of other algorithms for example WedgeBug implements TangentBug except that it uses wedges and conglomerates them as necessary. RoverBug removes the assumption of a point robot and sensors which do not necessarily detect obstacles. But nothing changes in with respect to the leaving rule.

For effective comparison, restrictions must be placed on the wall-following direction. Otherwise, a particular algorithm could generate a short path by a lucky guess in respect to the wall-following direction. Therefore algorithms must follow Bug convention and perform clockwise boundary following upon defining a hit point. Changing wall-following direction during "boundary following" is also prohibited. In practical terms, this means TangentBug must choose left as its boundary following direction and Alg2 cannot use stored points to reverse its boundary following direction upon meeting a previously stored point. As will be shown later, stored points are not of any benefit on semi-convex obstacles in any case.

4.3 Performance on a Single Semi-Convex Obstacle

Consider an environment which features only a single convex obstacle O. O lies on the M-line and T is reachable. Let M be the point where the M-line intersects O closer to S, M' the point where the M-line intersects O closer to T, V_T the point on O which is tangent to T and V_S the point on O which is tangent to S. Figure 4-1 illustrates these points.

Theorem 1: *Given an encounter with a convex obstacle O, once leaving occurs the robot can never reencounter O.*

Proof: If the robot left O at point P_1 and reencountered O at point P_2, then P_1 and P_2 are points on O and there exists a path in freespace between P_1 and P_2. If that path exists, then by definition O cannot be convex. ■

Theorem 2: *If a robot is following O, an algorithm that leaves earlier on O will produce a shorter path than an algorithm that leaves later.*

Proof: Traveling directly to T is always shorter than an indirect path along V_T to M' and then directly to T. ■

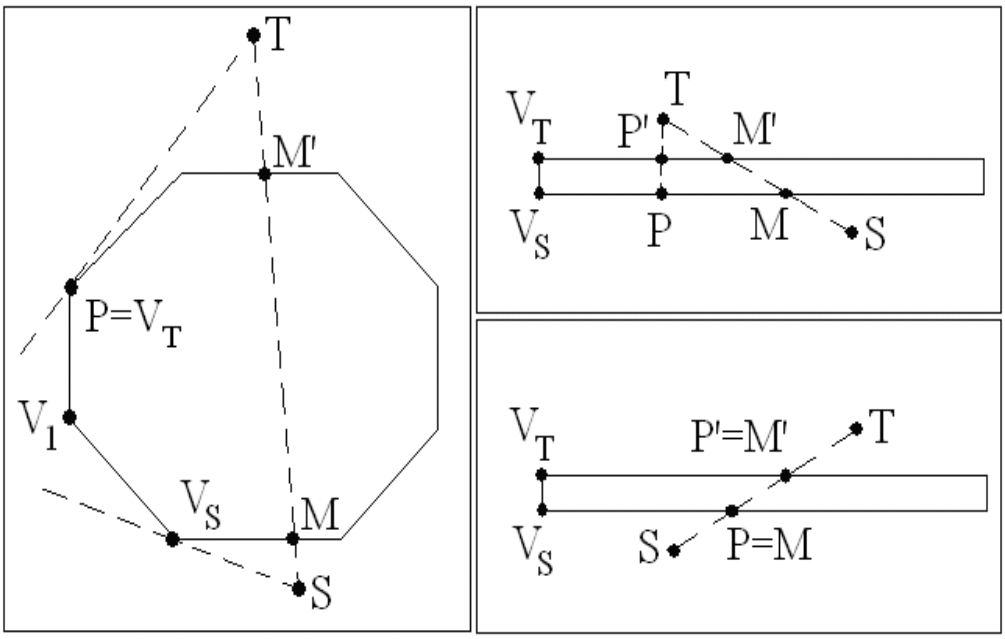

Figure 4-1 Left: P=VT. Top Right: P is in the segment M to VT. Bottom Right: P=M

Theorem 3: *Any point P from M to V_T when connected to T via the direct line segment [P,T] will always intersect the segment from V_T to M' at a point P' such that d(P,T) > d(P',T).*

Proof: The points M, V_T and T can be thought to form an infinite sector K centered on T with the tangents to V_T and M forming the two straight lines. O necessarily connects M and V_T and in that segment any point P will form a line segment [P,T] which is within K. Within K, there also exists another connection from V_T to M' which does not intersect the connection from V_T to M. Since d(M,T) > d(M',T), it follows that all P from V_T to M necessarily intersect the connection from V_T to M' at P' such that d(P,T) > d(P',T). Figure 4-2 illustrates this concept. ■

Theorem 4: *Assuming infinite sensor range, the algorithm that always produces the shortest path for an environment with reachable T and a single convex obstacle is TangentBug, followed by DistBug, Alg2, Bug2 and finally Bug1.*

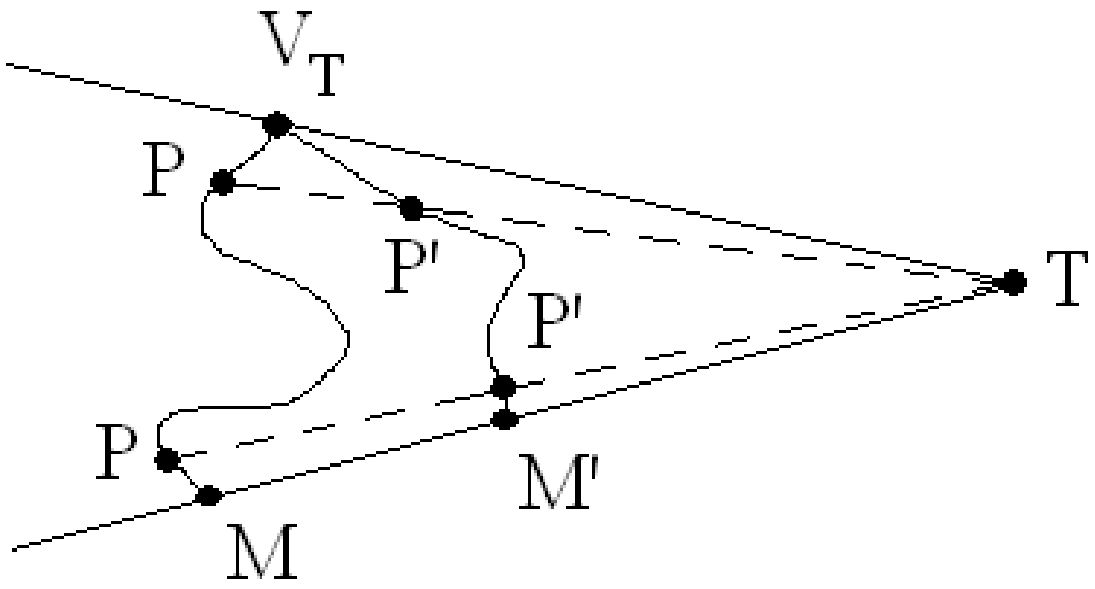

Figure 4-2: Any point P along M to V_T have a corresponding point P' along V_T to M' such that d(P,T) > d(P',T).

Proof: Consider the optimal path associated with traveling to the left of O. The optimal path starts from S to V_S. From V_S the optimal path passes through each successive vertex $V_1 \ldots V_N$ until it reaches V_T. From V_T the optimal path goes directly to T. Figure 4-1 left illustrates the optimal path where the optimal path is S, V_S, V_1, V_T, T. Figure 4-1 top right and bottom right both have optimal paths of S, V_S, V_T, T.

TangentBug can use the LTG to detect V_S allowing it to drive directly to it. Once V_S is reached, the robot uses the drives to each successive vertex and when it reaches V_T, it leaves because T is visible. Therefore, TangentBug always generates the optimal path.

DistBug can detect when the robot is at V_T since it continuously updates F. Also, it follows O's boundary from $V_1 \ldots V_N$ like all the other algorithms. However, from S to V_S, the robot does not necessarily follow the optimal path because it drives straight towards the target and then follows the

boundary to V_S. This means that TangentBug will always produce a shorter or equal length path to DistBug.

Next, consider Alg2 and DistBug. With no PSD sensor, Figure 4-1 left shows that it is still possible for Alg2 to leave at V_T but Figure 4-1 top right and bottom right show that it is also possible that the robot will continue following O and this produces a longer path than DistBug according to Theorem 2.

Next, consider Alg2 against Bug2. There exists a point P where P is the closest point to the target in the segment M to V_T. Hence, there are three possibilities: $P=V_T$, P is inside M to V_T or P=M.

Figure 4-1 left shows the case where $P=V_T$. When this occurs, the robot leaves at V_T and will always produce a shorter path than Bug2 according to Theorem 2.

Figure 4-1 top right illustrates the case where P is in the segment M to V_T. Theorem 3 shows that there will always exist P' inside V_T to M' such that $d(P,T) > d(P',T)$. Under Alg2, the robot will always leave at or before P' since $d(P,T) > d(P',T)$. Since P' is necessarily visited before M', Alg2 always produces a shorter path than Bug2.

The final case is demonstrated in Figure 4-1 bottom right where the P=M. In that case, the robot will always leave at or before M' since $d(P,T) > d(P',T)$ and the robot can travel directly to T from M'. If the robot leaves at M', the Alg2's path length will be the same as Bug2. Hence, Alg2's path length at very best is equal to DistBug and at very worst is equal to Bug2.

Finally, Bug1 will always produce the longest path length because it requires the robot to circumnavigate the obstacle first. ■

Placing a restriction that the obstacles must be convex is restrictive. Fortunately, there is another class of obstacle which is more general than a convex obstacle for which Theorem 4 holds. Consider the convex hull for any obstacle O. Any discrepancies between O and its convex hull are assigned an area A_X for X = 1, 2, 3.... If all A_X are convex, then the obstacle is said to be semi-convex. Figure 4-3 left illustrates a semi-convex obstacle because A_1 and A_2 are convex, but Figure 4-3 right is not a semi-convex obstacle because A_2 is non-convex.

For each area A_X, there are two vertices where the A_X, O and O's convex hull intersect. These vertices are labeled V_X and V_X' such that V_X will always be visited before V_X' for a robot which circumnavigates O in a clockwise manner. Figure 4-3 left shows V_X and V_X' for the semi-convex obstacle.

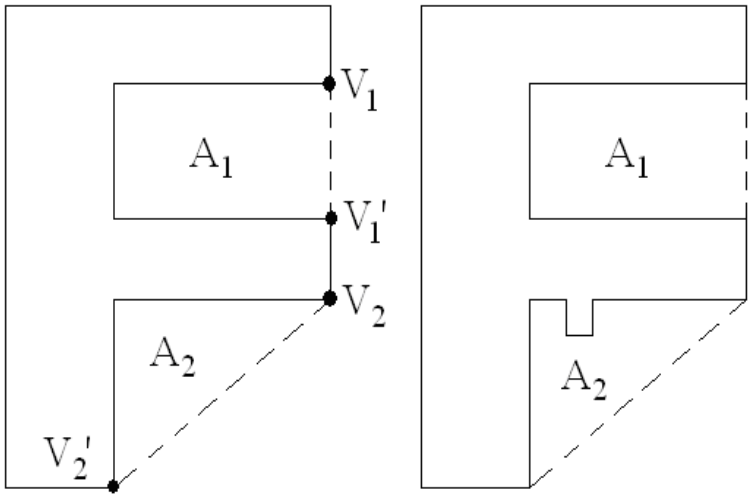

Figure 4-3 Left: A semi-convex obstacle. Right: A non semi-convex obstacle due to A_2

Each A_X is partitioned into segments according to whether or not the robot can drive towards T. If the robot is able, label that an enabling segment, otherwise label it a disabling segment. Let Q_X represent the beginning of the disabling segment in the clockwise direction and Q_X' the end of the disabling segment for a particular A_X. Let G_X be the enabling segment from V_X to Q_X, G_X' represent the disabling segment from Q_X to Q_X', and G_X" the enabling segment from Q_X' to V_X'. Figure 4-4 left illustrates these notations for A_1. Note that G_X, and G_X" do not necessarily have to exist as illustrated by A_2.

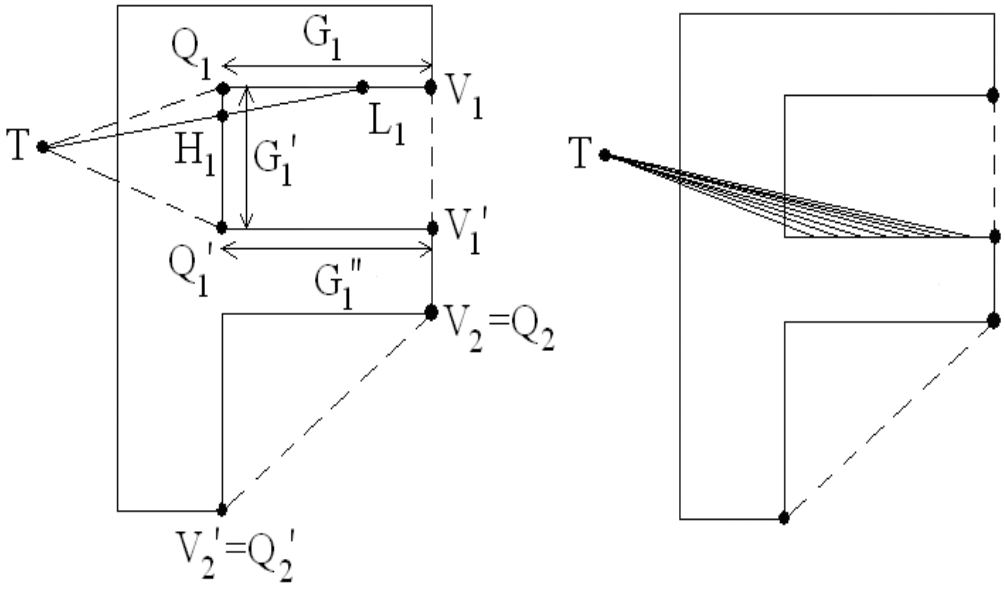

Figure 4-4 Left: Notations for semi-convex obstacles
Right: Any point along G_1" intersects G_1'

Theorem 5: *A "moving to target" segment can never intersect another "moving to target" segment.*

Proof: In all the algorithms, the robot can only transit to "moving to target" at a point that is closer to the target than any previously visited. During "moving to target" the distance to target monotonically decreases. Hence, the robot will never pass over any previously visited point during a "moving to target" segment. ■

Theorem 6: *A robot can never encounter an obstacle along a previously visited boundary segment.*

Proof: The robot leaves at a point that is closer to the target than any previously visited. During "moving to target" that distance monotonically decreases. If an obstacle is encountered then the hit point will be closer to the target than any point previously visited and therefore cannot be along a previously visited boundary segment. ■

Theorem 7: *Under the Alg2 and DistBug leaving rules, if the robot leaves at any point along O associated with A_X, it cannot leave again until it reaches V_X'.*

Proof: In an A_X, the robot can leave at V_X or along G_X at a leave point L_X. If this happens, A_X's convex property ensures an encounter along G_X' at point H_X. The robot turns left and proceeds to Q_X' and finally to V_X'. The robot cannot leave when it is on G_X' because it is a disabling segment. The only other place it can possibly leave is when traveling on G_X". It follows that the robot cannot leave on G_X" since A_X's convex property guarantees an encounter along G_X' thereby violating either Theorem 5 or Theorem 6. Consider that a "moving to target" segment which encounters G_X' along $[H_X,Q_X']$ violates Theorem 6. Alternatively, if a "moving to target" segment encounters G_X' along $[Q_X,H_X]$ then Theorem 5 is violated since the robot must pass through $[L_X,H_X]$ during "moving to target" mode. ■

Theorem 8: *Under the Bug2 leaving rule, if the robot leaves at L_X and reencounters O at H_X within an A_X, it cannot leave again until it reaches V_X'.*

Proof: If the robot left twice within A_X', this implies that the M-line intersects A_X more than two times. However, the convex property of A_X ensures that this cannot happen. ■

Corollary 1: *There can be at most one hit and leave point associated with any given A_X.*

Theorem 9: *If an additional restriction that S is outside O's convex hull is introduced then the robot cannot revisit a previously visited point on a semi-convex obstacle.*

Proof: The robot can only be in one of three states at any given time after encountering O and before leaving it. It can be traveling along O's convex hull towards V_T in which case, it can never leave because the direct path to T is immediately obstructed by O. Alternatively, it can be past V_T and along O's convex hull, thus if the robot leaves it will reach T. Finally, it can be inside any given A_X. Given Corollary 1 and a fixed wall-following direction, the robot will necessarily reach V_X' without revisiting any part of O associated with A_X.

An additional restriction that S must be outside O's convex hull is introduced to prevent the robot leaving along G_X". This may be possible if S causes H_X to be positioned such that leaving along G_X" does not violate Theorem 5 or Theorem 6. Figure 4-5 illustrates one such situation. ■

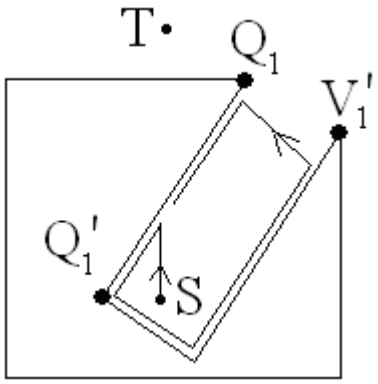

Figure 4-5: Alg2 revisits a previously visited point because S is inside the convex hull

Theorem 10: *In an environment featuring a single semi-convex obstacle O, reachable T and S outside O's convex hull, the shortest algorithm is TangentBug, followed by DistBug, Alg2, Bug2 and Bug1.*

Proof: The optimal path is from S to V_S to V_T to T where the robot travels along the convex hull of O from V_S to V_T.

Consider TangentBug traversing a semi-convex obstacle. Along its path, it may encounter a V_X. At any given V_X, A_X's convex property ensures that TangentBug can scan along every surface of O associated with A_X and hence drive directly to V_X'. Since both V_X and V_X' are on O's convex hull, the line segment [V_X, V_X'] is on the convex hull of O, hence TangentBug's path from V_S to V_T is optimal. Further, its path from S to V_S and V_T to T is optimal, and hence TangentBug will always travel on the optimal path for any given semi-convex obstacle.

Next consider Alg2 against Bug2. If the M-line does not intersect an area A_X then Bug2 must travel across the entire O associated with A_X. However, it may be possible for Alg2 to leave earlier and hence produce a shorter path.

Alternatively, consider A_X intersected by the M-line at M_X and M_X'. If the robot encounters P within [M_{X-1}',V_X], then there necessarily exists P' within the segment [V_X,M_X] such that d(P,T) > d(P',T) according to Theorem 3. Under Alg2, the robot leaves at or before P' which is always at or before M_X and hence a shorter or equal path length is always obtained for any A_X. Figure 4-6 illustrates a sample scenario and the robot can be made to leave at P' by arbitrarily narrowing W. Finally, Bug1 produces the longest path because it circumnavigates the obstacle without leaving. ■

4.4 Simulation Results

Simulations on Eyesim [7] were performed on two environments. Figure 4-7 and 4-8 show Bug1 (top left), Bug2 (top center), Alg2 (bottom left), DistBug (bottom center) and TangentBug (bottom right). By visual inspection, it is clear that Theorem 10 holds.

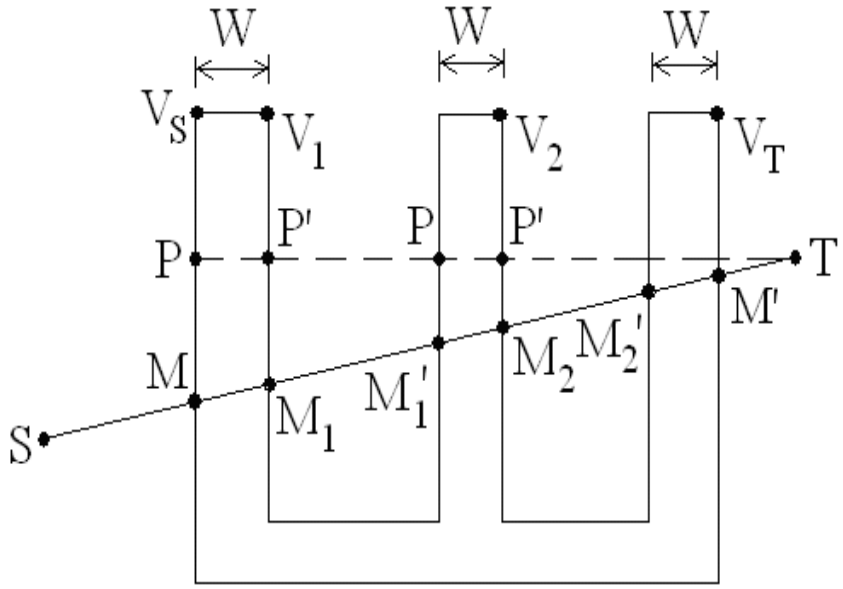

Figure 4-6: Alg2 always leaves before Bug1 and Alg2 because there always exists P' within V_X and M_X.

4.5 Two or More Obstacles

The possibility of extending Theorem 10 to environments featuring two or more semi-convex obstacles was investigated. However, it is possible to manipulate path length performance using two obstacles. Consider Figure 4-9 which features two convex obstacles. Alg2, DistBug and TangentBug are disadvantaged because they encounter the top obstacle whereas Bug1, Bug2 does not. Further, the width of the top obstacle can be made arbitrarily wide. Clearly, if Theorem 10 is to be extended, there would have to be a restriction such that all algorithms encountered the same obstacles.

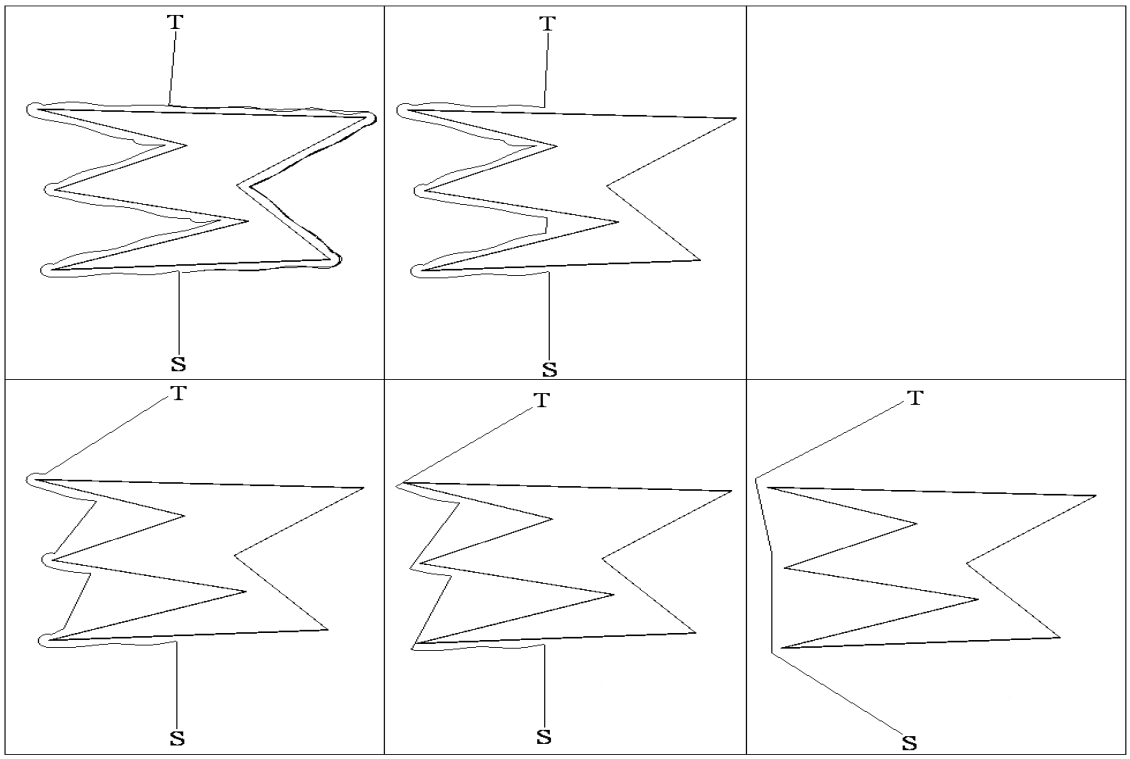

Figure 4-7: Bug algorithm paths. Top left: Bug1, top center: Bug2, bottom left: Alg1, bottom center: Alg2, bottom right: TangentBug

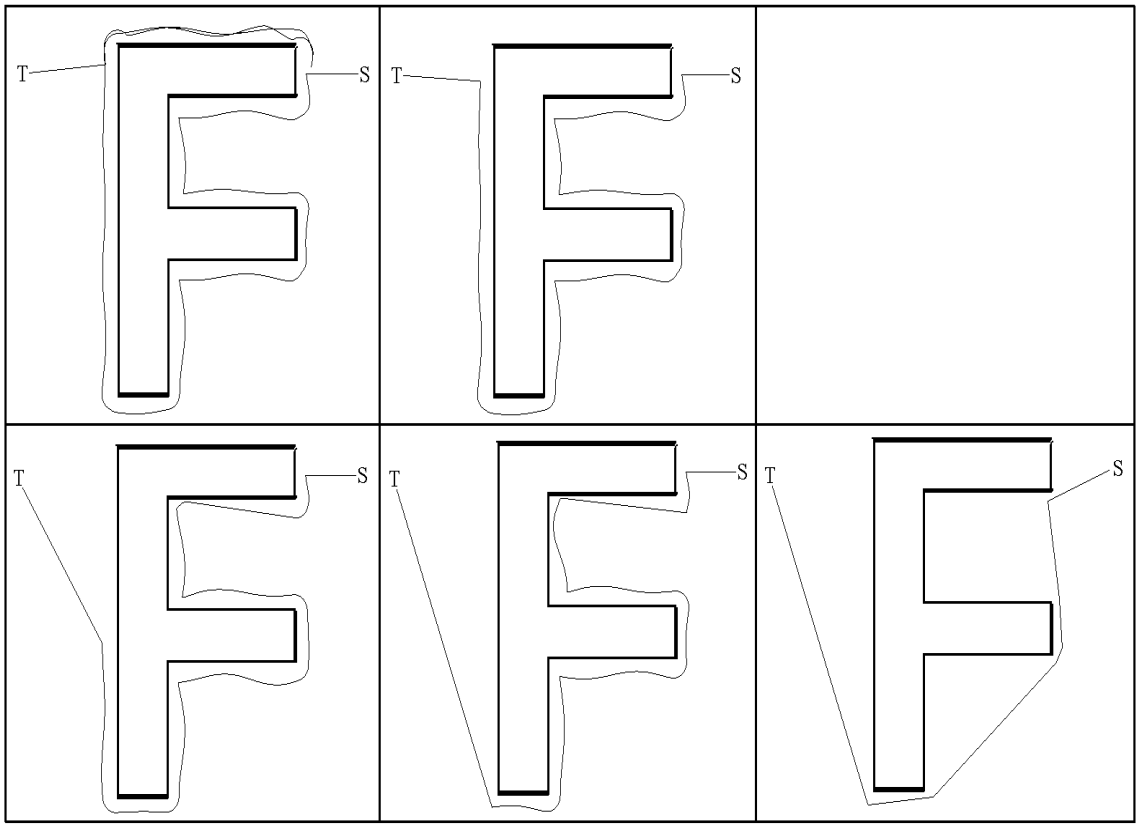

Figure 4-8: Bug algorithm paths. Top left: Bug1, top center: Bug2, bottom left: Alg1, bottom center: Alg2, bottom right: TangentBug

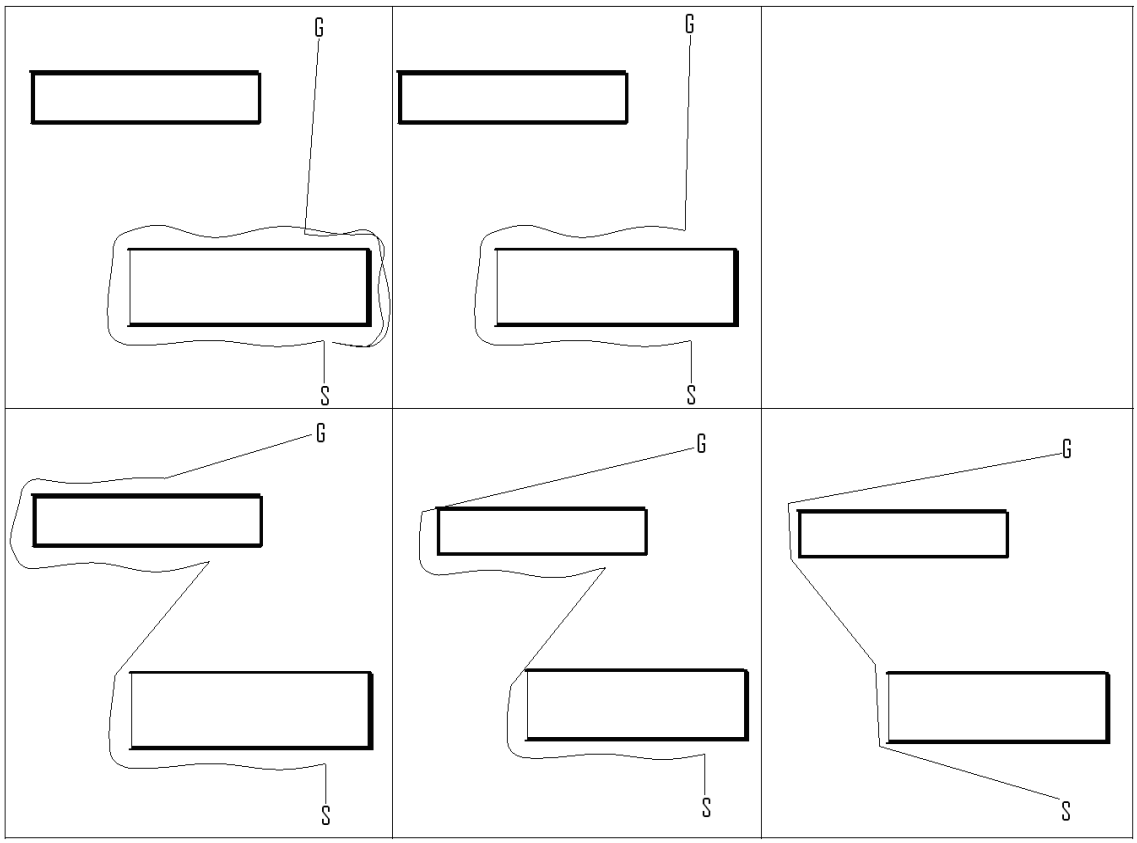

Figure 4-9: Performance on environments with more than one semi-convex obstacle can be manipulated. Top left: Bug1, top center: Bug2, bottom left: Alg1, bottom center: Alg2, bottom right: TangentBug

Chapter 5

Robot Navigation with a Guide Track

5.1 Introduction

This chapter examines a novel idea that there exists a guide track between the start and target [27]. Apart from that, everything is as before with the Bug algorithms.

Curv1 [19], a robot navigation algorithm, was developed to guide a robot to the target in an unknown environment with a single non-self intersecting guide track. Curv1 is expanded in four different ways. Firstly, self-intersecting track is explored and a new algorithm Curv2 is developed to guarantee termination. Secondly, the question of whether or not Curv1 is the only algorithm capable of guiding the robot is addressed. Thirdly, dynamic obstacles are considered. These obstacles can come and go during the robot's journey. Lastly, multiple start/target pairs and multiple trails are considered. A new algorithm Curv3 is developed to uniquely match start/targets.

We consider the problem of path planning for a robot in an unknown two-dimensional environment. Originating from a starting position, the robot is required to move to the target. There exists a trail from the start to the target. Also, there are finitely many obstacles each with finite perimeter. These obstacles may or may not lie on the trail. The robot is able to determine the direction of the guide track which it must follow to reach the target.

The motivation for this problem is in industry where it is common practice for a mobile robot follow a trail to navigate from one position to another [22]. The advantage of a trail is very similar to the advantage of roads which humans use – it allows the user to reach the destination with little autonomous navigation ability. There is no need for localization and thus error compensation techniques like probabilistic localization [20], Kalman Filters [23] and SLAM [24,25]. The disadvantage is that a trail must be created before navigation can proceed.

In an industrial setting, it is also foreseeable that objects may fall onto the floor and obstruct the trail. This is incorporated into the problem by allowing obstacles to lie on the trail. It is assumed that path planners know the trail before the robot commences its journey. On the other hand, obstacles are assumed random and dynamic and are always unknown. At no point does the robot possess knowledge of the trail, any obstacles or its own position. It simply follows the preprogrammed algorithm.

Prior work on this problem is presented. Then, four additional issues are explored. Firstly, self-intersections within the trail are permitted and necessary changes to Curv1 are implemented to guarantee termination. Previously, the trail cannot contain such intersections.

Secondly, the question of whether or not Curv1 is the only algorithm capable of solving the problem is explored. Thirdly, the problem of dynamic obstacles is explored. Dynamic obstacles are a new class of obstacle which can appear and disappear during the robot's movement to the target. Lastly, environments with multiple trails are considered. In such environments, each start is uniquely paired with a target. Curv1 is extended to guide a robot to a user-specified target.

5.2 Prior Work

Prior work was done by Sankar [3] in which the environment was restricted to static obstacles and a single non-intersecting trail. Based on the Pledge Algorithm [21] and the Bug algorithms [1,30,62] Curv1 was developed and it is repeated in Figure 5-2 with an example in Figure 5-1:

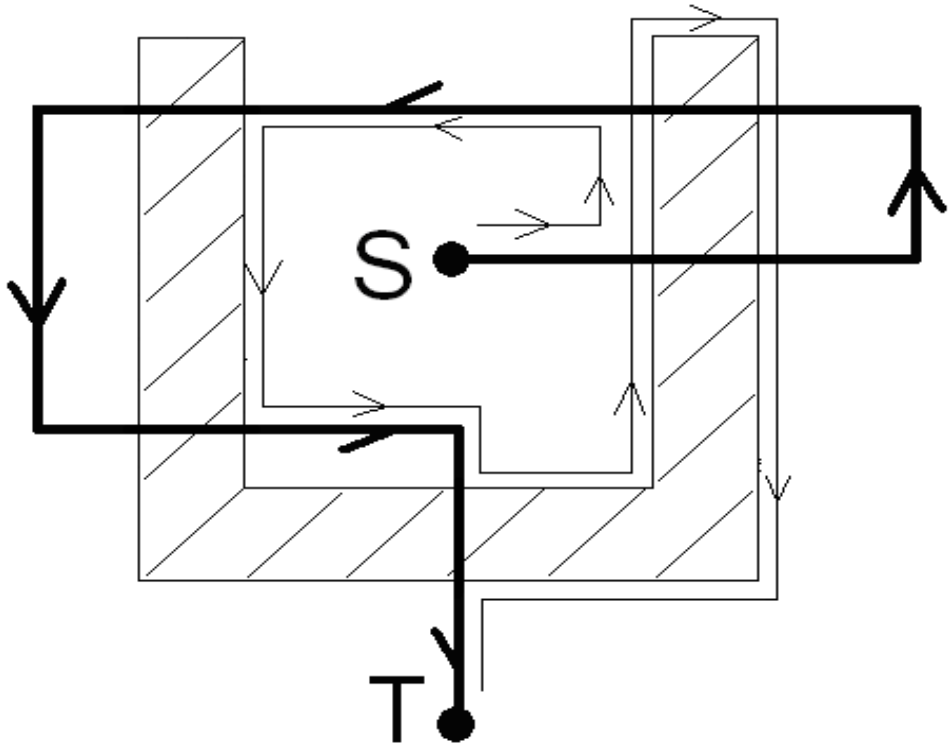

Figure 5-1: An example of the Curv1 algorithm

Curv1 assumes that the target is always reachable. Recall that in all the Bug algorithms, the robot always had to store at least one point to detect circumnavigation. If circumnavigation was detected, the robot would usually conclude that the target is unreachable. Since Curv1 assumes no localization ability, there is no way to detect that circumnavigation occurred and hence that the target is unreachable.

1. Set Counter C to zero. Start from point S and move towards T. Go to step 2.

2. Move along curve ST until one of the following occurs:

(a) The target T is reached. Stop

(b) An obstacle is met. Follow the obstacle boundary in the local direction left. Go to step 3.

3. Follow the obstacle boundary until one of the following occurs:

(a) The target T is reached. Stop

(b) The curve ST is met at point P. One of the following steps is executed:

i. The counter C reads zero and, at P, MA can move along curve ST towards T. Leave the obstacle and follow the curve ST. Go to step 2.

ii. The counter C is non-zero and, at P, MA can move along curve ST towards T. Decrement counter C by one. Continue moving along the obstacle boundary. Go to step 3

iii. At P, MA cannot move along curve ST towards T. Increment counter C by one. Continue moving along the obstacle boundary. Go to step 3.

Figure 5-2 The Curv1 Algorithm

5.3 Self-Intersecting Track

In this section, we explore whether Curv1 can be modified to accommodate a track which self-intersects at finitely many places and preserves Curv1's advantage that it does not require localization. It turns out that self-

intersections can be treated in exactly the same way as obstacles. That is, when the robot encounters a self-intersection, it can treat it like an obstacle except that it does not have to perform wall following. The ideas behind this are explained in more detail in this section.

The following notation is introduced for self-intersections. Let portions of the trail which enter the self-intersection be called inflows and denoted I_j where j represents the j^{th} inflow. Let portions of the trail which exit the self-intersection be called outflows and denoted O_j where j represents the j^{th} outflow. The inflows and outflows can be labeled arbitrarily. Let A represent the clockwise angle measured with respect to the inflow where robot enters the self-intersection, I_n. I_n can change if the robot revisits the same self-intersection. Figure 5-3 illustrates the notation.

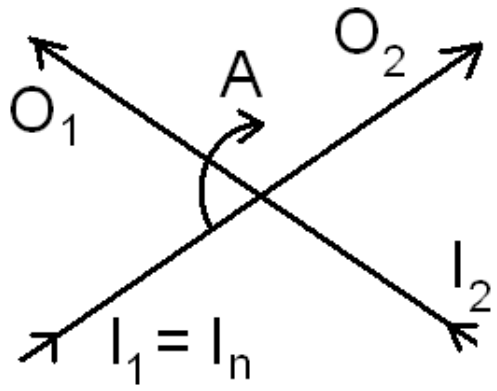

Figure 5-3: I_j represent inflows and O_j represent outflows and A is the clockwise angle measured with respect to the entry inflow I_n

Curv2 is shown in Figure 5-4.

1. Set Counter C to zero. Start from point S. Go to step 2.

2. Move along curve ST until one of the following occurs:
   (a) The target T is reached. Stop
   (b) An obstacle is met. Follow the obstacle boundary. Go to step 3.
   (c) A self-intersection is met. Initialize A=0. Go to step 4.

3. Follow the obstacle boundary until one of the following occurs:
   (a) The target T is reached. Stop
   (b) The curve ST is met at point P. One of the following steps is executed:
      i. The counter C reads zero and, at P, MA can move along ST towards T. Leave and follow the curve ST. Go to step 2.
      ii. The counter C is non-zero and, at P, MA can move along curve ST towards T. Decrement counter C by one. Continue moving along the obstacle boundary. Go to step 3.
      iii. At P, MA cannot move along curve ST towards T. Increment counter C by one. Continue moving along the obstacle boundary. Go to step 3.

4. Increase A until one of the following occurs:
      i. An outflow O_j is encountered and the counter C reads zero. Leave the self-intersection via outflow O_j. Go to step 2
      ii. An outflow O_j is encountered and the counter C is non-zero. Decrement counter C by one. Go to step 4.
      iii. An inflow I_j is encountered. Increment counter C by one. Go to step 4.

Figure 5-4 The Curv2 Algorithm

Figure 5-5 illustrates two examples of Curv2.

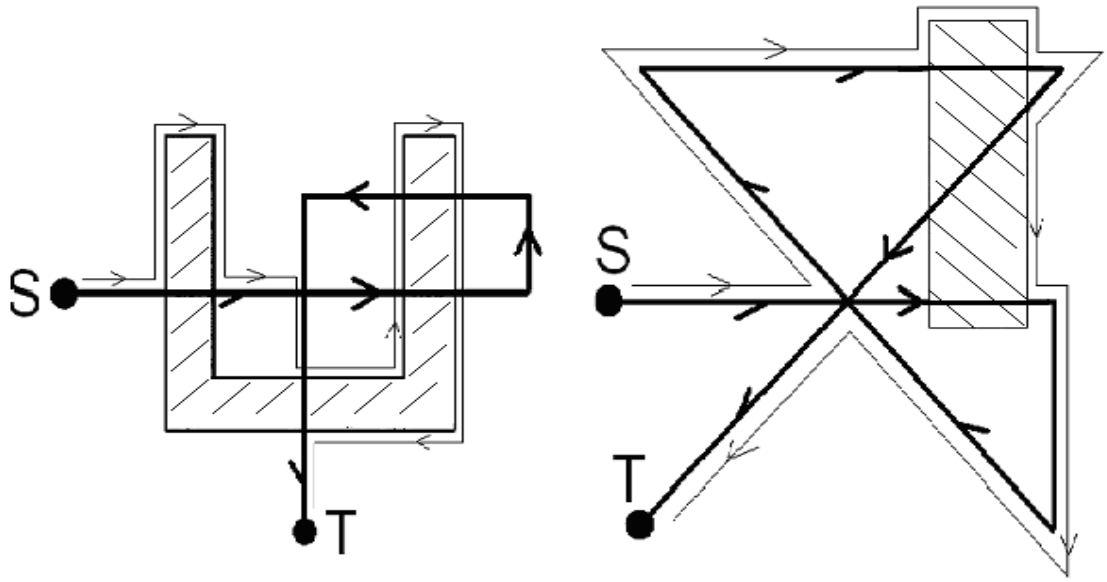

Figure 5-5: Two examples of Curv2

Obstacles can be thought of as "black boxes" which obstruct parts of the trail. It does not matter what the obstructed parts are since the robot cannot visit them. Imagine placing a very small obstacle over each self-intersection such that the self-intersection is concealed but nothing else. It follows that Curv1 could then be applied to those small obstacles to guarantee that the target will be reached.

The above reasoning is an intuitive proof as to why Curv2 can always guarantee that the target will be reached. However, a more rigorous and thorough proof is desired. Three elements of the proof have already been shown in [19] and it is trivial to adapt them to self-intersections. Firstly, Theorem 1 guarantees that a type one infinite loop [19] cannot occur. Hence, the robot will always leave a self-intersection. Secondly, Lemma 1 guarantees that the number of inflows must equal the number of outflows for any intersection. Thirdly, Lemma 4 guarantees that Curv2 generates a unique pairing between inflows and outflows for a self-intersection.

It remains to be shown that the second kind of infinite loop – where the robot leaves from the same outflow more than once – cannot occur in an environment with self-intersections. Sankar's proof may not necessarily hold because self-intersections allow the robot to reach portions of the track which make it impossible for Curv2 to reach the target. It will be shown that a robot can never visit those portions of the track.

The trail can be classified into two parts – "reachable" and "unreachable". A portion which is "reachable" is where a robot with can reach the target using Curv2. A portion which is "unreachable" is where the robot cannot reach the target using Curv2. Consider the trail depicted in Figure 5-6. On the left, the dashed trail is "unreachable" since a robot starting on the dashed trail would not be able to reach the target. It would circle the "unreachable" section indefinitely since it would turn left at the intersection. On the right, the entire trail is "reachable" since the robot can reach the target from any point on the trail using Curv2.

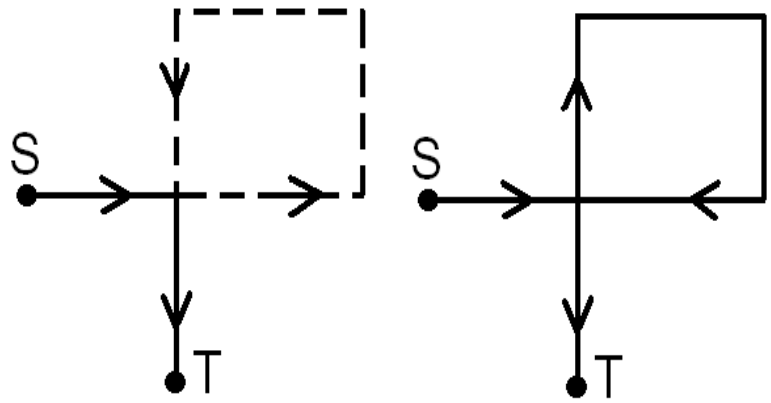

Figure 5-6 Left: Dashed track is "unreachable". Right: The whole track is "reachable"

Theorem 1: *If the robot is following the trail in an "unreachable" section then it will be in an infinite loop.*

Proof: By definition, the robot can never reach the target on an "unreachable" section. Since the robot keeps moving and the trail is of finite length, it must repeat its movements – thus causing an infinite loop. ■

Theorem 2: *The start must always lie in a "target reachable" portion of the trail.*

Proof: Suppose that start lies in an "unreachable" section. Theorem 1 implies that it must be part of an infinite loop. However, this cannot be so because it implies that the trail extends beyond start. ■

Theorem 3: *If a robot is on a "reachable" section, it can never begin following an "unreachable" portion.*

Proof: Suppose that Theorem 3 occurred and let the outflow associated with the "unreachable" portion be denoted O_U and its uniquely paired "reachable" inflow be denoted I_U. According to Theorem 1, O_U must be part of an infinite loop and therefore I_U must be part of that infinite loop too. Hence, I_U must be "unreachable" otherwise the infinite loop wouldn't exist – thus proving the theorem. ■

Corollary: *The robot can never follow a "target unreachable" trail.*

Proof: Theorem 2 guarantees that the robot will always start in a "reachable" portion of the trail. Theorem 3 guarantees that it if the robot enters a self-intersection or an obstacle in a "reachable" portion, then it will never end up in an "unreachable" portion. ■

5.4 Is Curv2 Unique?

It may be interesting to determine whether or not Curv2 is the only way to guarantee termination. For termination, there is essentially just one requirement on an algorithm. It must ensure that each inflow is associated with one and only one outflow. This requirement ensures that the robot is always able to leave the self-intersection, avoiding the first type of infinite loop, and that the second type of infinite loop cannot occur.

Theorem 4: *After entering via inflow I_j if the robot next encounters an outflow O_k a unique pairing must be formed with O_k.*

Proof: Consider if the robot did not pair with O_k. If this happens, it risks circumnavigating the obstacle if I_j and O_k are the only inflows and outflows. If circumnavigation occurs, the robot would not be able to determine this since localization is not permitted and hence a type one infinite loop is created. ■

Theorem 5: *If, the robot encounters an inflow I_n, it must not leave at I_n's Theorem1 associated outflow.*

Proof: If this occurs, then the robot would be associating an outflow with two inflows and this would violate the unique association requirement. ■

Clearly, Theorem 1 and Theorem 2 must hold for any algorithm to guarantee termination. Theorem 1 can also be thought of as the base case and Theorem 2 as the recursive component. Hence an algorithm could be structured to incorporate Theorem 1 and Theorem 2 as shown in Figure 5-7:

```
Global nextFlow = initial inflow

int getOutflow(int inflowID)
        update nextFlow to next clockwise flow
        while(nextFlow.type = Inflow)
                getOutflow(nextFlow.ID)
                update nextFlow to next clockwise flow
        end while
        return nextFlow.ID
end subroutine
```

Figure 5-7 An algorithm which incorporates Theorem 1 and Theorem 2

Given this reasoning, it appears that Curv2's strategy is the only strategy which would satisfy Theorem 1 and Theorem 2. Any modifications like switching wall-following direction would be superficial. If Curv2 is the only strategy which works, it would be interesting to know if it can still guarantee termination where there are dynamic obstacles and multiple trails.

5.5 Dynamic Obstacles

In this section, we consider modifying the problem so that obstacles can be added or removed from the environment whilst the robot is on the trail. Initially, each obstacle can be in one of two states, "on" or "off". If an obstacle is "on", then it is in the environment and the robot can detect and follow its boundary as in the static case. If an obstacle is "off", then it is not in the environment.

If the robot is following an obstacle's boundary, the obstacle cannot return to the "off" state. An obstacle cannot transition to the "on" state if the robot is inside the space occupied by its outer perimeter. Note that this includes cavities which may reside inside an obstacle. A robot must not become trapped inside a cavity of an obstacle thereby preventing it from reaching the target.

These assumptions are realistic for an industrial setting. In an industrial setting, objects could fall onto the trail and this is represented by the "off" to "on" transition. It is assumed that no obstacles will be dropped onto the robot. An obstacle can be removed from the environment during the robot's movement towards the target. This is represented from the transition from "on" to "off".

We have concluded that obstacles cannot be dynamic for Curv2 to guarantee termination. Consider the trail in Figure 5-8 left. On that trail, a robot at I_1 would move to O_2 and a robot at point I_2 would move to O_1. Now consider an obstacle transitioning to the "on" state in the position shown in Figure 5-8 right. Now, inflow I_1 is paired with outflow O_1 and inflow I_2 is paired with outflow O_2.

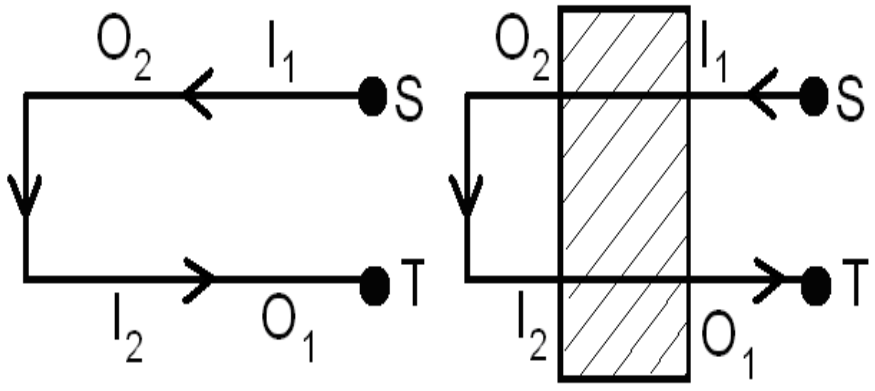

Figure 5-8 Left: With no obstacle, I_1/O_2 and I_2/O_1 are pairs
Right: With an obstacle, I_1/O_1 and I_2/O_2 are pairs

If the obstacle was “off” a robot at I_1 would move to O_2. At, I_2, it expects to move to O_1. However, if the obstacle becomes “on” whilst it is moving from O_2 to I_2, it moves back to O_2. The robot will then become stuck in an “unreachable” portion formed by O_2 and I_2 and can never move to O_1 until the obstacle is “off” state when the robot reaches I_2.

The opposite case, where the robot requires an obstacle to escape from an “unreachable” portion, can also occur. Consider the trail with the obstacle initially in the “on” state as depicted in Figure 5-9 left. The robot encounters the obstacle at I_1 and moves to O_2. Then, it follows the trail choosing to take the outflow associated with I_2 at the self-intersection. Figure 5-9 right illustrates the environment when the obstacle is “off”. If this occurs, when the robot reaches I_2, it travels to O_2 and is now part of an “unreachable” portion of the trail. The robot can only escape if the obstacle is “on” when it is at I_2.

Given these scenarios, it is apparent that restrictions must be placed on dynamic obstacles so that Curv2 still guarantees termination. Any such restrictions must, in combination, ensure that if the robot is following an “unreachable” portion of the trail, the obstacle must change its state such that the robot can get back on the “reachable” portions.

A simple restriction would require each obstacle to not change its state once the robot comes into contact with any of the inflows associated with the obstacle. “Unreachable” portions of the trail form when an obstacle transition causes a change in the unique inflow/outflow pairing and by preventing a transition, no change can occur. This restriction works by preventing changes which cause sections of the trail to become “unreachable”.

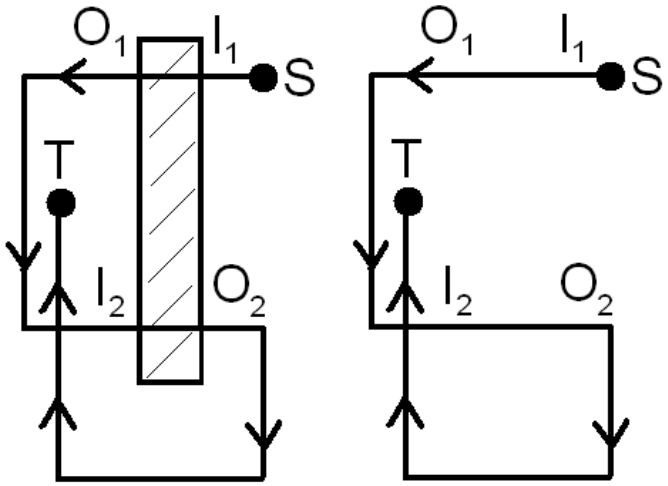

Figure 5-9 Left: With no obstacle, I_1/O_2 and I_2/O_1 are pairs
Right: With an obstacle, I_1/O_1 and I_2/O_2 are pairs

This simple restriction needs to be applied when the robot visits an inflow associated with the obstacle. At other times, the obstacle's state can fluctuate with no impact on termination. This restriction is quite imposing since it requires the obstacle's state to be fixed at certain times.

5.6 Multiple Trails

In this section, the problem of multiple trails is considered. Obstacles are assumed to be static and do not transition between states. Consider a robot using Curv2 which starts at S_i. The robot will travel along the trail and reach T_i if there are no intersections or obstacles. This is a trivial scenario. For a non-trivial scenario, each trail will intersect with another at least once. Figure 5-10 illustrates several non-trivial environments featuring multiple trails.

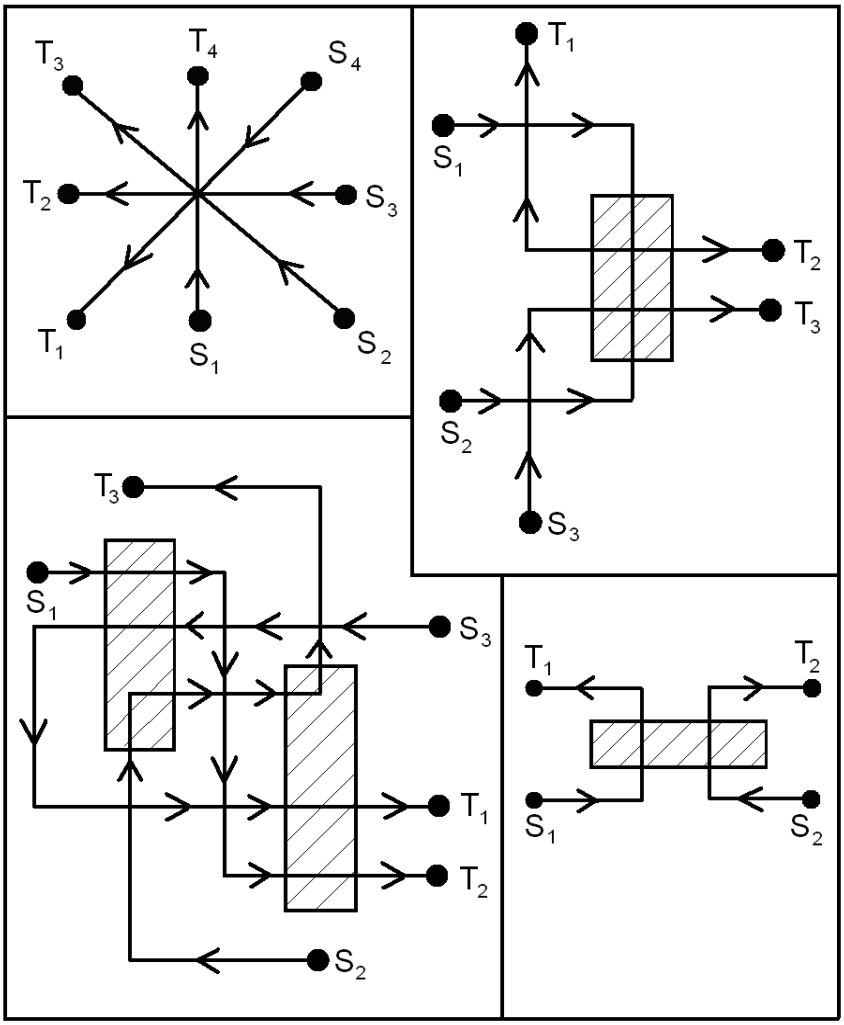

Figure 5-10: Examples of Curv2 in multi-trail environments

A robot starts at S_i. Then it encounters an inflow I_i. When it leaves via the Curv2 associated outflow O_i, it can reach either a target or an inflow to another obstacle. If it leads to T_i, then T_i is uniquely associated with O_i and therefore I_i and therefore S_i. If it leads to another obstacle at inflow I_j, it will uniquely leave via outflow O_j. If O_j leads to target T_i, then the unique associations will be S_i, I_i, O_i, I_j, O_j, T_j. This can be repeated for any finite number of intermediate inflow/outflow combinations. Since Curv2 pairs the equal number of inflow and outflows at each intersection uniquely, there must be a unique pairing between S_i and T_j.

5.7 Pairing Start and Targets

It may be desirable to allow the user to identify which target is associated with the start instead of always traveling to the Curv2 default. Presumably, this will require extra complexity but the aim is to minimize the additional complexity. To achieve this, all that needs to be done is to introduce a user-controlled mechanism which uniquely pairs the inflows and outflows of each intersection. For instance, Curv2 allows only the pairing S1 to T1 and S2 to T2 in Figure 5-11 left. However, what if we desired the pairing in Figure 5-11 right?

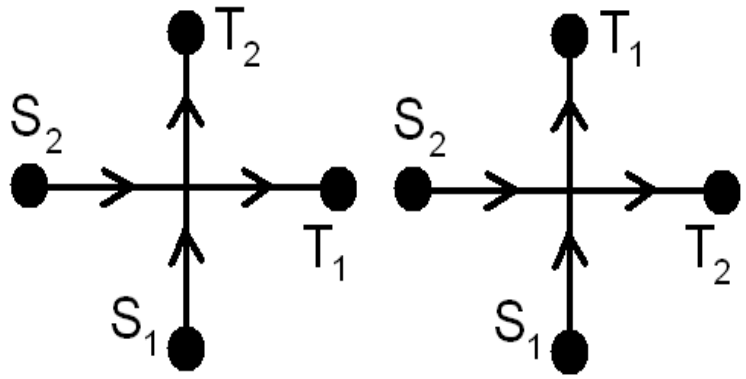

Figure 5-11 Left: S_1/T_1 and S_2/T_2 are paired under Curv2
Right: S_1/T_1 and S_2/T_2 are desired

This can be accomplished by associating a finite non-negative integer, Z, with each inflow to an intersection. Z represents the number of inflows which the robot must count before it is eligible to leave the self-intersection. Hence, a new algorithm, Curv3 can be written to accommodate for Z and is shown in Figure 5-12.

As an example, in Figure 5-11 right, associate Z=1 with both the inflows. Then, a robot starting at S_1 enters the intersection with Z=1. S_2 will be the next inflow encountered. Z is decremented to 0 and Curv3 moves into step 5. In step 5, part i will cause the robot to leave at T_1.

1. Set Counter C to zero. Start from point S. Go to step 2.
2. Move along curve ST until one of the following occurs:

   (a) The target T is reached. Stop

   (b) An obstacle is met. Follow the obstacle boundary in the local direction left. Go to step 3.

   (c) A self-intersection is met. Initialize A=0 and Z to the value associated with the inflow. If Z=0, go to step 5, else go to step 4.
3. Follow the obstacle boundary until one of the following occurs:

   (a) The target T is reached. Stop

   (b) The curve ST is met at point P. One of the following steps is executed:

   i. The counter C reads zero and, at P, MA can move along curve ST towards T. Leave and follow the curve ST. Go to step 2.

   ii. The counter C is non-zero and, at P, MA can move along curve ST towards T. Decrement counter C by one. Continue moving along the obstacle boundary. Go to step 3.

   iii. At P, MA cannot move along curve ST towards T. Increment counter C by one. Continue moving along the obstacle boundary. Go to step 3.
4. Increase A until an inflow I_n is encountered. Decrement Z. If Z=0, then go to step 5. Otherwise, repeat step 4.
5. Increase A until one of the following occurs:

   i. An outflow O_n is encountered and the counter C reads zero. Leave the self-intersection via outflow O_n. Go to step 2.

   ii. An outflow O_n is encountered and the counter C is non-zero. Decrement counter C by one. Go to step 4.

   iii. An inflow I_n is encountered. Increment counter C. Go to step 5.

Figure 5-12 The Curv3 Algorithm

A robot starting at S_2 enters the intersection with Z=1. It encounters, and ignores, T_1 and T_2 before reaching S_1. At S_1, Z is decremented to 0 and Curv3 moves into step 5. S_2 is the next inflow encountered and C is incremented to 1. At T_1, C is decremented to 0 and this causes the robot to leave at T_2.

Theorem 6: *Under Curv3, a type one infinite loop can never occur.*
Proof: Z is a finite quantity and, according to Curv3, can only decrease. Given that there must be at least one inflow to the intersection, it follows that Z must reach 0 eventually. At the inflow I_j where Z reaches 0, the robot will behave as if it entered the intersection at I_j under Curv2. Curv2 guarantees that the robot will leave the intersection. ■

Given control of the outflow associated with each inflow, it may be tempting to allow a target to be associated with more than one start. Figure 5-13 left illustrates the case where S_1 and S_2 lead to T_1T_2. The only way to achieve this would be for multiple inflows to lead to a single inflow for at least one intersection in the environment. If permitted, then termination can no longer be guaranteed. For instance, consider the environment in Figure 5-13 right below. In that environment, an infinite loop is created around the center.

Although it is possible for termination to be guaranteed by choosing Z carefully, it is outside the scope of this thesis. In this thesis, Z is always chosen such that a unique pairing between inflows and outflows exists at each intersection. Therefore, each start must be uniquely paired with a unique target. This means that the robot will never leave via the same outflow twice – a requirement necessary for guaranteeing termination.

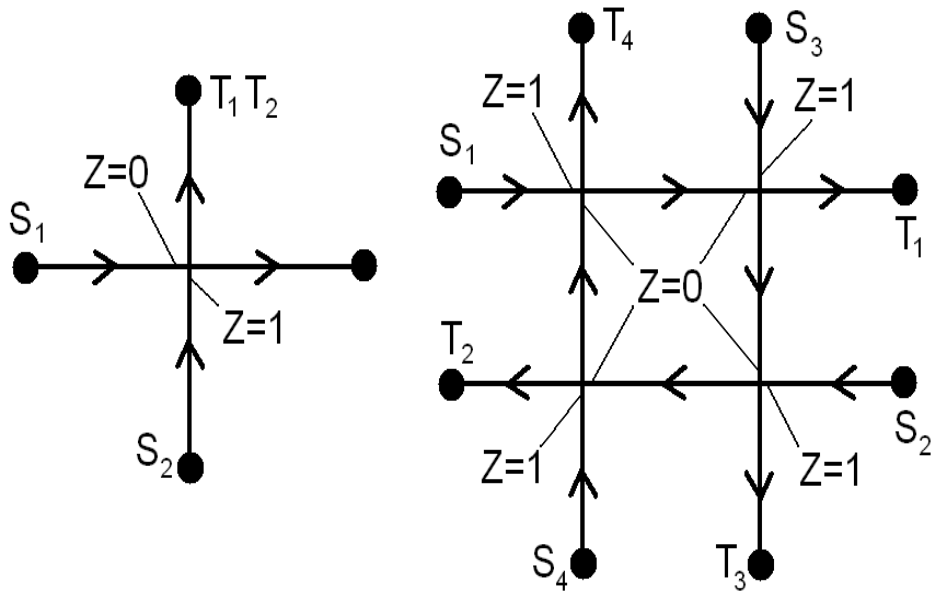

Figure 5-13 Left: S_1 and S_2 lead to the same target T_1T_2
Right: An infinite loop is created in the center

Note that if there is an obstacle along the guide tracks, then there is no guarantee that the robot will terminate at the desired target. However, it should still uniquely terminate at one of the targets because the obstacle's inflows and outflows are uniquely paired. Consider the situation depicted in Figure 5-14. On the left, the targets are uniquely paired but the unique pairing changes on the right. This is because, according to the assumptions, obstacles are "dynamic" and are not planned for, whereas tracks are "static" and hence Z can be changed when the robot enters via an inflow. Hence, Z is always 0 when it encounters an obstacle.

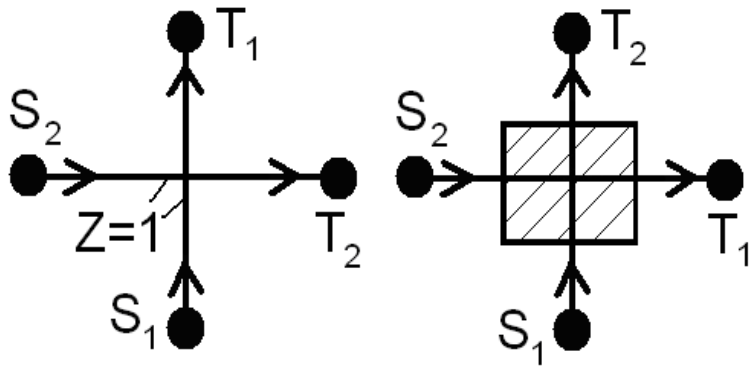

Figure 5-14 Left: Curv3 achieves a S_1/T_1 and S_2/T_2 pairing
Right: An obstacle changes the pairing.

Chapter 6

SensorBug: A Local, Range-Based Navigation Algorithm for Unknown Environments

6.1 Introduction

This chapter presents a new navigation algorithm developed by the author called SensorBug. It builds on the implementation experience in Chapter 2 and attempts to find an algorithm similar to TangentBug but without the LTG requirement. SensorBug is a new algorithm designed to use range sensors and with three performance criteria in mind: data gathering frequency, amount of scanning and path length. SensorBug reduces the frequency at which data about the visible environment is gathered and the amount of scanning for each time data is gathered. It is shown that despite the reductions, correct termination is still guaranteed for any environment. The disadvantage is that SensorBug only works when each obstacle is polygonal.

SensorBug is designed to achieve a fair balance between three competing criterion: path length, data gathering frequency and amount of scanning. Previously, data gathering frequency and amount of scanning has not been given much consideration in any previous Bug algorithm. These include Bug1 [1,62], Bug2 [1,62], Alg1[3], Alg2 [4], DistBug [5], Class1 [2], Rev1 [8], Rev2 [8], TangentBug [6], VisBug [16], HD-1 [9], Ave [17], and CautiousBug [11]. SensorBug achieves this balance by implementing the Q method which guarantees a finite number of leave points and a new

"moving to target" mode which ensures a smooth transition from the "boundary following" mode.

6.2 The Q Method

In Chapter 3, it was suggested that for any Bug algorithm to guarantee termination it necessarily has to use the "shrinking disc". The "shrinking disc" has a center at T and a radius equal to d(P,T) where P is the closest point to the target ever detected. When a Bug algorithm transitions from "boundary following" to "moving to target" mode, it must ensure that obstacles which do not intersect or are not contained within the "shrinking disc" will never be encountered. If not, then it is possible for an infinite number of obstacles to be encountered and termination is not guaranteed.

Further, a Bug algorithm must supplement the "shrinking disc" a method to keep the number of possible leave or hit points finite and ensure that the robot can leave on at least one of those points if the target is reachable. If the first criterion is not satisfied, the algorithm could have an arbitrarily long path length as manifested in Class1. If the second criterion is not satisfied, then the robot can circumnavigate O and conclude that a reachable target is unreachable.

Q-Bug implements the Q method which assumes that each obstacle is polygonal. Then, it generates a finite set of points, Q, for each polygonal obstacle O. The procedure shown in Figure 6-1 shows how Q is generated:

For each vertex V on an obstacle which intersects or is inside the "shrinking disc":

1) Place the robot at V and admit V into Q if not already in Q.

2) Without moving, use range sensors to follow O's boundary in the clockwise direction until either:

    a) V is visible. Go to step 3.

    b) O leaves the robot's visible range. If caused by a vertex, drive to that vertex. Otherwise drive to the last visible point. When the robot stops, if that point is not in Q, admit that point into Q. Repeat step 2.

3) Place the robot at V.

4) Without moving, use range sensors to follow O's boundary in the counter-clockwise direction until either:

    a) V is visible. Choose another V and repeat.

    b) O leaves the robot's visible range. If caused by a vertex, drive to that vertex. Otherwise drive to the last visible point. When the robot stops, if that point is not in Q, admit that point into Q. Repeat Step 4.

Figure 6-1 A method for determining the set Q

Using this style of moving, there are only three possible transitions between points where the robot has to stop. The first is called a vertex transition because it is caused by a vertex on the followed obstacle. A vertex transition is illustrated in Figure 6-2A for a robot at X moving to N_C. The second type of transition is called an occlusion transition because it is caused by a perceived or actual separate obstacle obstructing boundary following. An occlusion transition is illustrated in Figure 6-2B for a robot at X moving to N_C. The last visible point on the followed obstacle is

labeled N_C*. Finally, an out-of range transition occurs when the followed obstacle leaves the robot's visible range as illustrated in Figure 6-2C. This type of transition is impossible when the robot's sensor range, R, is infinite.

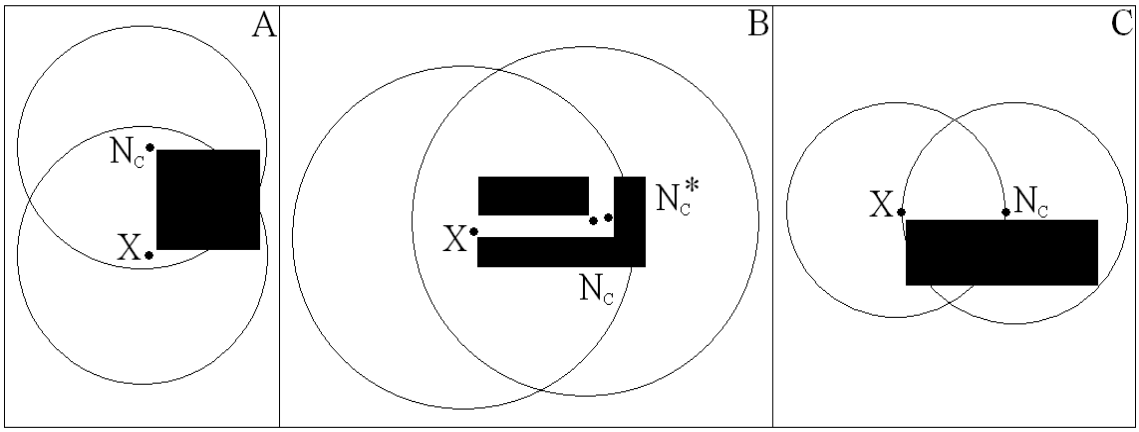

Figure 6-2: The three possible types of transition

Figure 6-3 illustrates finding Q for one particular V of one particular O. In Figure 6-3A, the robot begins at V and proceeds to the next point in Q. In Figure 6-3B, the robot travels to a vertex which is not on O. However, that vertex is still considered part of Q. Clockwise circumnavigation continues until V is visible in Figure 6-3E. The robot is placed at V and counter-clockwise circumnavigation begins in Figure 6-3F. Finally, the additions to Q are presented in Figure 6-3L.

Note that there are instances where Q is null because there are no vertices. This could occur if the robot is trapped inside an obstacle as depicted in Figure 6-4. Theorem 2 will show that SensorBug guarantees correct termination for such environments.

The Q method must satisfy the second criteria. That is, if the target is reachable then the robot will always leave on a point in Q. Theorems 10, 11, 12 and 13 will show that the Q method satisfies the second criteria if the robot retrieves the closest point to the target in the visible environment P_{ENV}

and compares $d(P_{ENV},T) < d(P,T)$ for all points in Q visited during circumnavigation. This effectively checks if P_{ENV} is inside the "shrinking-disc". The robot's position X is considered a point in the visible environment and hence $d(P_{ENV},T) \leq d(X,T)$ must always hold.

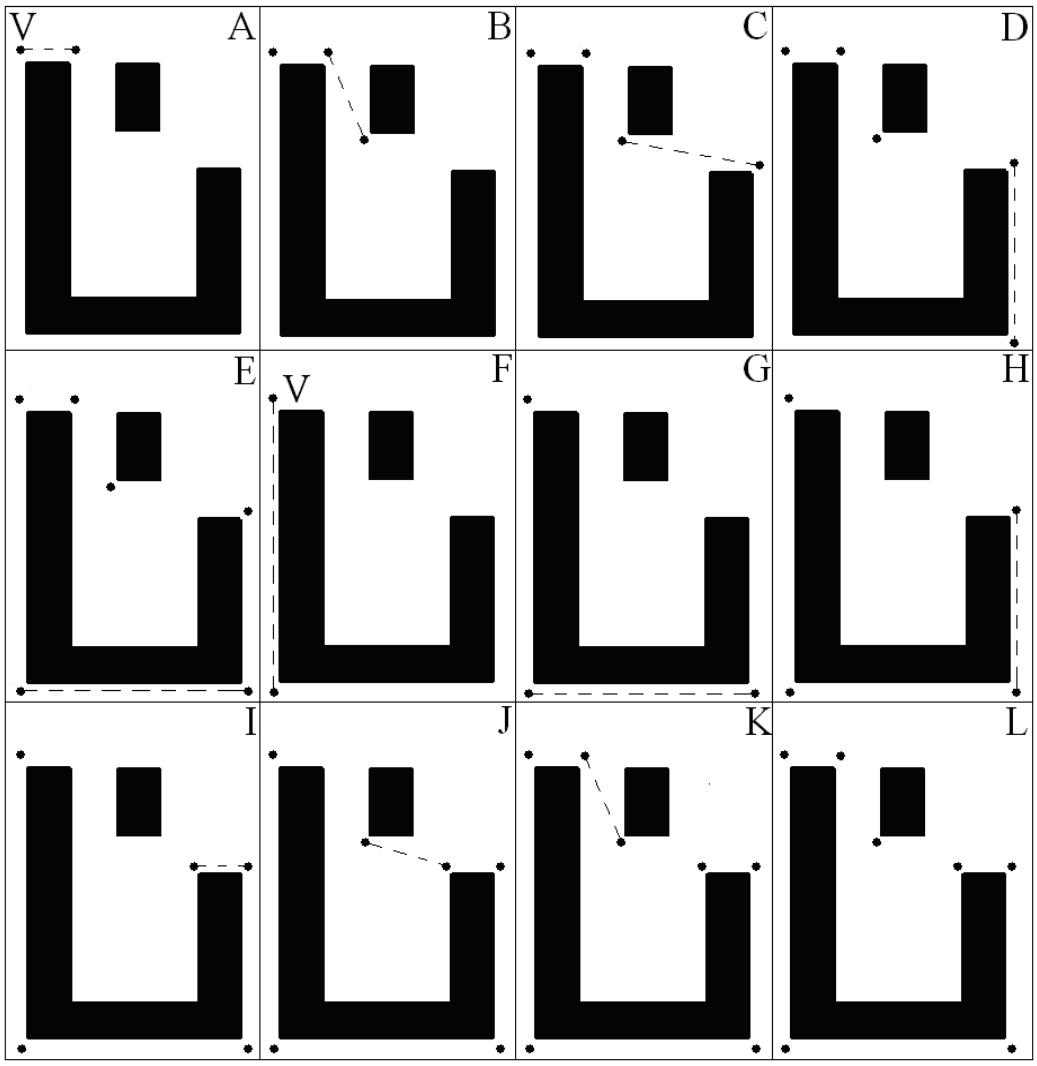

Figure 6-3: Determining Q for a particular vertex

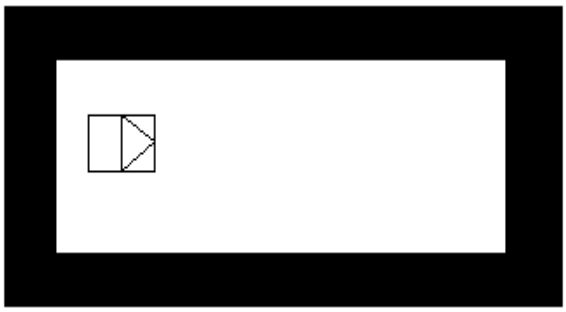

Figure 6-4: An instance where Q is null

6.3 Boundary Following Mode

To implement the Q method, SensorBug must ensure that the robot can only leave at points within Q. If this did not occur, then the number of leave points could be infinite and termination cannot be guaranteed. This can be achieved by requiring the robot to travel to at least one vertex before leaving.

Given the "shrinking-disc", the leaving rule and implementation of the Q method, the "boundary following" mode of SensorBug can be designed. It can be assumed that at least one point P_{ENV} on O is visible when "boundary following" begins. Hence, the following pseudo-code for boundary following is shown in Figure 6-5.

1) Perform clockwise boundary following on the obstacle on which P_{ENV} lies whilst updating P until either:

a) the robot reaches a vertex V on O. Go to step 2.

b) the robot completes circumnavigation around O. Target is unreachable and SensorBug terminates.

2) Retrieve P_{ENV}. Locate the next clockwise point N_C in Q associated with V. Update P.

If $d(P_{ENV},T) < d(P,T)$,

go to "moving to target" mode.

Else If, V is visible and it is in the clockwise direction,

the target is unreachable and SensorBug terminates.

Else,

drive to N_C and repeat step 2.

Figure 6-5 Boundary following mode for SensorBug

To implement step 1a, a vertex can be detected before the robot has reached it because a vertex will produce a sudden discontinuity in the range sensor readings when attempting to follow O with range sensors. The only eligible replacements of P are points that are on O's boundary. To replace P, an eligible point P_{OBS} must satisfy $d(P_{OBS},T) < d(P,T)$. In any case, if P_{ENV} can replace P, the leaving condition should hold and the robot should switch to "moving to target".

Note that when driving to N_C in step 2, the SensorBug algorithm thread can sleep. That is, the robot does not have to continuously generate an LTG [6], update F [5], check the M-line [1,3] or any other continuous task. An implementation strategy to retrieve P_{ENV} is suggested in section 6.10. P_{ENV} is retrieved using least amount of range scanning possible. Also, in Section 6.6, the use of multiple previously stored points [3] is discussed and a more complicated "boundary following" mode is derived to use them.

6.4 Moving to Target Mode

Having established a "boundary following" mode, a compatible "moving to target" needs to be designed. Amongst the tactile sensor algorithms, this transition is straight-forward because the robot would already be inside the "shrinking disc". However, with range sensors, it is more complicated because although the robot has located a point inside the "shrinking disc" using range sensors, its actual position X may not be inside the "shrinking disc". Hence, care must be taken to ensure that no obstacles can be admitted to the finite set in the "moving to target" segment when the robot is outside the "shrinking disc".

Other Bug algorithms with range sensors have overcome this problem in different ways. In DistBug [5], only F is evaluated to generate P_{ENV} which is inside the "shrinking disc". If the leaving condition is satisfied, then the robot travels directly to the target thereby guaranteeing that P_{ENV} will be visited. In TangentBug [6], there is a transition phase where the robot heads towards the node which caused leaving if the robot is outside the "shrinking disc". This terminates when the robot is inside the "shrinking disc". Whilst these solutions suit their respective algorithms, they would be incompatible with SensorBug "boundary following".

The pseudo-code in Figure 6-6 describes SensorBug's "moving to target" mode assuming a line-of-sight to P_{ENV} retrieved by a previous "boundary following" segment or during SensorBug's initialization:

```
Update P = P_ENV.
    If P = T,
        drive to the target and SensorBug terminates.
    Else if P is on an obstacle,
        perform "boundary following" on the obstacle which P is on
    Else,
        drive to P, retrieve a new P_ENV and repeat.
```

Figure 6-6 SensorBug's "moving to target" mode

Clearly, this "moving to target" mode ensures that obstacles outside the "shrinking disc" cannot be admitted since P must be visible in the environment where a new P_{ENV} is retrieved. This implies that P_{ENV} must fall

inside the “shrinking-disc” and therefore any obstacle which P_{ENV} should lie on must also be inside the “shrinking-disc”.

Once again, it should be noted that whilst driving to P_{ENV}, the SensorBug thread can sleep. However, this strategy may not achieve a desirable balance between environment scanning frequency and path length. If the user desires shorter path length at the expense of increased environment scanning frequency, then the following strategy can be employed.

Assume that the only obstacles in the entire environment are the ones within the current visible environment and that the currently visible obstacles are thin walls. These assumptions were previously used in TangentBug for the expected nodal distance to target calculations [6]. Draw a line from X to P. Then, identify all all vertex transitions. For each vertex, draw a line from the vertex to the target and intersect it with the line from X to P. In Figure 6-7 Left, Y_1 and Y_2 are stop points whereas in Figure 6-7 Right, Y_1 is not a stop point since the intersection is closer to the target than the endpoint.

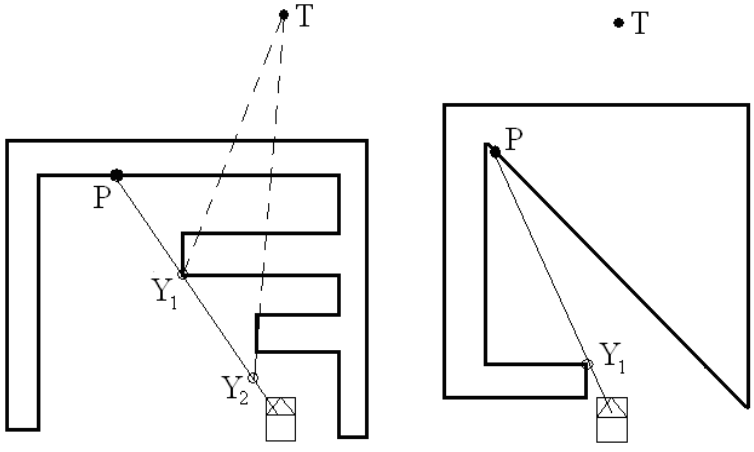

Figure 6-7: Illustration of stop points on two obstacles

The "moving to target" mode can be rewritten to accommodate stop points as shown in Figure 6-8:

1) Update $P = P_{ENV}$. Evaluate stop points on the direct line segment starting at X and ending at P_{ENV}.

    If $P_{ENV} = T$,

        drive to the target and SensorBug terminates.

    Else If a stop point exists,

        drive to the stop point. At the stop point, retrieve P_{ENV} and repeat step 1.

    Else,

        go to step 2.

2) If P_{ENV} is on an obstacle,

        begin "boundary following" on the P_{ENV}'s obstacle.

    Else,

        drive to P_{ENV}. At P_{ENV}, retrieve P_{ENV} and go to step 1.

Figure 6-8 "Moving to target" mode with stop points

6.5 Scenarios

In this section, two examples of SensorBug are illustrated. In both cases, the robot has infinite range sensors and stop points are used. In Figure 6-9A, the robot is identifies P_{ENV} along with Y_1 and Y_2 as stop points and then drives to Y_1. In Figure 6-9B, the robot retrieves P_{ENV} but since P_{ENV} did not move, it drives to stop point Y_2. In Figure 6-9C, P_{ENV} moves closer to the target and the robot drives to stop point Y_1. In Figure 6-9D, the target is visible and the robot drives directly to it.

In Figure 6-10A, the robot identifies P_{ENV} and drives to stop point Y_1. In Figure 6-10B, P_{ENV} moves closer to the target and the robot drives to a new stop point Y_1. In Figure 6-10C, P_{ENV} moves closer to the target but this time, there are no stop points and P_{ENV} lies on an obstacle. Therefore, the robot transitions to "boundary following" mode. All points on O's boundary from P_{ENV} to V are eligible to replace P and P moves to P_{ENV} since that is closest to T. The robot drives to V and since it has traveled to a vertex, step 2 of "boundary following" is executed. In Figure 6-10D, P_{ENV} has moved but $d(P,T) < d(P_{ENV},T)$ holds and the robot cannot leave. So, the robot locates N_C in Q and drives directly to it. In Figure 6-10E, P_{ENV} has moved but once again the leaving condition does not hold and the robot drives to N_C in Q. In Figure 6-10F, P_{ENV} = T and hence the leaving condition holds. Then Step 1 of "moving to target" drives the robot directly to T where it terminates.

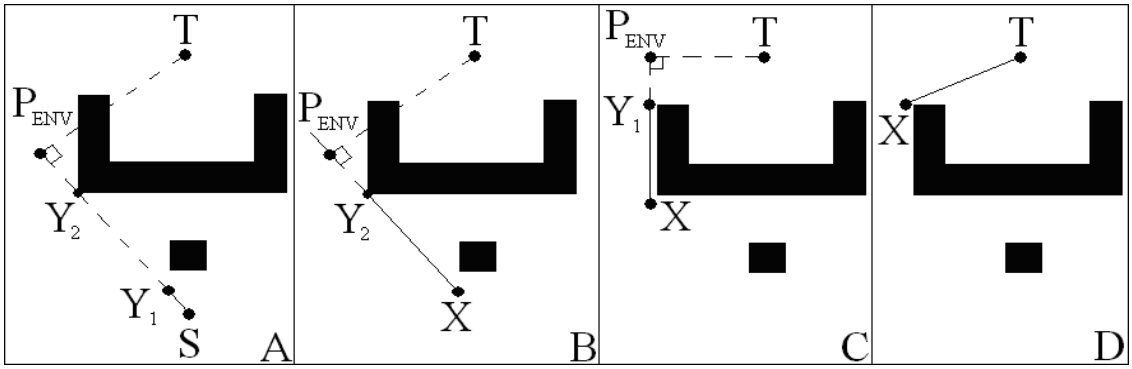

Figure 6-9: SensorBug example

6.6 Multiple Previously Stored Points

The "boundary following" mode presented in section 6.4 required only one stored point to be maintained so that circumnavigation can be detected. However, other Bug algorithms [3,4] have used multiple previously stored points to simultaneously guarantee termination and reduce path length. If

the user wishes to tradeoff additional complexity and memory requirements for lower path length then such a scheme is desired for SensorBug. In SensorBug, termination is already guaranteed using one stored point so the only purpose is to reduce path length. The new "boundary following" mode with multiple stored points is presented.

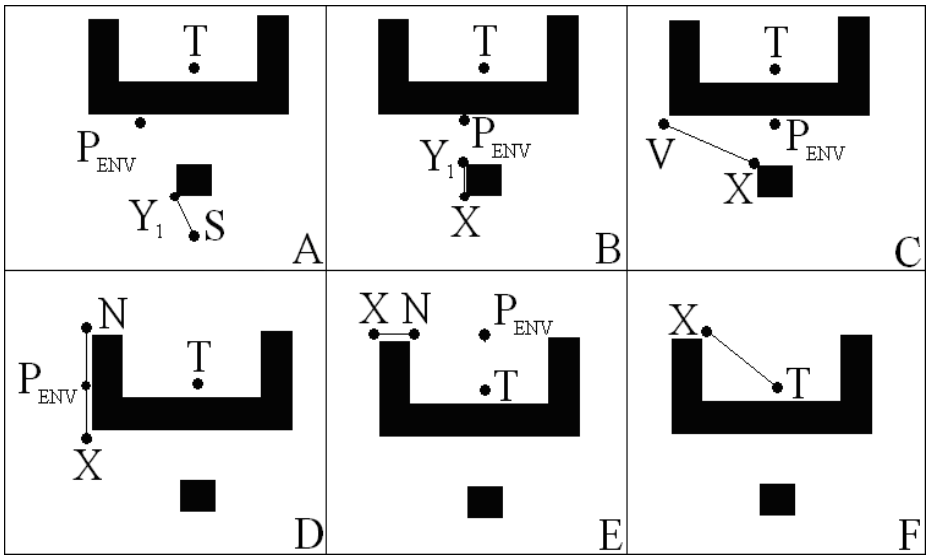

Figure 6-10: SensorBug example

At a higher level of abstraction, the stored points scheme keeps track of sections of O's boundary which have already been explored by storing two points N_C and N_{CC}. N_C represents the furthest clockwise point on which the leaving condition has been evaluated, likewise for N_{CC} except that it represents the counter-clockwise point. The scheme directs the robot to the closest unexplored region along O should it find itself in a previously explored region. The remaining details are necessary to guarantee that SensorBug terminates correctly and to minimize memory requirements.

1) Perform clockwise boundary following on the obstacle on which P_{ENV} lies whilst updating P until:

a) the robot reaches a vertex V on O. Go to step 2.

b) the robot completes circumnavigation around O. Target is unreachable and SensorBug terminates.

c) a previously stored N_{CC} point is visible. Drive to that point and go to step 4.

2) Store N_C = X and N_{CC} = X along with $d_{PATH}(N_C, N_{CC})$. Locate the next clockwise point N_C in Q associated with V. Update P. Retrieve P_{ENV}.

If $d(P_{ENV},T) < d(P,T)$,

go to "moving to target" mode.

Else If, a previously stored N_{CC} point is visible,

Drive to the previously stored N_{CC} point and go to step 4.

Else,

Drive to N_C and go to step 3.

3) Update N_C = X. Locate the next clockwise N_C in Q associated with V. Update P and the distance $d_{PATH}(N_C, N_{CC})$. Retrieve P_{ENV}.

If $d(P_{ENV},T) < d(P,T)$,

go to "moving to target" mode.

Else If, V is visible and it is in the clockwise direction,

the target is unreachable and SensorBug terminates.

Else If, a previously stored N_{CC} point is visible.

Drive to the previously stored N_{CC} point and go to step 4.

Else,

Drive to N_C and repeat step 3.

4) If the current $d_{PATH}(N_C, N_{CC})$ is less than the stored $d_{PATH}(N_C, N_{CC})$,

Delete all stored points data associated with the current O except for the previous N_C. Go to step 5.

Else,

Delete all stored points data associated with the current O except for the current N_{CC}. Go to step 6.

5) Update N_{CC} = X. Locate the next counter-clockwise point N_{CC} in Q associated with V. Update P and the distance $d_{PATH}(N_C, N_{CC})$. Retrieve P_{ENV}.

If $d(P_{ENV},T) < d(P,T)$,

go to "moving to target" mode.

Else If, N_C is visible and it is in the counter-clockwise direction,

terminate with no success.

Else,

Drive to N_{CC} and repeat step 5.

6) Update N_C = X. Locate the next clockwise point N_C in Q associated with V. Update P, and the distance $d_{PATH}(N_C, N_{CC})$. Retrieve P_{ENV}.

If $d(P_{ENV},T) < d(P,T)$,

go to "moving to target" mode.

Else If, N_{CC} is visible and it is in the clockwise direction,

terminate with no success.

Else,

Drive to N_C and repeat step 6.

It should be shown that N_{CC} will always be the first visible previously stored point as assumed in steps 1, 2 and 3 of multiple previously stored points. Using Theorem 1 and the fact that clockwise boundary following is always chosen, it follows that N_{CC} will always be the first visible previously stored point.

Theorem 1: *The P_{ENV} which lies on O and initiates "boundary following" must always reside on a previously unexplored region of O.*

Proof: When a robot leaves an obstacle, this indicates that there exists P_{ENV} which is closer to the target than any explored point on the currently followed O. During the subsequent "moving to target" phase, P will always be replaced with P_{ENV} closer to the target than its predecessor. When P_{ENV} lies on an obstacle at the end of the "moving to target" segment, it is closer to the target than any previously explored point and hence could not have been explored previously. ■

6.7 Examples for Multiple Previously Stored Points

The following examples illustrate SensorBug with multiple previously stored points. Figure 6-11 illustrates the first environment which was first used by Sankar [3] in demonstrating Alg2. The beauty of this terrain is that it is complicated enough to demonstrate multiple stored points but is not overwhelming.

In part A, the robot detects that there are no stop points on the segment from S to P_{ENV} and that P_{ENV} lies on an obstacle. The robot switches to "boundary following" mode and drives to V. In part B, the robot stores N_C and N_{CC} at X and determines that P_{ENV} is closer than P, consequently leaving. In part C, the robot locates stop points Y_1 and Y_2 driving to Y_1.

In part D, the robot attempts to find P_{ENV} closer than P but is unsuccessful so it drives to Y_2. In part E, P_{ENV} closer than P is detected and P_{ENV} is on an obstacle with no stop points. The robot switches to "boundary following" mode and drives to V. In part F, the robot stores N_C and N_{CC} at X and drives to N_C. In part T, the robot follows O's boundary. In part H, the robot firstly checks the leaving condition at X and then it detects a previously stored N_{CC} point is visible and drives to it. In part I, the robot evaluates the

clockwise and counter-clockwise directions and decides that the clockwise direction is shortest and deletes all stored points associated with O except for the current N_{CC}. In part J, the robot follows O's boundary not leaving as in part B because P is closer to the target and hence the leaving rule does not hold. In part K, N_C lies on a perceived separate obstacle. Obviously, N_C cannot be stored alone. Instead, the point N_C* is also stored and associated with N_C such that if N_C* becomes visible in any future "boundary following" mode, the robot will drive to N_C if it is visible, otherwise, it will drive to N_C* and then to N_C. In part L, the robot firstly updates $N_C = X$ and its associated N_C* point. Then it evaluates the leaving condition which holds and it switches to "moving to target" mode. In part M, the robot drives to stop point Y_1. In part N, the target is visible and the robot drives directly to it. Part O displays the final SensorBug path.

The second example is very similar to the first except for a modification which makes the target unreachable. The robot proceeds as per parts A-L of Figure 6-11. In Figure 6-12 part A, the robot drives to stop point Y_1. In part B, the robot finds that P_{ENV} lies on O and no stop points exist. It switches to "boundary following". In parts C, D, and E, the robot the robot follows O's boundary clockwise. In part F, a previously stored N_{CC} point is visible and the robot drives to it. In part T, the robot evaluates the path lengths of the clockwise and counter-clockwise directions as illustrated. It concludes that the counter-clockwise direction is shortest. In parts H, I, J, K and L, the robot continues counter-clockwise boundary following. In part M, N_C* is visible. This is to be differentiated against N_C* being visible in part L since in part L N_C* is not in the boundary following direction but on a perceived separate obstacle. This implies that the target is unreachable. Part N illustrates the final path.

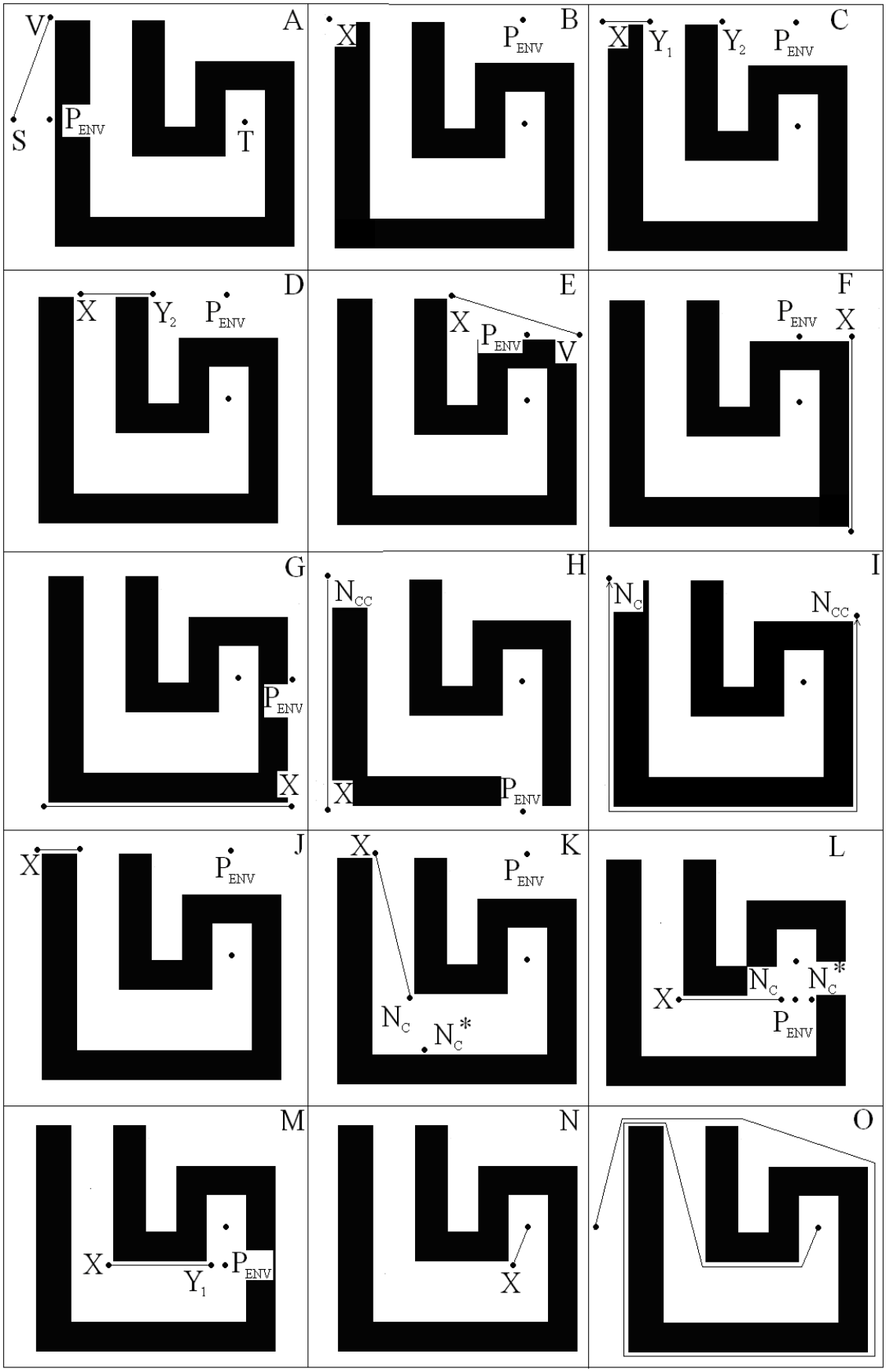

Figure 6-11 SensorBug example

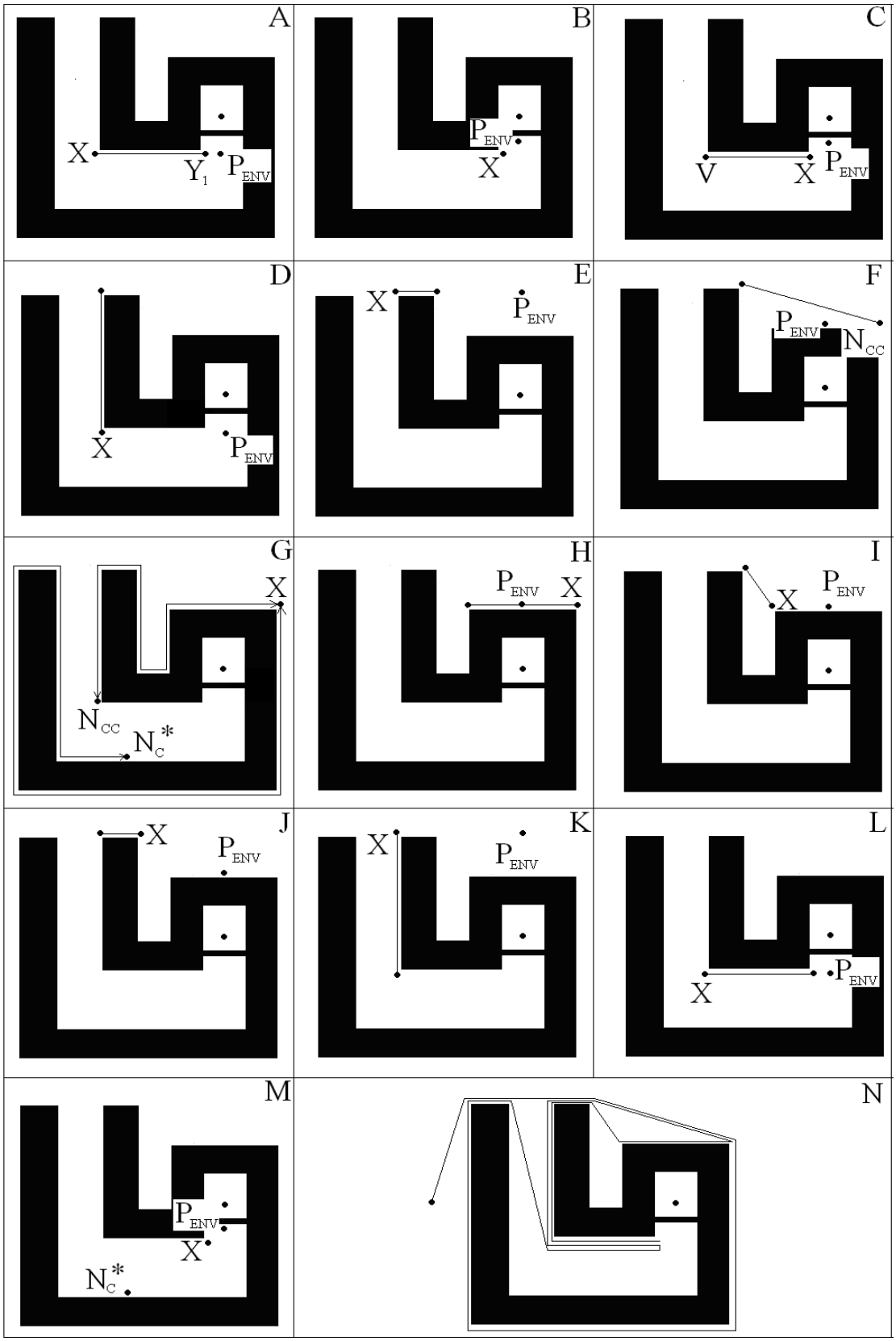

Figure 6-12: SensorBug example

6.8 Termination Proof

All Bug algorithms have proven, using reasoned arguments, their ability to meet the termination requirements with finite path length. There are two parts to SensorBug's proof. The first part shows SensorBug always terminates. The second part shows that the robot will always terminate correctly.

For this first part, the proof's structure is illustrated in Figure 6-13. In the center is the statement which is to be proved, that is, SensorBug always terminates with finite path length. Path length is the sum of path lengths of each "boundary following" and "moving to target" segment. Hence, there are three sub-theorems which are used: 1) the number of "boundary following" and moving to target" segments are finite, 2) the path length of each "boundary following" segment is finite and 3) the path length of each "moving to target" segment is finite. Each sub-theorem in turn has its own sub-theorems which can be found on Figure 6-13. Firstly, however, it should be shown that SensorBug always terminates when Q is null as illustrated in Figure 6-4.

Theorem 2: *If Q is null, SensorBug will always terminate correctly.*
Proof: If no vertices exist, the robot must be completely surrounded by O. Otherwise, an aperture would exist on which a vertex can be found. The target can either be inside the enclosure or outside the enclosure. If the target is outside the enclosure, the robot is never able define a leave point because there are no convex points within the enclosure. Therefore, it will circumnavigate the enclosure and correctly conclude that the target is unreachable.

If the target is inside the enclosure, the line [S,T] should not intersect any part of O. If the straight line [S,T] is intersected by O, then a vertex must be formed to ensure a path to T is still available. Given that the straight line [S,T] is not intersected by O, then the robot remains in "moving to target" mode until T is reached with P_{ENV} never residing on O. ■

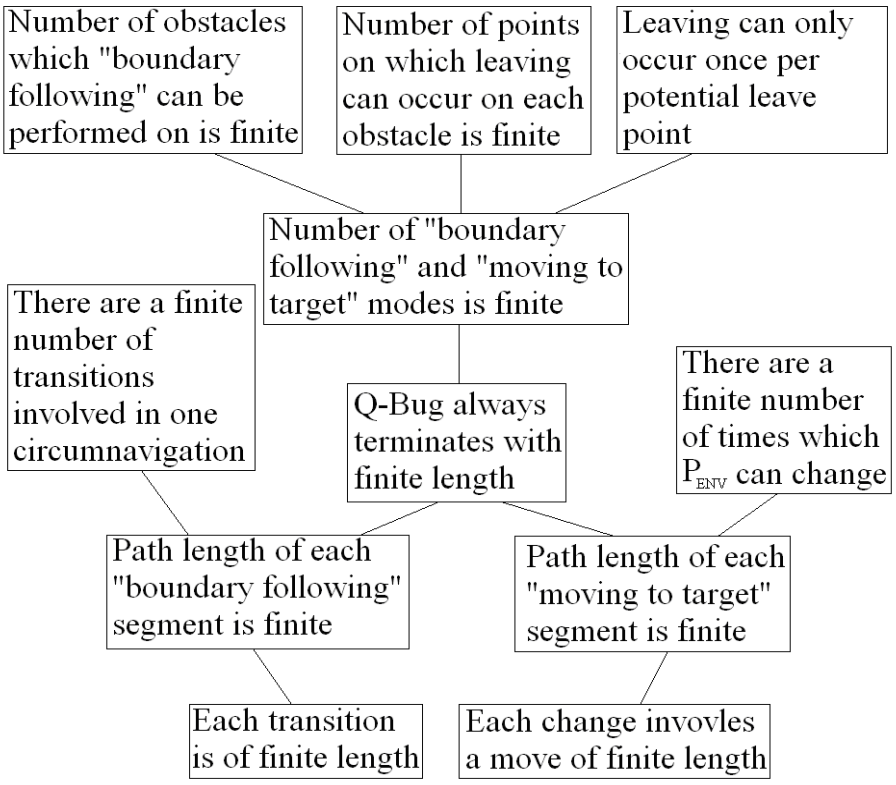

Figure 6-13: Proof structure for first proof section

Theorem 3: *The number of obstacles which the robot can perform "boundary following" on is finite*

Proof: In "moving to target" mode, $d(P_{ENV},T) \leq d(X,T)$ must hold since X is considered a member of the visible environment. Hence, d(P,T) can only decrease or remain constant in "moving to target" mode. In "boundary following" mode, P can only be replaced if another point along O's boundary is closer to the target than P. Therefore, d(P,T) can only decrease during SensorBug's execution.

Consider the "shrinking-disc" centered at T with radius d(P,T). Hit points (and therefore their corresponding "boundary following" segments) can only begin on obstacles which intersect or are contained in that circle because P_{ENV} must be inside the "shrinking" disc. According to the initial assumptions, there can only be a finite number of obstacles intersecting or contained within the "shrinking-disc". As SensorBug is executed, the radius decreases and no new obstacles can be added. ■

Theorem 4: *The number of points in Q associated with V on a particular O is finite.*

Proof: Consider the three types of transitions which can occur to travel from one point in Q to the next as illustrated in Figure 6-2. In one complete circumnavigation, the number of vertex transitions is less than or equal to the number of vertexes on O. There should be a finite number of vertices since O is assumed to be polygonal. The number of occlusion transitions is less than or equal to the number of vertices inside O's convex hull. The convex hull is of finite area and hence this quantity should be finite. The number of out-of-range transitions can be determined by

$$\sum_i \lfloor R / L_i \rfloor$$

where L_i represents the length of the i^{th} side on polygonal O.

Since the number of transitions is finite, it follows that the number of points which the robot stops to retrieve P_{ENV} is also finite and hence the number of points on which leaving can occur is also finite. ■

Given Theorem 4, it is imperative that the robot only leaves at points in Q. With multiple stored points, the driving to the N_{CC} point must be conducted such that in the subsequent clockwise or counter-clockwise segment, only

points in Q can evaluate the leaving condition. Hence, the reason why the robot must visit N_{CC}, which will always be a vertex, before making the decision.

Theorem 5: *The leaving rule can only hold once per point in Q.*

Proof: If the leaving rule holds at a particular X in Q, then during the subsequent "moving to target" segment, P is updated to the P_{ENV} which caused the leaving rule to hold. Therefore, if the robot visits X again and retrieves P_{ENV}, $d(P_{ENV},T) \geq d(P,T)$ will necessarily hold and the leaving condition can never hold. ■

Corollary 1: *The number of "boundary following" and "moving to target" segments is finite because leaving can occur only finitely many times.*

Theorem 6: *There are a finite number of transitions involved in one "boundary following" segment.*

Proof: If the robot has visited V to align itself with Q, then Theorem 4 can be used to prove finite transitions for the circumnavigation which follows. When the robot has just begun "boundary following", consider the finite area A formed by the straight lines $[X,P_{ENV}]$, [X,V] and along O's boundary from P_{ENV} to V as illustrated in Figure 6-14. A finite area contains finitely many obstacles, and each obstacle contains a finite number of vertices, so there can only be a finite number of vertices associated with A.

Whilst driving to V, the robot can only drive inside A. This can best be observed by considering two cases. First, consider if V is visible at X. Then, the robot will drive straight towards V. Second, consider if only P_{ENV} is visible. In that case, the robot will drive straight to P_{ENV} and then follow

O's boundary to V. Anything in the middle will involve the robot traveling inside A.

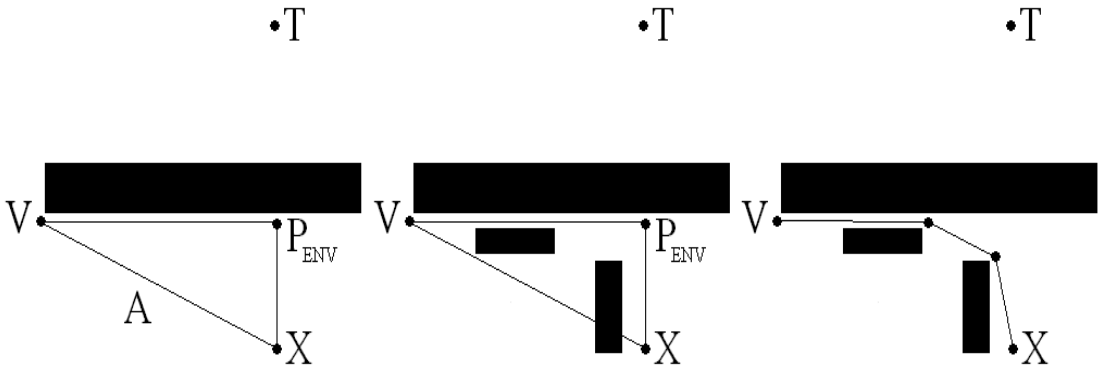

Figure 6-14: Illustration of A and the robot's subsequent maneuvers to V

Since the number of vertices is finite and the robot can only drive inside A, then the number of vertex and occlusion transitions is bounded by the number of vertices in A. The number of out-of-range transitions is bounded by $\lfloor R / d_{PATH}(P_{ENV}, V) \rfloor$ ■

Theorem 7: *Each transition is of finite length.*

Proof: For a robot with finite R, a transition cannot occur if it is not visible. Hence, the length of each transition is bounded by R. For a robot with infinite R, each transition is bounded by the length to the vertex causing either the vertex or occlusion transition. This length is finite because O's perimeter is finite. ■

Corollary 2: *The path length of each "boundary following" segment is finite because it is composed of a finite number of transitions, each of which have finite length.*

Theorem 8: *During any "moving to target" segment there is a finite number of times for which P_{ENV} can change.*

Proof: At any given X, if the line segment [X,T] is free of obstacles, then there can be at most $\lceil R/d(X,T) \rceil$ number of P_{ENV} changes. If R is infinite then T is visible and no changes are needed.

However, if an obstacle obstructs [X,T], then for P_{ENV} to not lie on O, a deviation is necessary. Consider the vertex V associated with such a deviation as illustrated in Figure 6-15. P_{ENV} is always associated with V because that association will always yield a P_{ENV} which is closer to the target than any other P_{ENV}' which is not associated with V. Such a situation is illustrated in Figure 6-15.

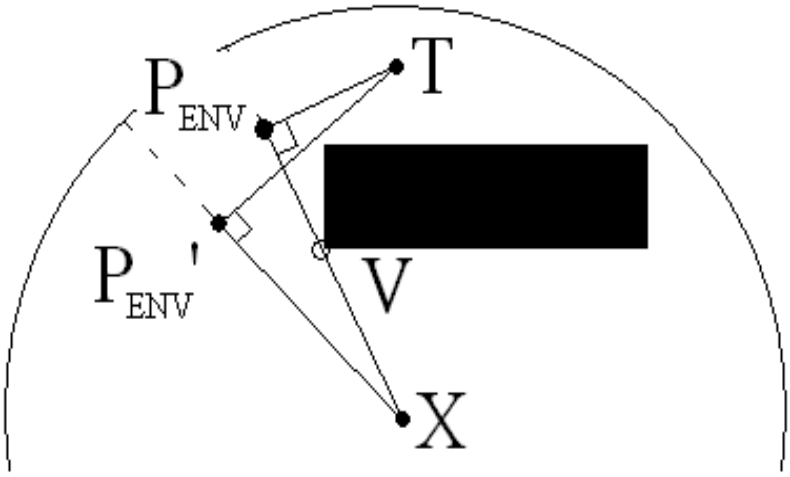

Figure 6-15: P_{ENV} must always be associated with a vertex V if deviation from [X,T] has occurred and P_{ENV} does not lie on O

Since P_{ENV} represents the minimal point in the environment and P_{ENV} is not on an obstacle, it follows that $d(V,T) > d(P_{ENV},T)$ must always hold. P is assigned to P_{ENV} and this means that the "shrinking-disc" will exclude V. Any future P_{ENV} retrieved must fall inside the "shrinking-disc" because the current P_{ENV} is always visible at the point where P was reassigned to P_{ENV}. This argument implies that each V can be associated with a P_{ENV} at most once. Given that the number of polygonal obstacles which intersect or are

inside the "shrinking-disc" is finite, the number of vertices is finite and therefore the number of times which P_{ENV} can change is also finite. ■

Theorem 9: *Each change of P_{ENV} produces a path of finite length.*

Proof: For a robot with infinite range sensors and P_{ENV} the maximum path length from X to P_{ENV} is $\sqrt{d(X,T)^2 - d(P_{INF},T)^2}$ where P_{INF} is the point on the infinitely extended line starting from X to P_{ENV} which is closest to the target. The lines $[X,P_{INF}]$ and $[P_{INF},T]$ should always be perpendicular. Hence a right-angled triangle can be formed between the line segments $[X,P_{INF}]$ and $[P_{INF},T]$. Then Pythagoras Theorem can be applied to achieve the final result. This is illustrated in Figure 6-15 where $P_{INF} = P_{ENV}$.

For a robot with finite range sensors, each segment's path length cannot exceed R. ■

Corollary 3: *The path length of each "moving to target" segment is finite because it is composed of a finite number of P_{ENV} changes, each of which have finite length.*

Corollary 4: *Given corollary 1, 2 and 3, it follows that the path length of SensorBug is finite because there are a finite number of "boundary following" and "moving to target" segments each of which have finite length.*

This next part of the proof involves showing that SensorBug always terminates with the correct result. Whilst corollary 4 implies that a result will always be achieved, this part shows the correct result will always be achieved. There can only be two results, the target is unreachable or the

target is reached. If the target is unreachable then the robot cannot reach it. However, if the target is reachable then it could incorrectly conclude that the target is unreachable if it circumnavigates O. The following theorems prove that it should never occur with SensorBug.

Proposition 1: *For every obstacle, there is always a non-empty and finite set of points C for which the distance to target is minimal compared to the rest of the obstacle. Figure 6-16 illustrates some examples.*

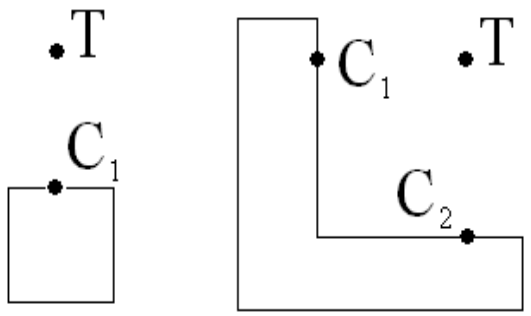

Figure 6-16: C illustrated for two obstacles

Proposition 2: *The line segment from any C_i to T does not intersect O.*

Proposition 3: *Any point in the line segment from C_i to T (excluding C_i itself) is closer to the target than any point on O.*

Theorem 10: *During "boundary following" of a particular O, the minimum value of d(P,T) is $d(C_i,T)$ for a C_i on O.*

Proof: Firstly, for the robot to perform "boundary following" on O, P_{ENV} must have resided on some point on O. Hence, initially $d(P,T) \geq d(C_i,T)$ must hold. Since only points on O are eligible to replace P, the minimum value which d(P,T) can achieve is $d(C_i,T)$ on that particular O. ■

Theorem 11: *All the line segments from C_i to T are at least partially visible from at least one point in the subset of Q associated with the starting vertex V.*

Proof: For a vertex transition, consider the robot's visible environment at both X and N_C. Any line originating from a point on O within X to N_C inclusive will fall inside the combined visible environment. Similarly, for an occlusion transition, the any line originating from X to N_C* will be visible from the combined visible environment at X and N_C. This also holds for an out-of-range transition. ■

Theorem 12: *If the robot acquires P_{ENV} at X where $d(X,T) < d(C_i,T)$, the leaving condition will always hold at X.*

Proof: At any point where P_{ENV} is retrieved, X is considered part of the visible environment, hence the relationship $d(P_{ENV},T) \leq d(X,T)$ always holds. Given the assumption $d(X,T) < d(C_i,T)$, it follows that $d(P_{ENV},T) < d(C_i,T)$. Theorem 10 suggests that $d(P,T) \geq d(C_i,T)$. Hence, $d(P_{ENV},T) \leq d(P,T)$, the leaving condition, must be hold at X. ■

Theorem 13: *If a C_i to T line segment is partially visible, the leaving condition will always hold.*

Proof: Assume that all points in Q are such that $d(Q_i,T) \geq d(C_i,T)$. If one point in Q does not satisfy that criterion, then Theorem 12 applies and the robot leaves O as required. Theorem 11 shows that at least one point in Q will have a C_i to T line segment partially visible. At that point, C_i is visible and $d(X,T) \geq d(C_i,T)$ holds. Furthermore, proposition 3 implies that if a C_i to T line segment is visible then there exists a point C_i* on the line C_i to T such that $d(Ci,T) > d(C_i*,T)$. Hence, the relationship $d(X,T) \geq d(Ci,T) > d(C_i*,T)$ holds. Since C_i* is visible and $d(X,T) > d(C_i*,T)$ holds, it follows that $d(P_{ENV},T) \leq d(C_i*,T)$. Theorem 10

suggests that $d(P,T) \geq d(C_i,T)$ and consequently $d(P,T) > d(C_i^*,T)$. Hence, $d(P_{ENV},T) \leq d(P,T)$, the leaving condition, must be hold at X. ■

Given Theorems 10, 11, 12 and 13, it is imperative that any line originating from O is visible from at least one of the members of Q associated with every V on O. "Boundary following" with multiple stored points has been designed specifically to ensure that this holds. If an N_{CC} point is visible, the robot would have tested the leaving condition at X before driving to N_{CC}. Similarly, if the robot is executing step 5 or step 6 then it tests the leaving condition at X before concluding that T is unreachable. These measures ensure that the C_i to T line is visible for an obstacle which does not prevent termination.

6.9 Suggested Implementation Strategies

It is desirable to minimize the range sensor usage at each point in Q. Whilst, the boundary following part must be performed, a sub-algorithm "getPenv()" can be devised to minimize the scanning required to retrieve P_{ENV}. At a higher level of abstraction, "getPenv()" firstly checks to see robot is capable of scanning inside the "shrinking-disc" given its current position and R. If incapable, then no attempt is made. If it is capable, then scanning should begin directly at T. If $d(X, P_{ENV}) = R$ and P_{ENV} lies on [X,T] then obviously that is the minimum point in the environment and no further scanning should occur. Otherwise, scanning should "fan out" until scanning inside the "shrinking-disc" is no longer possible.

"getPenv()" assumes an angular resolution of AngleRes and that subroutines "isInsideDisc()" and "getMinPOnLine" are available. These

will be explored later. Pseudo code for "getPenv()" is shown in Figure 6-17.

```
Subroutine getPenv(X, R, T, P)
        scanAngle = 0
        P_ENV = X
        If d(X,T) > R+d(P,T)
                return P_ENV
        Else
                While isInsideDisc(scanAngle, P_ENV, P, X, R, T)
                        P_LINE = getMinPOnLine(scanAngle)
                        if d(P_LINE,T) < d(P_ENV,T)
                                P_ENV = P_LINE
                        end if
                        scanAngle = scanAngle + AngleRes
                end while
                return P_NEW
        end if-else
```

Figure 6-17 Suggested code for getPenv

Subroutine "getMinPOnLine(scanAngle)" returns the closest point to the target, P_{LINE}, on the left and right scan lines specified by "scanAngle". "scanAngle" is always non-negative and measured with respect to the line [X,T]. For instance, Figure 6-18 part A shows that if "scanAngle" is 0 then P_{LINE} is the intersection of the line [X,T] with O. Figure 6-18 part B shows that as "angle" increases then P_{LINE} is the point where either the left or right line intersects the obstacle since they are equidistant. Figure 6-18 part C

shows that P_{LINE} occurs on the right where the lines [X, P_{LINE}] and [P_{LINE}, T] are perpendicular.

Subroutine "isInsideDisc(scanAngle, P_{ENV}, P, X, R, T)" checks if any point on the scan line parameterized by scanAngle, X and R is inside the disc centered at T with a radius the smaller of d(P_{ENV},T) and d(P,T). In other words, the subroutine checks if a P_{LINE} along the scan line could possibly replace the smaller of P_{ENV} and P. To implement this, perform getMinPOnLine(scanAngle) on the scanline assuming that no obstacles are visible and therefore without actually taking a scan. Then, if d(P_{LINE},T) < d(min(P_{ENV},P),T), a potential replacement for P_{ENV} or P is possible.

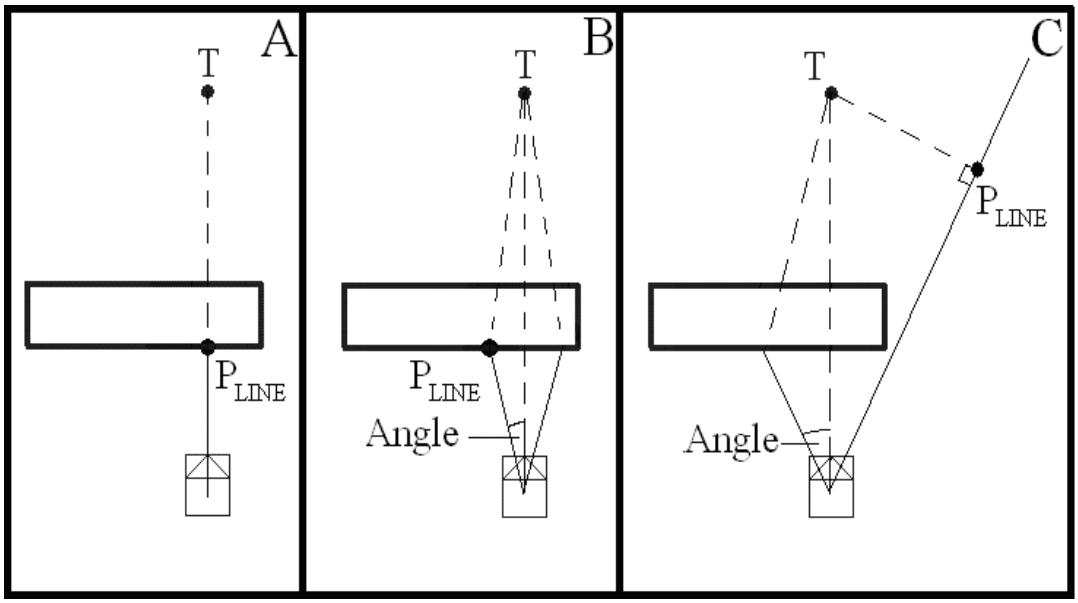

Figure 6-18: The robot calls "getMinPOnLine" with varying "angle" and obtains P_{LINE} at different locations

6.10 Conclusion

This thesis presented SensorBug which solves the Bug problem for an environment of polygonal obstacles. To allow the robot to gather data only at fixed intervals necessarily requires that the environment be made of

polygonal obstacles. Otherwise, a curve on the convex hull will always require continuous data gathering for the robot to follow its boundary. Another solution, which could be further investigated, is to allow the robot to follow the obstacle's boundary from a distance. This may allow non-polygonal obstacles to be included.

It may be tempting to compare SensorBug's path length with other Bug algorithms but recall that a fundamental property of Bug algorithms is making arbitrary decisions in spite of the uncertainty. For example, when the robot needs to follow an obstacle, it does not know whether to do so in a clockwise or counter-clockwise fashion. A lucky guess could reduce path length. Therefore, it is quite meaningless to compare algorithm path lengths when an environment could be constructed to favor one algorithm over another. This was quite well observed in [1,62] where if the environment had many intersections with the M-line, Bug2 fared much worse than Bug 1.

One of the features of SensorBug is that only vertexes, occlusions and out-of-range transitions are required except when a previously stored N_{CC} point is observed. These are all essential to boundary following. It was decided that detecting vertices, occlusions and out-of-range transitions which are not associated with the V would be too complicated and not worth the potential benefits. These benefits include a simpler model of Q where the obstacle is effectively delimited by vertices and a shorter path length when a previously stored N_{CC} point is observed.

Chapter 7

Summary and Significant Findings

7.1 Summary

Bug algorithm performance varies greatly depending on the environment. TangentBug produces the shortest path in environments with wider spaces that allow it to make use of its range sensors. Here, TangentBug can drive directly towards a vertex whereas other algorithms have to rely on wall following. The second shortest path in environment A (Figure 2-4) was achieved by DistBug, because it uses range sensors to immediately detect that the target is visible. In environment B (Figure 2-6), Rev2 produced the shortest path, because the alternative wall following strategy minimized wall-following paths globally. In environment C (Figure 2-8), DistBug produced the second shortest path because its range sensor allows the robot to leave an obstacle earlier. In environment D (Figure 2-10), LeaveBug and Bug1 tied for the second shortest path, as the environment required one continuous circumnavigation.

As for implementation complexity, we subjectively ranked the Bug algorithms from simple to complex as: Class1, Bug2, Bug1, OneBug, LeaveBug, DistBug, Alg2, Rev2, Alg1, Rev1 and finally TangentBug.

Simulations were conducted both in a perfect, noise-free world, as well as under more realistic noise settings with small errors in sensor reading and localization. In simulation runs with noise, we encountered situations where algorithms did not terminate, terminated incorrectly or terminated at

an inadequately large distance from the target. Although performance comparisons under noise were not the main focus, it needs to be noted that Bug algorithms in general do not exhibit fault-tolerance properties, which are the advantage of probabilistic navigation techniques [20, 24, 25].

It was established that "moving to target" and "boundary following" were essential to any Bug algorithm. Further, a Bug algorithm needs a method to ensure that its path will not be arbitrarily long. Five methods were identified and various Bug algorithms were classified into them. The disabling segments method was used to create a new Bug algorithm OneBug which guarantees only one hit point per disabling segment. Then, two more methods were created. The finite leave points was used to create the new Bug algorithm LeaveBug which guarantees only one leave point per enabling segment. Finally, the Q method is to be used as a basis for a new Bug algorithm SensorBug which will use range sensors in a simple fashion to guarantee termination.

Bug1+ was also developed to shorten the path lengths produced by Bug1. This raised a question for future research as to what the Bug algorithms should do when an obstacle is completely circumnavigated. Clearly, if the robot is given full information about an obstacle then there are much better ways in which it can leave.

TangentBug produces the shortest path because of the LTG to assist it in finding shortcuts during the "boundary following" segment and to travel to V_S. DistBug has an advantage over Alg2 since it can use the PSD sensor to scan for points inside the disc centered at T with radius $d_{min}(T)$. Alg2 produces shorter paths than Bug2 because the robot is not constrained to

leave on the M-line. Bug1 produces the longest path because it must circumnavigate O.

Four new aspects of guide track following extending the Curv1 algorithm were explored. Firstly, self-intersections were permitted and Curv2 was developed to ensure termination. Curv2 works on the idea that a small obstacle can be placed on the intersection and this allows Curv1 to guarantee termination. Secondly, we explored whether Curv1 was the only algorithm which could guarantee termination without localization. It was shown that it is the only such algorithm. Thirdly, dynamic obstacles were introduced. These obstacles can change status during the robot's journey. It was found that without significant restrictions, termination cannot be guaranteed. Lastly, environments which contain multiple trails were explored. It was shown that a unique pairing between start and targets was always achieved and hence termination is guaranteed. If a particular start/target pair is desired, a non-negative Z value can be associated with particular inflows to intersections. Then, Curv3 can be applied to achieve the desired pairing. However, it was observed that the desired pairing may not be achieved if an obstacle does not allow the robot to reach the intersection.

7.2 Significant Additions to the Bug Family

Foremost, the development of SensorBug is a significant improvement because it reduces the amount of range sensor data which needs to be gathered. In SensorBug, data only needs to be gathered on the blocking obstacle and on the closest point to the target at the current location. It also reduces the frequency at which sensor data needs to be gathered, restricting it to three types of transition points. SensorBug also incorporates the

previously stored points concept which was first suggested in Alg1 and Alg2.

The analysis of Bug algorithm termination is also important because it allows consideration of the "crux" of the Bug algorithm, independent of all the other nice features which an algorithm may possess. It was found that there are currently six methods which exist in current algorithm. Further analysis of these methods produced OneBug, MultiBug and LeaveBug. These algorithms incorporate the methods outlined and contain no frills or extra features. A seventh method was found, but it relies on the obstacles being polygonal and it was used in the formation of SensorBug. Future Bug algorithms will utilize one of the methods but for a truly unique new Bug algorithm, a new method must firstly be created.

A theoretical analysis was performed on Bug algorithms on semi-convex obstacles and this revealed that algorithms which could quickly leave were always going to outperform algorithms which were more conservative. This analysis has implications for the Bug algorithms on real robots since unknown environments may be offices, homes or outdoors where the vast majority of obstacles are semi-convex. Of the examined algorithms, the order was TangentBug, DistBug, Alg2, Alg1, Bug2 and Bug1.

7.3 Future Work

Future work can involve implementing and simulating all the Bug algorithms contained in Chapter 1. The results can be added to those in Chapter 2 and a more detailed study will be the result. Another area an improvement to the Bug1+ algorithm which will minimize the distance to target when full circumnavigation has been performed. Investigations on

Environments with multiple semi-convex obstacles can also be performed with many more Bug algorithms. Subsequent analysis is likely to yield a better understanding of Bug algorithm behaviour on this special class of obstacle. An upper bounds on SensorBug's path length can also be found. Although it has been shown that SensorBug's path length is finite, an upper bounds on path length would be preferable since it allows comparison to other Bug algorithms.

Bug algorithms can also be tested in simulation environments where localization error is introduced to see which are more robust. The Bug algorithms can be combined with Simultaneous Localization and Mapping or landmark recognition techniques to compensate for the introduced error.

References

[1] V. Lumelsky and A. Stepanov. *Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape.* Algorithmica**,** vol. 2, pp. 403-430, 1987.

[2] H. Noborio. *A path-planning algorithm for generation of an intuitively reasonable path in an uncertain 2-D workspace.* Proc. of the Japan-USA Symposium on Flexible Automation, Pages 477-480, July 1990

[3] 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

[4] 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 IEEE Int. Conf. on Decision and Control, Pages 1111-1119, 1991.

[5] I. Kamon and E. Rivlin. *Sensory-Based Motion Planning with Global Proofs.* IEEE Transaction on Robotics and Automation, Vol. 13, Pages 814-822, December 1997.

[6] I. Kamon, E. Rivlin and E. Rimon. *TangentBug: A range-sensor based navigation algorithm.* Journal of Robotics Research, vol. 17, no. 9, Pages 934-953, 1998.

[7] T Bräunl. *Embedded Robotics.* 2nd Edition, Springer, Berlin, 2006

[8] Y. Horiuchi and H. Noborio. *Evaluation of Path Length Made in Sensor-Based Path-Planning with the Alternative Following.* Proc. of the 2001 IEEE International Conference on Robotics and Automation, Pages 1728-1735, May 2001.

[9] H. Noborio, Y. Maeda and K. Urakawa. *Three or More Dimensional Sensor-Based Path-Planning Algorithm HD-1.* Proc. of the 1999 IEEE/RSJ International Conference on Intelligent Robotics and Systems, 1999.

[10] S. L. Laubach and J. W. Burdick. *An Autonomous Sensor-Based Path-Planner for Planetary Microrovers.* Proc. of the IEEE Int. Conf. on Robotics and Automation, Pages 347-354, 1999

[11] E. Magid and E. Rivlin. *CautiousBug: A Competitive Algorithm for Sensor-Based Robot Navigation.* Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Pages 2757-2762, 2004

[12] K. Kreichbaum. *Tools and Algorithms for Mobile Robot Navigation with Uncertain Localization.* Ph.D thesis, California Institute of Technology, 2006

[13] V. Lumelsky and S. Tiwari. *An Algorithm for Maze Searching with Azimuth Input.* Proc. of the 1994 IEEE Int. Conf. On Robotics and Automation, pp. 111-116, 1994

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

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

[16] V. J. Lumelsky and T. Skewis. *Incorporating range sensing in the robot navigation function.* IEEE Transactions on Systems, Man and Cybernetics, Vol. 20, Pages 1058-1068, 1990

[17] H. Noborio, R. Nogami and S. Hirao. *A New Sensor-Based Path-Planning Algorithm whose Path Length is Shorter on the Average.* Proc. of the 2004 Int. Conf. on Robotics and Automation, Pages 2832-2839, 2004

[18] H. Choset, K. Lynch, K. Hutchinson, G. Kantor, W. Burgard, L. Kavarki and S. Thrun. *Principles of Robot Motion: Theory, Algorithms and Implementations.* MIT, Cambridge, MA, 2005

[19] A. Sankaranarayanan and M. Vidyasagar. *A New Algorithm for Robot Curve-Following Amidst Unknown Obstacles, And a Generalization of Maze-Searching.* IEEE International Conference on Robotics and Automation, Pages 2487-2494, May 1992

[20] S. Thrun et al. *Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva.* The International Journal of Robotics Research, Pages 972-999, 2000

[21] H. Abelson and E. DiSessa. *The Turtle Geometry.* MIT Press, Cambridge, Pages 176-199, 1980.

[22] T. Tsumura. *Survey of Automated Guided Vehicle in Japanese Factory.* IEEE Conference on Robotics and Automation, 1986

[23] Kalman R. E, *A New Approach to Linear Filtering and Prediction Problems.* Transactions of the ASME – Journal of Basic Engineering. Vol. 82: Pages 35-45, 1960

[24] G. Dissanayake et al. *A Computationally Efficient Solution to the Simultaneous Localization and Map Building (SLAM) Problem.* IEEE International Conference on Robotics and Automation, Pages 1009-1014, April 2000

[25] T. Bailey. *Constrained Initialization for Bearing-Only SLAM.* IEEE International Conference on Robotics and Automation, Pages 1966-1971, September 2003

[26] J. Ng and T. Bräunl. *Performance Comparison of Bug Navigation Algorithms.* Journal of Intelligent Robotic Systems, Pages 73-84, April 2007

[27] J. Ng and T. Bräunl. *Robot Navigation with a Guide Track.* Fourth International Conference on Computational Intelligence, Robotics and Autonomous Systems, Pages 37-43, November 2007

[28] A. Yufka and O. Parlaktuna. *Performance Comparison of Bug Algorithms for Mobile Robots.* 5th International Advanced Technologies Symposium, May 2009.

[29] Mobile Robots Inc, ActivMedia Robotics, LLC, http://www.mobilerobots.com/, 2008.

[30] V. Lumelsky and P. Stepanov. *Effect of Uncertainty on Continuous Path Planning for an Autonomous Vehicle*. Proceedings of the 23rd Conference on Decision and Control, pp. 1616-1621, December 1984.

[31] H. Noborio, Y. Maeda and K. Urakawa. *A comparative study of sensor-based path-planning algorithms in an unknown maze.* Proc. of the IEEE/RSI Int. Conf. on Intelligent Robots and Systems 2, pp. 909–916, 2000.

[32] C.H. Chiang, J.S. Liu and Y.S. Chou. *Comparing Path Length by Boundary Following Fast Matching Method and Bug Algorithms for Path Planning.* Opportunities and Challenges for Next-Generation Artificial Intelligence, Springer, pp. 303-309, 2009

[33] J. Sethian. *Level Set Methods.* Evolving interfaces in geometry, fluid mechanics, computer vision, and materials science. Cambridge University Press, Cambridge, 1999

[34] C.H. Chiang and J.S. Liu. *Boundary Following in Unknown Polygonal Environment Based on Fast Marching Method.* IEEE International Conference on Advanced Robotics and its Social Impacts, 2008

[35] S LaValle. *Planning Algorithms.* University of Illinois Press, 2003

[36] J. Antich and A. Ortiz. *Bug2+: Details and formal proofs*. Technical Report A-1-2009, University of the Balearic Islands, 2009. The paper can

be downloaded from http://dmi.uib.es/~jantich/Bug2+.pdf

[37] J. Antich, A. Ortiz and J. Minguez. *ABUG: A Fast Bug-derivative Anytime Path Planner with Provable Suboptimality Bounds.* The 14th International Conference on Advanced Robotics. ICAR'09, Munich (Germany), 2009

[38] J. Antich, A. Ortiz and J. Minguez. *A Bug-Inspired Algorithm for Efficient Anytime Path Planning.* International Conference on Intelligent Robots and Systems. IROS'09 St. Louis (USA), 2009

[39] J. Pearl. *Heuristics*. Addison-Wesley, 1984.

[40] R. Zhou and E. Hansen. *Multiple sequence alignment using A**. Proc. of the National Conference on Artificial Intelligence, 2002.

[41] J. Antich and A. Oritz. *Bug-based T^2: A New Globally Convergent Potential Field Approach to Obstacle Avoidance*. Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 430-435, October 2006.

[42] J. Antich and A. Ortiz. *Extending the potential fields approach to avoid trapping situations.* Proceedings of the Intl. Conference on Intelligent Robots and Systems, pp. 1379–1384, 2005.

[43] Y. Koren and J. Borenstein. *Potential field methods and their inherent limitations for mobile robot navigation*. Proceedings of the Intl. Conference on Robotics and Automation, pp. 1398–1404, 1991.

[44] O. Khatib. *Real-time obstacle avoidance for manipulators and mobile robots.* Intl. Journal of Robotics Research, vol. 5, no. 1, pp. 90–98, 1986.

[45] S. Charifa and M. Bikdash. *Adaptive Boundary-Following Algorithm Guided by Artificial Potential Field for Robot Navigation*. IEEE Workshop on Robotic Intelligence in Informationally Structured Space, pp. 38-45, March 2009.

[46] C. O'Dunlaing and C. K. Yap. *A retraction method for planning the motion of a disc*. Journal of Algorithms, vol. 6, pp. 104-111, 1985.

[47] D. T. Lee and R. L. Drysdale. *Generalization of Voronoi diagrams in the plane*. SIAM Journal on Computing, vol. 10, pp. 73-87, 1981.

[48] D. Leven and M. Sharir. *Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams*. Discrete and Computational Geometry, vol. 2, pp. 9-31, 1987.

[49] M. Sharir. *Algorithmic motion planning*. Handbook of Discrete and Computational Geometry, 2nd Ed., J. E. Goodman and 1. O'Rourke, Ed. New York: Chapman and Hall/CRC Press, pp. 1037-1064, 2004.

[50] N. J. Nilsson. *A mobile automaton: An application of artificial intelligence techniques*. International Joint Conference on Artificial Intelligence, Washington, DC., pp. 509-520, May 1969.

[51] J. S. B. Mitchell. *Shortest paths and networks*. Handbook of Discrete and Computational Geometry, 2nd Ed., J. E. Goodman and J. O'Rourke, Ed. New York: Chapman and Hall/CRC Press, pp. 607-641, 2004.

[52] M. Mikson and A. Russell. *Close Range Inspection Using Novelty Detection Results.* Intelligent Robotics and Applications, Springer, pp. 947-956, 2009.

[53] W. Ko, L. Senevieatne, S. Earles. *Space representation and map building-A triangulation model to pathplanning with obstacle avoidance*. Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, pp. 2222–2227 (1993)

[54] G. Foux, M. Heymann, A. Bruckstein. *Two-dimensional robot navigation among unknown stationary polygonal obstacles*. IEEE Transactions on Robotics and Automation 9, 96–102 (1993)

[55] R. Jarvis. *Collision-Free Trajectory Planning Using the Distance Transforms*. Mechanical Engineering Trans. of the Institution of Engineers Australia, September 1985

[56] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars *Probabilistic roadmaps for fast path planning in high dimensional configuration spaces*. IEEE Trans. on Robotics and Automation, vol. 12, pp. 566-580, 1996.

[57] R. Geraerts and M. H. Overmars. *A Comparative Study of Probabilistic Roadmap Planners*. Workshop on the Algorithmic Foundations of Robotics, pp. 43-57, 2002.

[58] C. Lucas, V. Lumelsky and A. Stepanov. *Comments on "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment"*. IEEE. Transactions on Robotics and Automation, pp. 511-512, 1988.

[59] C. Yap. *Algorithmic Motion Planning*. Advances in Robotics, Vol. 1: Algorithmic and Geometric Aspects (J. T. Schwartz and C. IC. Yap Eds.), 1987.

[60] J. Schwartz and M. Sharir. *On the "Piano Movers" Problem: The Case o f a Two-dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers*. Comm. Pure Appl. Math., 36, 1983.

[61] T. Lozano-Perez. *Spatial Planning: A Configuration Space Approach*. IEEE Transactions on Computers, Vol. C-32, No. 2, 1983.

[62] V. Lumelsky and A. Stepanov. *Dynamic Path Planning For A Mobile Automaton With Limited Information On The Environment.* IEEE Transactions On Automatic Control, Vol. AC.31, No. 11, Nov. 1986.

[63] T. Skewis and V. Lumelsky. *Experiments with a Mobile Robot Operating in a Cluttered Unknown Environment.* International Conference on Robotics and Automation, pp. 1482-1487, 1992.

[64] I. Kamon, E. Rimon and E. Rivlin. *Range-Sensor Based Navigation in Three Dimensions.* International Conference on Robotics and Automation, pp. 163-169, 1999.

[65] J. Crowley. *Navigation for an intelligent mobile robot*. IEEE Journal of Robotics and Automation, pp. 31-41, 1985.

[66] M. Drumheller. *Mobile robot localization using sonar.* IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 325-332, 1987

[67] J. Leonard. *Directed sonar sensing for mobile robot navigation*. Ph.D thesis, University of Oxford, 1990.

[68] A. Alfes. *Sonar-based real-world mapping and navigation.* IEEE Journal of Robotics and Automation, 1987.

[69] H. Moravec and A. Alfes. *High resolution maps from wide angle sonar.* International Conference on Robotics and Automation, March 1985.

[70] V. Lumelsky. *A Comparative Study on the Path Length Performance of Maze-Searching and Robot Motion Planning Algorithms*. IEEE Transactions on Robotics and Automation, pp. 57-66, 1991.

[71] S. Kim, J. Russell and K. Koo. *Construction Robot Path-Planning for Earthwork Operations*. Journal of Computing in Civil Engineering, pp. 97-104, 2003.

[72] M. Trevisan, M. A. P. Idiart, E. Prestes and P. M. Engel. *Exploratory Navigation Based on Dynamical Boundary Value Problems.* Journal of Intelligent and Robotic Systems, vol. 45, no. 2, pp. 101-114, Feb. 2006.

[73] A. Masoud and S. Masoud. *Motion planning in the presence of directional and regional avoidance constraints using nonlinear, anisotropic, harmonic potential fields: a physical metaphor.* IEEE Transactions on Systems Man and Cybernetics pp. 705 - 723, Nov. 2002.

[74] V. Lumelsky. *Continuous Robot Motion Planning in Unknown Environment*. In Adaptive and earning Systems: Theory and Applications. (ed. K. Narendra), Premium-Press, 1986.

[75] V. Lumelsky. *Effect of Robot Kinematics on Motion Planning in Unknown Environment*. Proceedings of the 24th IEEE Conj. on Decision and Control. December 1985.

[76] V. Lumelsky. *Continuous Motion Planning in Unknown Environment for a 3D Cartesian Robot Arm*. IEEE International Conference on Robotics and Automation, San Francisco, April 1986.

[77] K. Sun and V. Lumelsky. *Computer Simulation of Sensor-Based Robot Collision Avoidance in an Unknown Environment*. Robotica. 1987.

[78] V. Lumelsky. *Dynamic Path Planning for a Planar Articulated Robot Arm Moving Amidst Unknown Obstacles*. Automatica, Vol.23 No.5, pp. 551-570, September 1987.

[79] V. Lumelsky. *Effect of Kinematics on Dynamic Path Planning for Planar Robot Arms Moving Amidst Unknown Obstacles*. IEEE Journal of Robotics and Automation. June 1987.

[80] V. Lumelsky and K. Sun. *Gross Motion Planning for a Simple 3D Articulated Robot Arm Moving Amidst Unknown Arbitrarily Shaped Obstacles*. Proc. IEEE International Conf. on Robotics and Automation. April 1987.

[81] V. Lumelsky. *On the Connection Between Maze-Searching and Robot Motion Planning Algorithms*. Proc. of the 24th Conference on Decision and Control, pp. 2270-2275, December 1988.

[82] V. Lumelsky and T. Skewis. *A paradigm for incorporating vision in*

the robot navigation function. IEEE Conference on Robotics and Automation, April 1988.

[83] O. Ore. Theory of Graphs. Providence, RI: American Mathematics Society, 1962.

[84] B. Bullock, D. Keirsey, J. Mitchell, T. Nussmeier. and D. Tseng. *Autonomous vehicle control: An overview of the Hughes project*. Proceedings of the IEEE Computing Society Conference on Trends and Applications, 1983.

[85] A. Thompson. *The navigation system of the JPL robot*. Proceedings of the 5th Joint International Conference on Artificial Intelligence, August 1977.

[86] H. Moravec. *The Stanford cart and the CMU rover.* IEEE Conference on Robotics and Automation, 1993.

[87] R. Langer, L. Coelho and G. Oliveira. *K-Bug, a new bug approach for mobile robot's path planning*. IEEE International Conference on Control Applications, pp. 403-408, October 2007.

[88] L. Sciavicco and B. Siciliano. *Modelling and Control of Robot Manipulators*. Springer, 2000.

[89] Y. Takahashi, T. Komeda, and H. Koyama. *Development of assistive mobile robot system: Amos*. Advanced Robotics, vol. 18, no. 5, 2004.

[90] J. Mclurkin. *Using cooperative robots for explosive ordnance disposal*.

Massachussets Institute of Technology. Artificial Intelligence Laboratory, 1995.

[91] M. Littman, T. Dean, and L. Kaelbling. *Markov games as a framework for multi-agent reinforcement learning*. Eleventh International Conference on Machine Learning. San Francisco, pp. 157-163, 1994.

[92] NASA. *Mars pathfinder project*. California Institute of Technology, Jet Propulsion Laboratory. Pasadena, CA, Tech. Rep., 1997.

[93] C. Petres, Y. Pailhas, P. Patron, Y. Petillot, J. Evans, D. Lane. *Path Planning for Autonomous Underwater Vehicles*. IEEE Trans. Robotics, vol. 23, no. 2, pp. 331-341, Apr. 2007.

[94] S. Ge, X. Lai, A. Mamun. *Boundary Following and Globally Convergent Path Planning Using Instant Goals*. IEEE Trans. on Systems, Man, and Cybernetics, Part B: Cybernetics, Vol. 35, No. 2, April 2005.

[95] Choset, H. *Coverage for robotics – A survey of recent results*. Annals of Mathematics and Artificial Intelligence 31, pp. 113–126, 2001.

[96] Y. Huang, Z. Cao and E. Hall. *Region filling operations for mobile robot using computer graphics*. Proceedings of the IEEE Conference on Robotics and Automation, pp. 1607–1614, 1986.

[97] M. Ollis and A. Stentz: *First results in vision-based crop line tracking*. IEEE International Conference on Robotics and Automation, 1996.

[98] S. Land and H. Choset. *Coverage path planning for landmine location*.

Third International Symposium on Technology and the Mine Problem, Monterey, CA (1998)

[99] C. Lee. *A framework of Adaptive T-S type Rough-Fuzzy Inference Systems (AFRIS)*. PhD Thesis, The University of Western Australia, 2009.

[100] M. Sarkar and B. Yegnanarayana. *Rough-Fuzzy Membership Functions.* IEEE International Conference on Fuzzy Systems in WCCI 1998, pp. 796-801, May 1998.

[101] M. Sarkar. *Rough-fuzzy functions in classification.* Journal of Fuzzy Sets and Systems, vol. 132, pp. 353-360, 2002.

[102] L. Zadeh. *Fuzzy Sets*. Information and Control, pp. 338-353, 1965.

[103] P. Kouchakpour. *Population Variation in Canonical Tree-Based Genetic Programming*. PhD Thesis, The University of Western Australia, 2008.

[104] L. Fogel, A. Owens and M. Walsh. *Artificial Intelligence through a simulation of evolution*. Biophysics and Cybernetics Systems, pp. 131-156, 1965.

Appendix

Implementing the Bug Algorithms on EyeSim

A.1 The EyeSim Simulation System

The EyeSim [7] simulation system allows code for robots to be written and simulated on a computer system. The programming language is C and the RoBios library allows the programmer to work with a high level interface. Once the code is written, it is compiled and loaded onto a robot in an artificial environment. When started, the robot in the simulator will behave according to the programmed code.

A.2 Common Modules

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 A-1 shows the common modules and the navigation module which can be altered for implementing a specific algorithm.

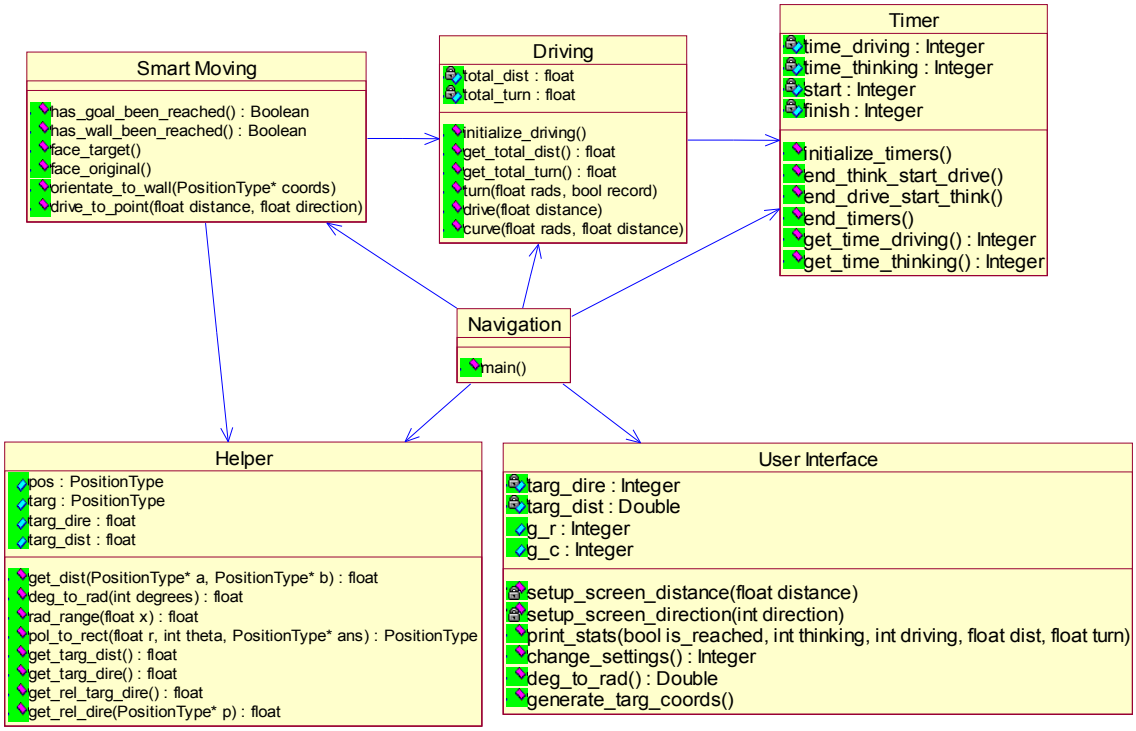

Figure A-1 The class diagram of the common modules

A.2.1 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 applications.

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.

The driving module handles all the functions associated with driving the robot and reporting the total distance moved and total angle rotated. The timer module records time taken processing navigation algorithms and time spent driving. The user interface module handles all user interfacing

including setup of screens and reporting of statistics.

A.2.2 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.

A.2.3 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 target. Figure A-2(a) shows the screen which allows the user to edit the distance to target and Figure A-2(b) shows the screen which allows the user to edit the direction to target.

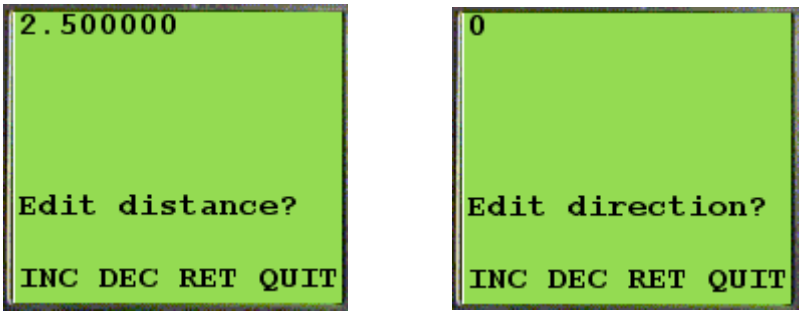

Figure A-2 (a) Left. The user can edit the distance to target. (b) Right. The user can edit the direction to target

The user interface module also displays the navigation results to the user. Figure A-3 shows the screens which appear when convergence is achieved. Figure A-3(a) shows computation and driving time, in milliseconds. Figure

A-3(b) shows distance traveled in metres and the rotation in radians. Figure A-3(c) shows the number of calls to the math library or process-state() if D* is run.

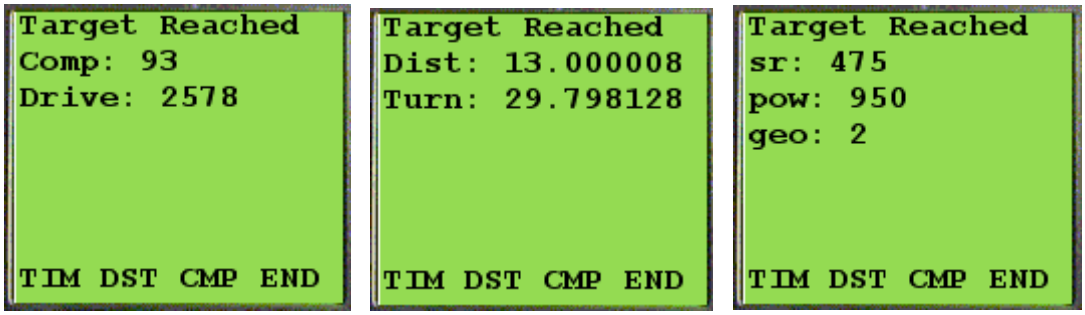

Figure A-3 (a) Left. Computation and driving times. (b) Centre. Total distance traveled and total rotation performed. (c) Right. Calls to the maths library or process-state() in D*.

A.2.4 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-dependent 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 A-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 A-4 The drive method

A.2.5 The Smart Moving Module

The smart driving module's role is to provide abstract driving functions as required by the main navigation algorithm. The has_goal_been_reached() method determines if the robot is currently at the target. The has_wall_been_reached() method determines if a wall has been reached. The face_target() method will rotate the robot such that it faces the target. The face_original() method will rotate the robot such that it faces the origin. The orientate_to_wall() method will rotate the robot such that it is parallel to the wall. The most complicated method, follow_the_wall(), is depicted in Figure A-5 and it drives the robot such that it follows the wall.

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 A-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 A-6(c). If not, the robot calls turn_and_move the robot turns and moves as shown in Figure A-6(b). After calling the above

methods robot aligns to the wall by calling the orientate_to_wall() method as depicted in Figure A-6(d).

```
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 A-5. The follow_the_wall method

A.3 Algorithm Implementation

A.3.1 Bug1 Implementation

The Bug1 algorithm is implemented by calling methods from the common modules as shown in figure A-7. It shows that Bug1 implements the “drive to target” and “follow the wall” states using the methods drive_to_target() and follow_wall_Bug1() respectively.

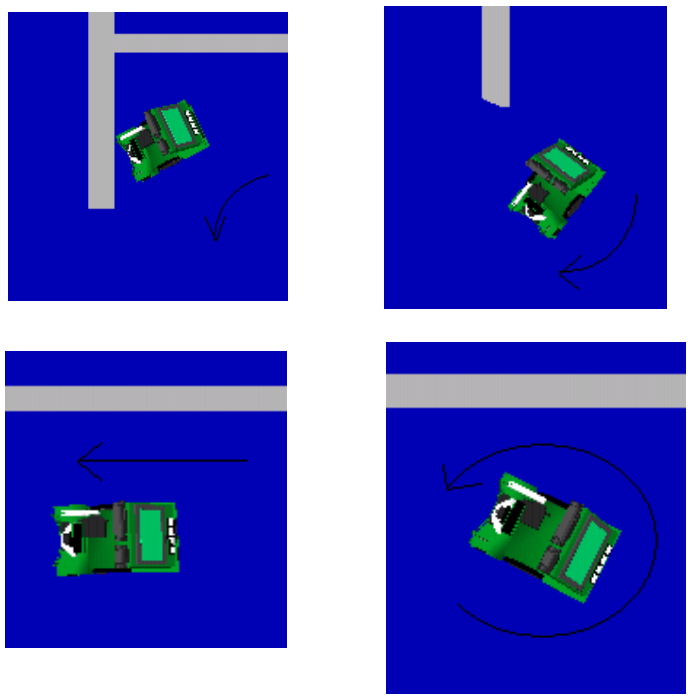

Figure A-6. (a) Top left. Turn_not_move(). (b) Top right. Turn_and_move() (c) Bottom left. Follow_wall_straight() (d) Bottom right. Orientate_to_wall()

```
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);
     }
}
```

Figure A-7(a) The drive_to_target() function orientates the eyebot to the target and drives towards it until either the target is reached or a wall is hit.

```
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(get_dist(&leave, &hit)<=TARG_ERROR){
          face_original();
          return TARGET_UNREACHABLE;
     }

     /*Determine the shortest route to the min point.  Then
     follow the wall to the min point minimizing travel*/
     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;
}
```

Figure A-7(b) This function follows the wall according to Bug1.

```
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 A-7(c) The Bug1 function drives the robot towards the target using the Bug1 algorithm

A.3.2 Bug2 Implementation

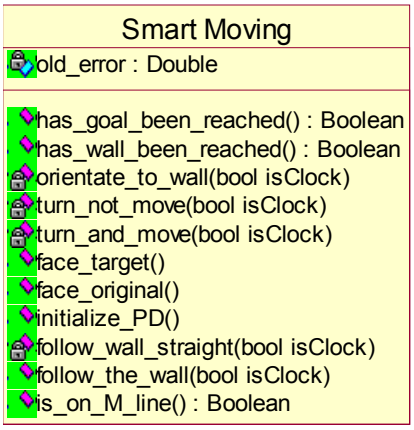

Figure A-8 The extended Smart Moving module for bug2

The Bug2 navigation class calls the common modules in a similar fashion to Bug1. 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 A-8. The rationale behind the is_on_M_line() function is discussed in chapter 3. The code is presented in Figure A-9:

```
/*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 A-9 The is_on_M_line() method

A.3.3 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 2.3.3, is reused. However, a new method, freespace(), needs to be created to determine F. Figure A-10 shows the updated Smart Moving module which includes the two new methods.

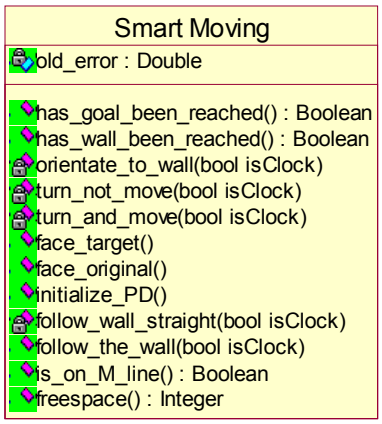

Figure A-10 The extended Smart Moving module for Alg1

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 A-11:

```
/*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 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 A-11 The freespace method

In this particular robot, there are 8 PSD sensors. Figure A-12 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.

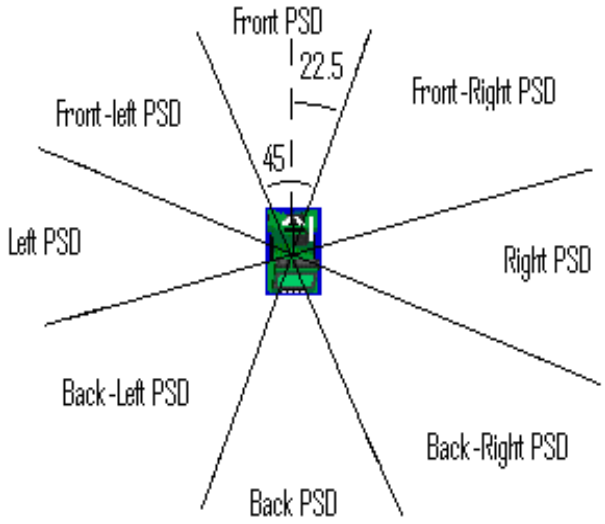

Figure A-12 Each PSD covers a 45 degree sector. Maximum rotation is 22.5 degrees

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 A-13.

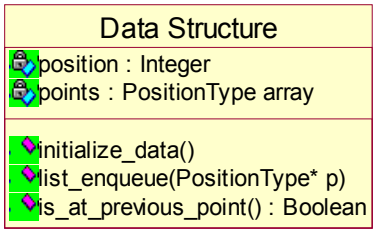

Figure A-13 The Data Structure Module

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 A-14(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 A-14(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.

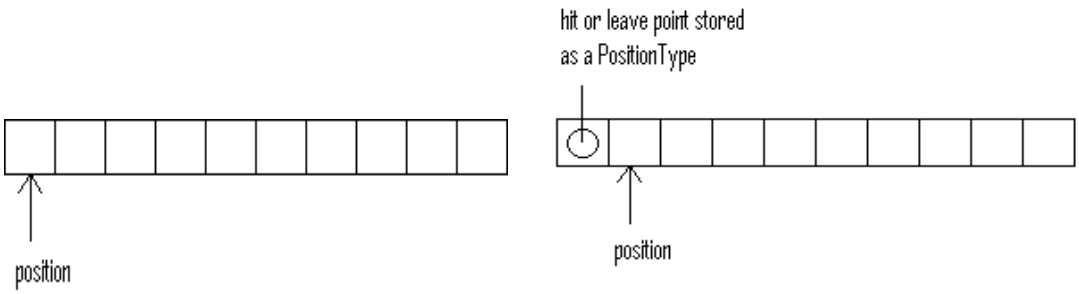

Figure A-14 (a) Left. The data structure after initialization. (b) Right. The data structure after list_enqueue() is called

A.3.4 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 2.3.4. It also uses the common modules to implement navigation states similar to the Bug1 implementation in section A.3.1

A.3.5 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.

A.3.6 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 data gathering.

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 A-15.

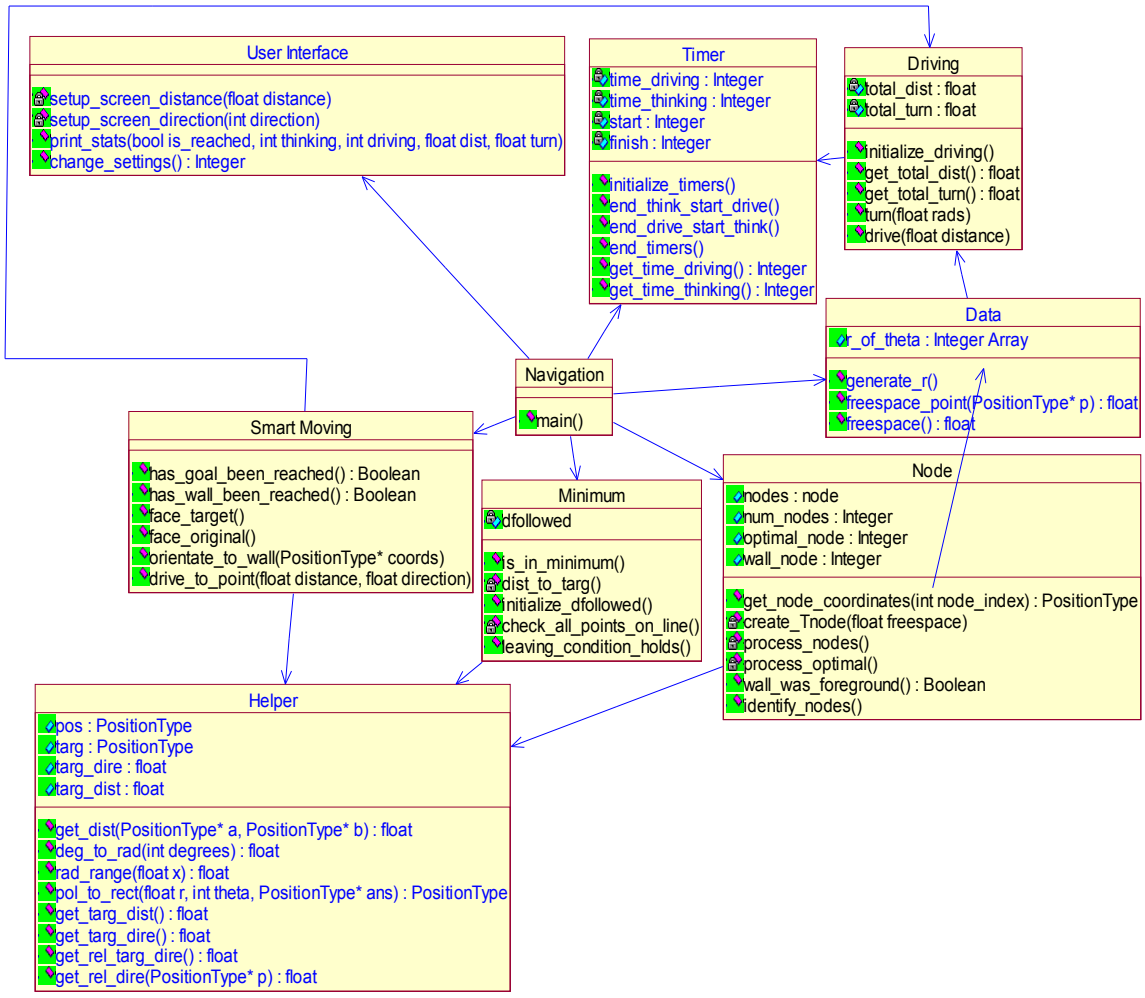

Figure A-15 The tangentbug class diagram

A.3.6.1 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 A-16 shows the division of sectors.

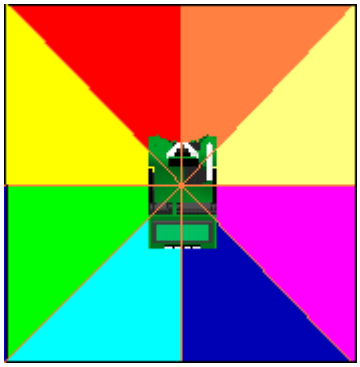

Figure A-16 Each PSD collects data in its sector

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 A-17 shows the source code which implements data gathering and figures A-18(a) and A-18(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_i
                  ndex] = PSDGet(psd[psd_index]);
            }
            turn(deg_to_rad(DEG_BET_SAMPLES), FALSE);
      }
      turn(-deg_to_rad(360/NUM_PSD), FALSE);
}
```

Figure A-17 The generate_r method

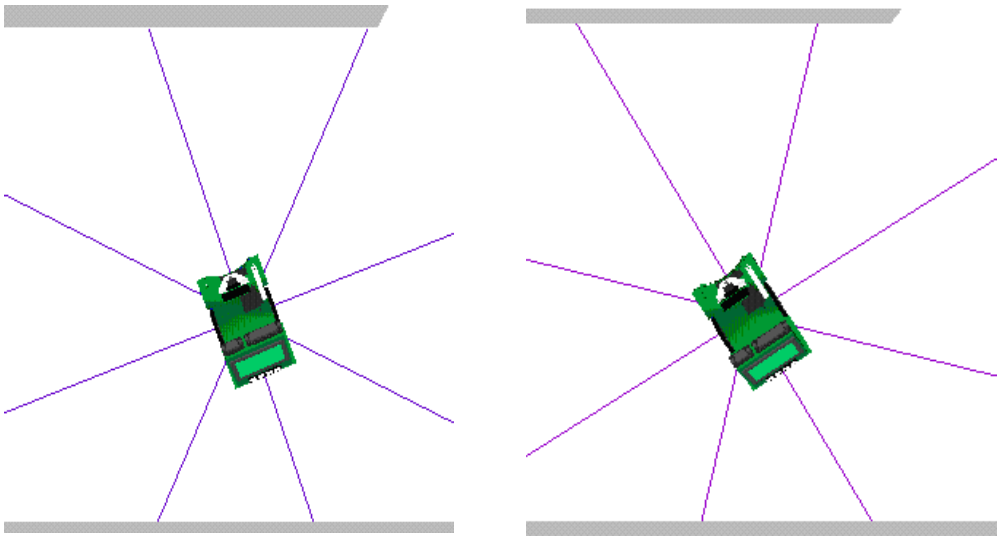

Figure A-18 (a) Left. The robot gathers data from all 8 PSD sensors (b) Right. After a 3 degree rotation, the robot gathers data from 8 PSD sensors again

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.

A.3.6.2 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 A-19.

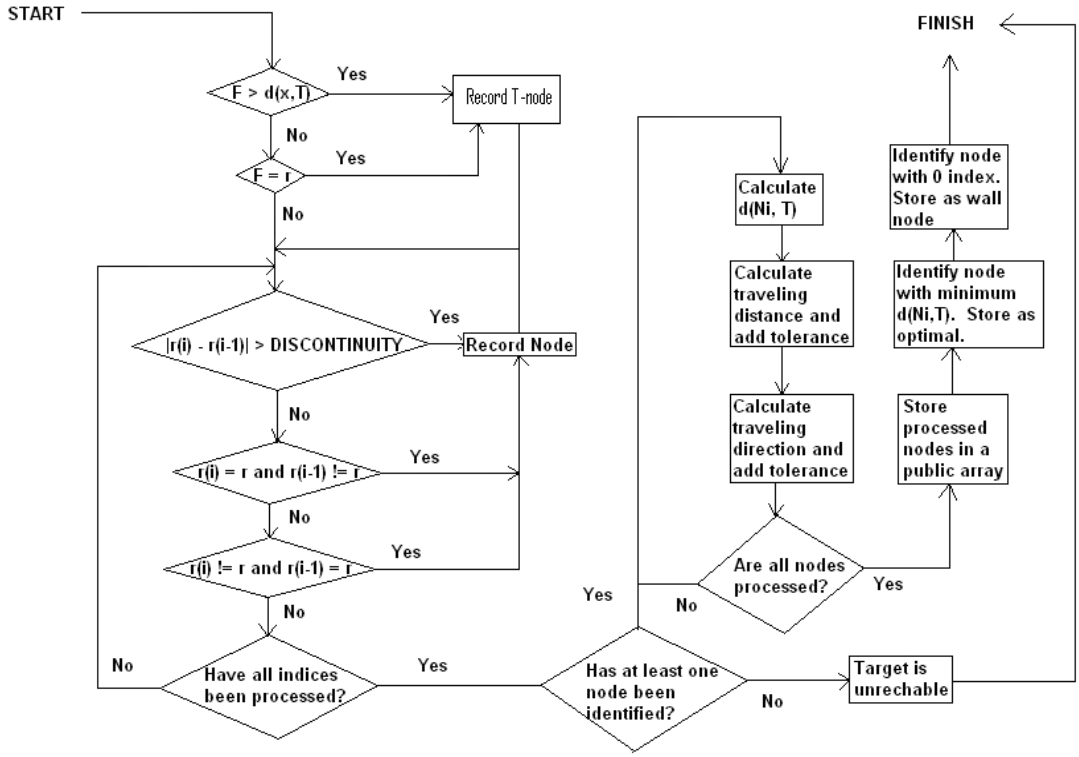

Figure A-19 The node processing algorithm

A.3.6.3 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 A-20. 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 $d_{reach}(T)$.

```
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 A-20 The leaving_condition_holds() method

To evaluate dfollowed(T), for each index from 0 to w determine the distance to target at end-points. In figure 2-21, 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 dfollowed(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 A-21, 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 $d_{reach}(T)$.

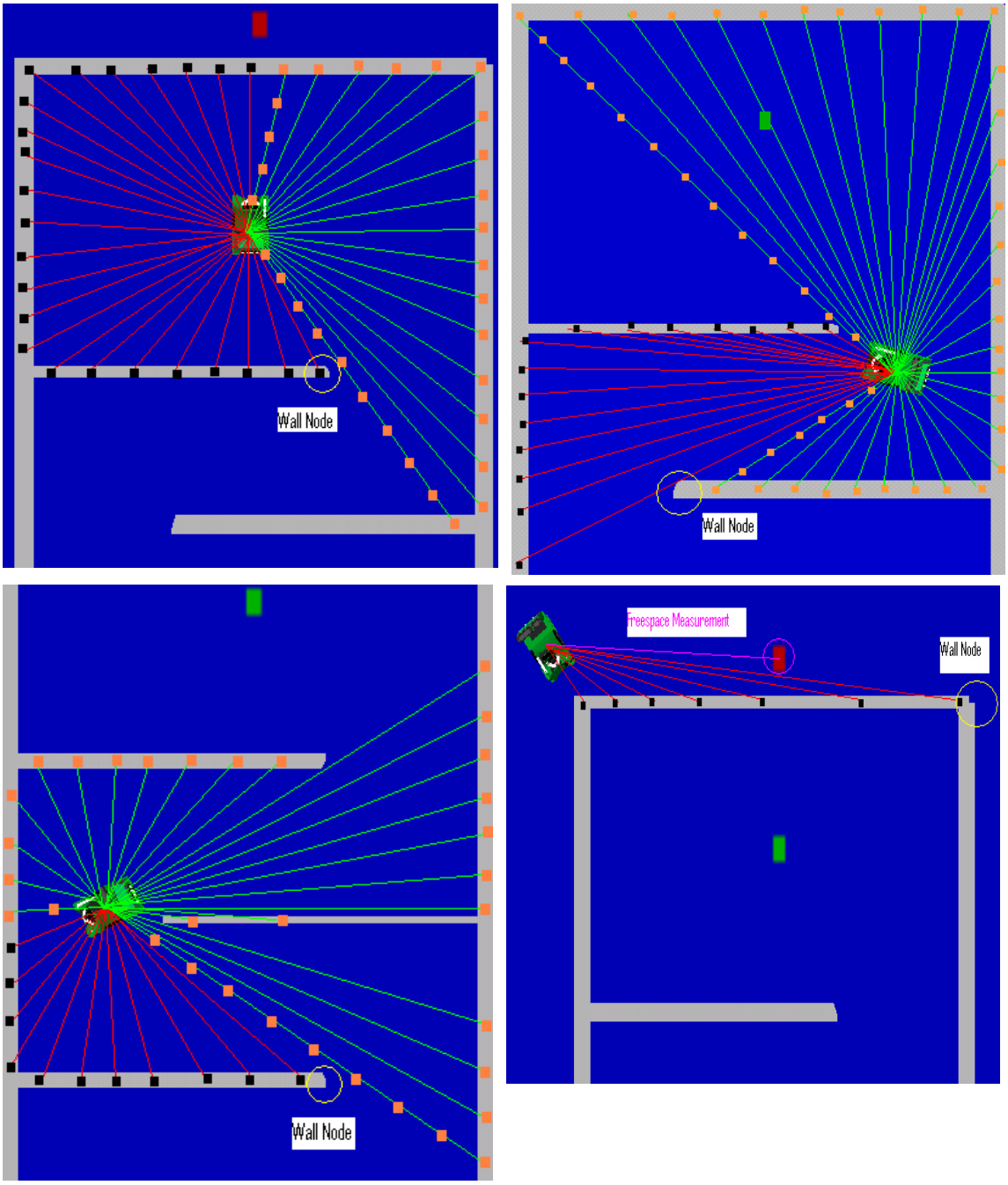

Figure A-21. (a) Top left. The scanning performed at its initial position. (b) Top right. The scanning performed after traveling one node. (c) Bottom Left. The scanning performed after traveling two nodes. (d) Bottom Right. Freespace identifies a visible target.

A.3.7 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 A-22 shows the D* class diagram.

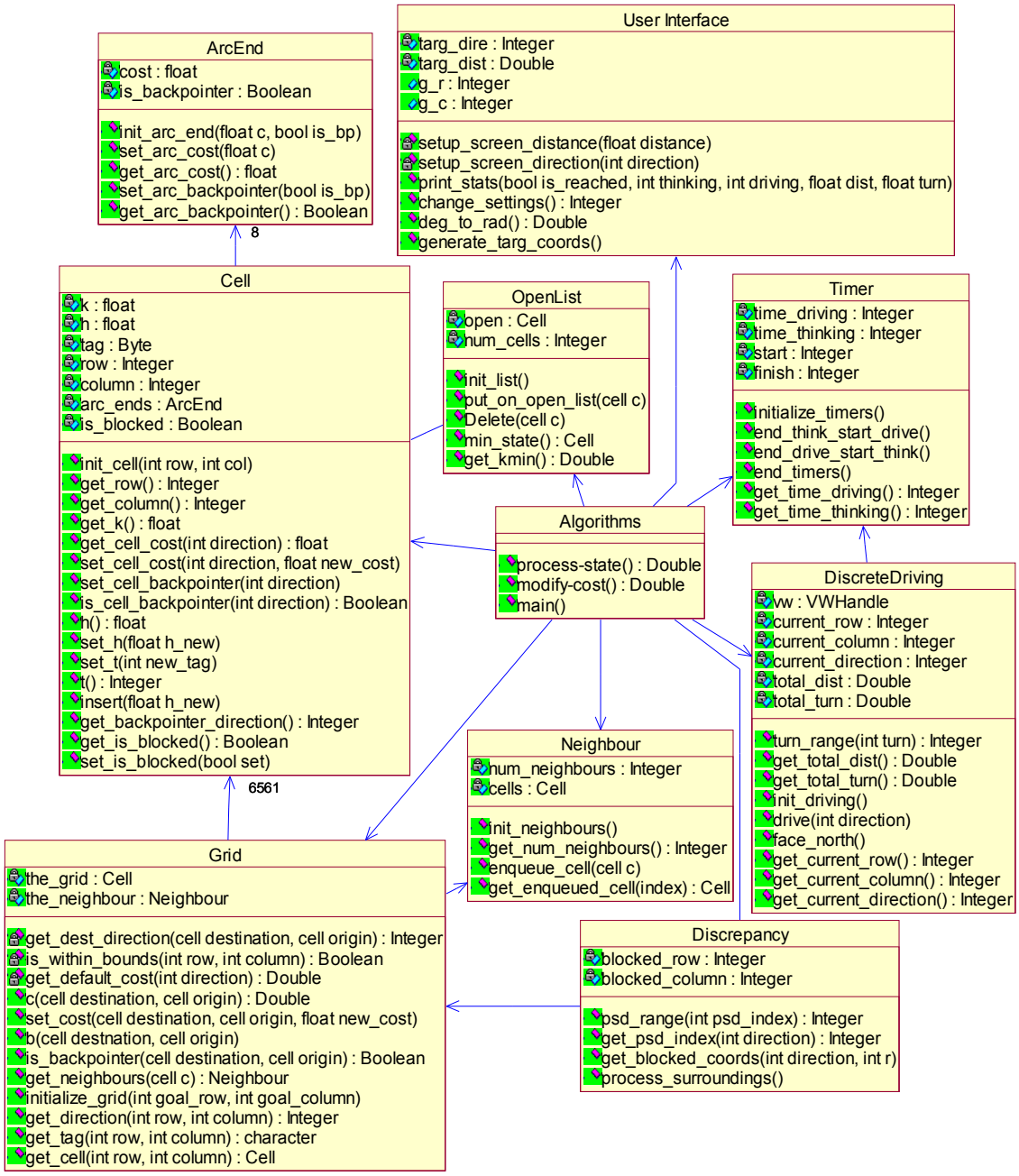

Figure A-22 The D* class diagram

A.3.7.1 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 target following the cell's current backpointer trail.
- its k value, as specified by the original article. This represents the lowest cost of reaching the target 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 target established.
    - Open means that the cell is a neighbour of the target cell or a cell which is closed. Open cells are continually evaluated for minimum cost to target in the table fashion described in section 1.5.7. 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.

A.3.7.2 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.

A.3.7.3 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 1.5.7. 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.

A.3.7.4 The Grid Class

The Grid class is composed of all cells in a grid-like formation analogous to the grid diagrams in section 1.5.7. 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 A-23 shows how the grid class handles the call.

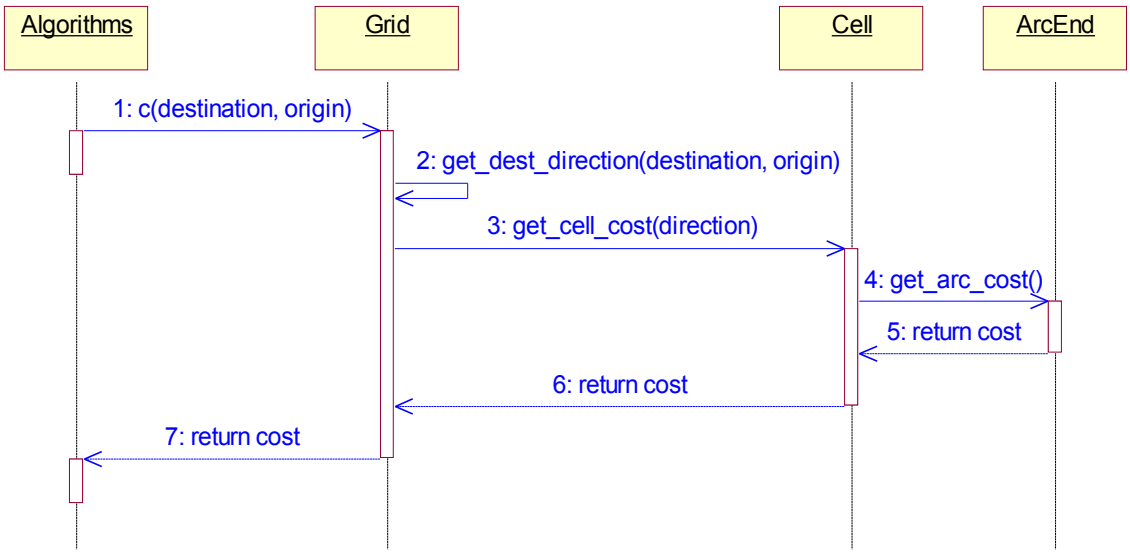

Figure A-23 The sequence diagram for the c call

Another function prescribed by the original article is b(destination, origin) which sets the origin's backpointer in the destination's direction. Figure

A-24 shows how the grid class handles the call.

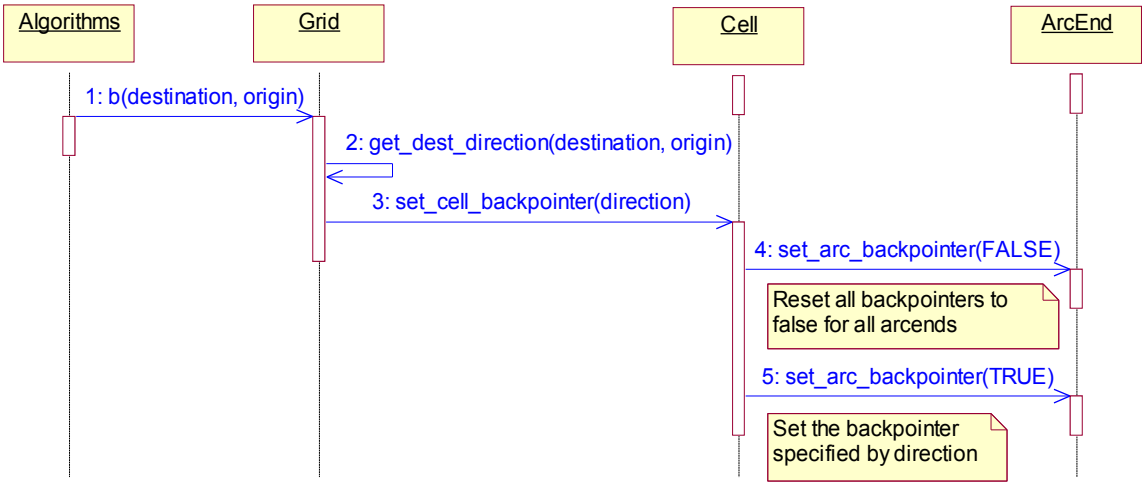

Figure A-24 The sequence diagram for the b call

A.3.7.5 The Neighbour Class

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

A.3.7.6 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 1.5.7.2.

A.3.7.7 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 A-25 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 A-25 the main method