Autonomous Robot Path Planning

A THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH

By
C. B. Crous
March 2009

Supervised by: Prof. A. B. van der Merwe

Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: ..... Date: .....

Summary

In this thesis we consider the dynamic path planning problem for robotics. The dynamic path planning problem, in short, is the task of determining an optimal path, in terms of minimising a given cost function, from one location to another within a known environment of moving obstacles.

Our goal is to investigate a number of well-known path planning algorithms, to determine for which circumstances a particular algorithm is best suited, and to propose changes to existing algorithms to make them perform better in dynamic environments.

At this stage no thorough comparison of theoretical and actual running times of path planning algorithms exist. Our main goal is to address this shortcoming by comparing some of the well-known path planning algorithms and our own improvements to these path planning algorithms in a simulation environment.

We show that the visibility graph representation of the environment combined with the A* algorithm provides very good results for both path length and computational cost, for a relatively small number of obstacles. As for a grid representation of the environment, we show that the A* algorithm produces good paths in terms of length and the amount of rotation and it requires less computation than dynamic algorithms such as D* and D* Lite.

Afrikaanse opsomming

In hierdie tesis beskou ons dinamiese roetebeplanningalgoritmes vir robotika. Kortliks bestaan die dinamiese roetebeplanningprobleem daaruit dat 'n optimale roete, in terme van die minimering van 'n gegewe koste funksie, vanaf een posisie na 'n ander in 'n bekende omgewing met bewegende obstruksies bepaal moet word.

Ons doel is om bekende roetebeplanningalgoritmes te ondersoek, te bepaal onder watter omstandighede 'n gegewe algoritme die beste vaar en om veranderinge aan bestaande algortimes voor te stel sodat hierdie algorimes beter werk in dinamiese omgewings.

Tans bestaan daar geen deeglike vergelyking in terme van teoretiese en werklike looptyd van roetebeplanningalgoritmes nie. Ons hoofdoel is om hierdie leemte te vul deur bekende algoritmes en verbeterde weergawes in a simulasie omgewing te ondersoek.

Ons wys dat indien 'n sigbaarheidsgrafiek voorstelling van 'n omgewing met relatief min obstruksies gekombineer word met die A* algoritme, goeie reslutate verkry word in terme van die roetelengtes en die berekeningskoste. Wanneer 'n roostervoorstelling van 'n omgewing gebruik word, lewer die A* algoritme goeie roetes in terme van berekeningskoste, padlengte en hoeveelheid rotasie wat 'n robot moet uitvoer. In die geval van 'n roostervoorstelling is die A* algoritme ook in terme van berekeningskoste aansienlik goedkoper as dinamiese algoritmes soos byvoorbeeld D* en D* Lite.

Acknowledgements

I would like to thank

- first and foremost, my supervisor Prof. Brink van der Merwe, for his guidance and motivation;
- Dr. Lynette van Zijl for organising financial support;
- my family for their continous support; and
- my fiance Melanie for all her help and encouragement.

Special appreciation to the Harry Crossley Bursary and the National Research Foundation (NRF) for their financial assistance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past two decades, many advancements have been made in the field of robotics. In 1993 the international RoboCup competition was founded. The goal of the competition is: "By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team." This competition has inspired many academic intuitions to move into the field of autonomous robotics.

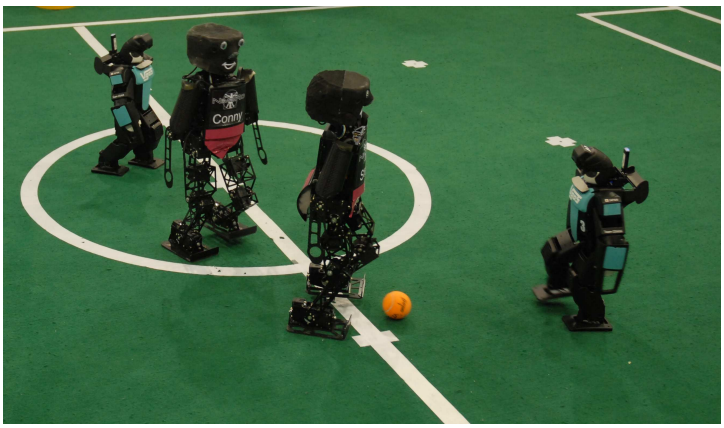

Figure 1.1: RoboCup 2008 Humanoid League [2]

The Defense Advanced Research Projects Agency (DARPA) in the U.S. have sponsored the DARPA Grand Challenge competition for autonomous cars. In the competition, driverless cars must complete a complex off-road course within a limited time. The first DARPA Grand Challenge took place in 2004 in the Mojave Desert along a 240 km course. None of the

contestants finished the course and the furthest distance reached was 11.78 km. In the 2005 Grand Challenge five teams finished with the first doing so in just under 7 hours.

Figure 1.2: Stanley, the winner of the 2005 DARPA Grand Challenge [1]

The twin Mars exploration rovers launched in 2003 are one of the most notable robotics achievements. These two rovers need to be autonomous because of the delay in sending and receiving signals to and from Earth.

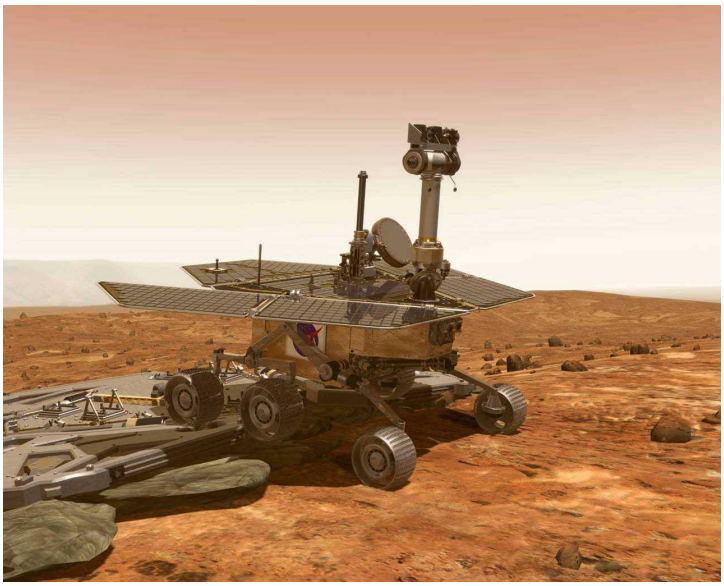

Figure 1.3: The Mars exploration rover drives onto the surface of Mars [25]

For a robot to be autonomous, it must make its own decisions to achieve its goal. One major component of an autonomous robot is its path planner. This is the component responsible for

getting the robot from one location or state to another while avoiding obstacles.

Path planning is used in many fields other than robotics. Transport engineers use traffic models that plan various paths through a traffic network to model the flow of traffic. Games also require some form of path planning to move computer controlled players around the game environment. Both of these applications require a fast path planner, since multiple paths often need to be planned at given time instances. The time and space complexity of robotic path planning algorithms must also be low, since in most cases embedded processors have limited computational power.

Path planners are usually required to produce the shortest collision-free path, however, other objectives such as danger or fuel consumption could also be used. Another common objective is to cover an entire area. This objective is used by domestic robots that clean floors or by security robots that have to periodically patrol a given area. Coverage algorithms are an entirely different class of path planning algorithms which we do not consider.

1.1 Thesis outline

Chapter 2 focuses on defining the path planning problem. It discusses various aspects closely related to path planning. The chapter concludes with how a graph can be used to solve the path planning algorithm. In Chapter 3 a number of different path planning algorithms are discussed. Each is described in detail and their strengths and weaknesses are investigated. Since graph algorithms are a common choice to solve the path planning problem, Chapter 4 focuses on various methods to represent the environment as a graph. Chapter 5 revisits some of the previously discussed algorithms and investigates how they adapt to changes in a dynamic environment. In Chapter 6 the algorithms are compared with each other using a simulator and the experimental results are discussed. Finally, conclusions are presented in Chapter 7 and possible future work is discussed.

Within the figures throughout this thesis the following conventions will be used. A robot will always be illustrated with a small black dot while its goal position will be represented with a cross. Grid cells that are occupied by an obstacle will be filled with dark grey while the

obstacle's actual shape may be given as a light grey region within the occupied cells. Other environments may represent obstacles filled with a crosshatch pattern or a solid colour. A dashed line from the robot's position to the goal will indicate a potential path.

Chapter 2

Background

In this chapter the robotic path planning problem is defined, and various aspects of path planning such as the configuration space, localisation and mapping is also discussed. The chapter concludes with a definition for a graph representation of the environment and two common graph solutions.

2.1 Configuration space

To plan paths for a robot within an environment, both the robot and the environment need to be modelled. Certain assumptions can be made to simplify these models.

One method of modelling a robot and its environment is by defining an N-dimensional vector which describes the state of the robot within the environment. This vector is referred to as the robot's configuration and should be complex enough to accurately describe the state of the robot within the environment. It should also be simple enough to keep its dimensionality relatively low so that the path planning problem is computationally feasible. For instance, a robot which has a fixed body and can move in any direction within 3-dimensional space has six degrees of freedom, namely its position (x, y and z) and its rotation (pitch, yaw and roll). An external moving part of the robot, for example a manipulator arm, should also be included in the configuration if it has a significant impact on the operation of the robot. The set of all possible robot configurations is known as the configuration space.

For this research, we will only consider circular robots that can turn on the spot. We also assume that the environment is two dimensional. This gives us three degrees of freedom, namely, the x and y position of the robot and the direction the robot is facing (the yaw). The robot's direction will also often be ignored, however, we will briefly consider the effect on the efficiency of path planning algorithms of also taking the direction of the robot into consideration. These assumptions are not very restrictive, since any robot can be enclosed by a minimum radius and, therefore, be considered circular or be designed to have a circular shape. Terrain information can also be added to the metric used in the path planning to plan paths across terrains that are not flat. The only assumption that seriously restricts the class of robots is that we only consider robots that can either turn on the spot or are omni-directional (can move in any direction). This includes many robot designs with the exception of car-like robots.

With the given configuration definition, the configuration space is a continuous 2-dimensional Euclidean space $\mathbb{R}^2$, or, when taking direction into account, $\mathbb{R}^2 \times S^1$, where S^1 is the unit circle. For certain path planning solutions this space has to be discretised.

To simplify the path planning, we will regard the robot as a point in space. Since the robot is not actually a point, we will need to make some adjustments to get valid collision free paths. A common approach is to expand the borders of all the obstacles and reduce the boundaries of the environment by the radius of the robot. Now, if the point representation of the robot is touching the edge of an expanded obstacle, the physical robot is just touching the obstacle's edge and not penetrating it. Figure 2.1 shows an illustrative example with a circular robot in an environment with two square obstacles on the left, and the point representation of the robot in the same environment with the boundary-expanded obstacles on the right. This figure shows that the robot is unable to move through the gap between the obstacles and the expanded obstacles prevent the point-size robot from doing so.

2.2 Localisation and Mapping

For a robot to move through an environment, it needs to know its own position as well as the positions of the obstacles. These problems are generally referred to as localisation and mapping, respectively.

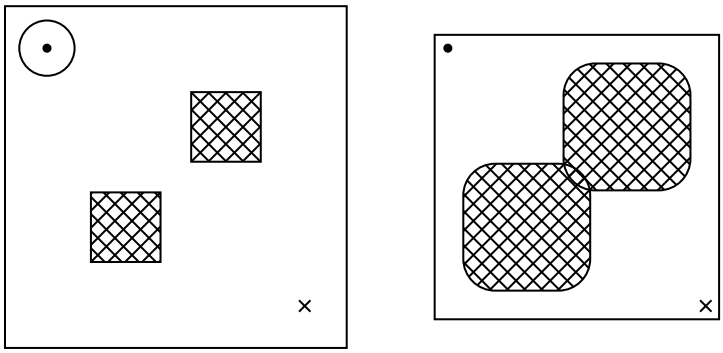

Figure 2.1: The left figure shows a robot as a dot with its radius as a circle around it. The right figure shows the same environment with the obstacles expanded to allow the robot to be modelled as a point-size object.

The solutions to these problems have evolved over the past few decades. In the mid-70's exact models of the environment were used and, therefore, no sensors were needed [21, 28]. Following this in the mid-80's, the opposite extreme was favoured where no models were used and the robots relied entirely on good sensor data [8, 22]. From the 90's hybrid models became popular [3, 23]. These used environment models on a higher level while using sensors at lower levels. Since the mid-90's probabilistic approaches have been used which integrate models and sensing and allow for inaccuracies in both.

The most profound breakthrough for these problems was the realisation that localisation and mapping can be combined into a single estimation problem and that this problem is convergent [9]. This means that the estimation for the robot's location and the location of the landmarks in the environment becomes more accurate as more sensor readings are taken and used. This approach is known as simultaneous localisation and mapping (SLAM) and was first proposed in [10].

The most common approach to model inaccuracies in sensors is to use Gaussian noise. This lead to the use of the extended Kalman filter (EKF) to solve the SLAM problem [9]. Another approach, which uses a more general non-Gaussian probability distribution lead to the Rao-Blackwellized particle filter, also called FastSLAM [24].

This research will focus primarily on the problem of planning a path in a dynamic environment and will, therefore, assume that the whole or part of the environment is known as well as the robot's position. In a real world setting, it often happens that the robot's map of the

environment is not accurate. In this case, the robot will sense a change in the environment at some point which will have a similar effect on the path planner as moving obstacles. Therefore, if a path planner handles moving obstacles well, it will also handle inaccuracies in the environment map well.

2.3 Problem description

The robot's configuration has been defined to be a point in its environment which is given by its position and rotation. With this, the path planning problem can now be defined as determining the next best move to make from the current configuration so that the robot will eventually assume the goal configuration. The choice made must minimise a suitable metric, such as distance travelled, from the initial position to the goal. The information available regarding changes in the environment should also be used when determining the next best move.

This definition does not require a complete path from the starting configuration to the goal configuration. It only requires that the path planner is able to repeatedly give the next best move to make to reach the goal configuration. This also states that the robot will move by discrete amounts. This restriction is a necessity, since it is impossible for a robot to make continuous adjustments to its drive system. Even if it issues another instruction immediately, some time will elapse between two consecutive commands.

2.4 Graph algorithms

One method to model the robot's environment is to use a directed graph. A directed graph $G(V, E)$ is defined to be a set of vertices V and directed edges E. An edge $e = (v, w) \in E$ is a directed link from the source vertex $v \in V$ to the destination vertex $w \in V$. Each edge $e \in E$ has a cost associated with it given by $c(e)$. For each vertex v, the set of vertices which is the destination of some edge on that vertex is given by $out(v) = \{w | e = (v, w) \in E\}$ and the set of vertices which is the source of some edge on that vertex is given by $in(v) = \{w | e = (w, v) \in E\}$. The start vertex is defined as the vertex which represents the robot's physical position and is denoted by v_S. The goal vertex represents the destination that the robot is required to reach

and is denoted by v_G.

The vertices in the graph represent locations in the environment and edges indicate paths that exist between the endpoints of edges. The edge cost can be defined in many ways with the simplest being distance. One could also factor in danger, fuel consumption, gradient, and many other metrics.

There are many ways to represent the environment as a graph. For a 2-dimensional world, the graph could be given in a grid pattern with edges between adjacent grid cells. If there is an obstacle between two adjacent vertices, the edge could either be removed or its cost could be set prohibitively high. This and other graph representations are investigated in Chapter 4.

Many graph algorithms calculate a path by expanding vertex neighbours from one vertex to another until a path is found between the vertices. Depending on the algorithm, the start vertex or the goal vertex will be expanded first. One might guess that the direction in which the algorithm processes the vertices would not have a great impact on the distance of the path, but in a dynamic environment it turns out to have a profound effect.

Figure 2.2 gives an illustration of two city blocks with a start position as a dot towards the top left and a goal position as a cross towards the bottom right. There are three distinct paths given in the figure, namely, a, b and c, and each covers the same distance. Path a was constructed with the strategy from the start position to reduce the distance from the goal position as fast as possible. Path c has the same strategy as a, however, it was planned starting at the goal. Now suppose a robot takes either path a or c. When the robot reaches the road running vertically in the middle of the figure, it might be unable to cross immediately because of traffic. If it took path c, it would have no other option other than to wait until the road is clear. However, if it took path a, it has the option to travel down the road until the traffic clears up and can then cross the road, assuming jaywalking is allowed as indicated in path b. This would mean that the robot would not lose any time waiting at the intersection. This simple example does not prove that one strategy is better than the other. Rather, it illustrates that the performance of closely related strategies might differ significantly in a dynamic environment. Section 6.2 will show which strategy is better by using experimental results.

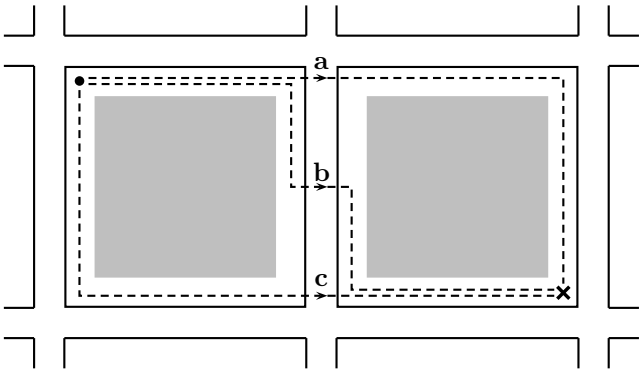

Figure 2.2: Three equal length paths are given. Choosing path a allows for path b to be taken if the road in the centre cannot be crossed immediately.

2.4.1 Breadth-first search

Breadth-first search (BFS) is a simple algorithm which assumes all the edges have the same weight (see Appendix A.1). Optimal paths are obtained when using BFS. It processes vertices in the order of their start distances, measured in edge traversals. This is done by maintaining two lists of vertices. The first list contains the vertices that must be processed in the current level while the second list contains the vertices that will be processed in the next level. The algorithm is initialised with the start vertex marked as visited and placed in the first list while the second list remains empty. While the first list is not empty, a vertex is removed from it and expanded. When a vertex is expanded, it adds all its neighbours which have not been visited to the second list and marks them as visited. Once all the vertices in the first list have been expanded, the lists are swapped and the process is repeated. Once the goal has been reached, i.e., it is added to the second list, the algorithm can stop since a path has been found. If both lists are empty and the goal has not been reached, there is no path between the start and the goal.

The lists used can be implemented as linked lists and insertion and removal can thus be done in constant time. The vertex tags (or labels) and back pointers can be stored as part of the vertex data and be given the required initial values when the vertex is initialised. Thus we may assume that getting and setting vertex tags requires constant time. In the worst case where the graph consists of two connected components, one of which contains only the goal vertex and the other every remaining vertex, each vertex except the goal will be visited and all edges

will be examined. This gives a worst case running time of $O(n + m)$ for breadth-first search, where n is the number of vertices and m is the number of edges in the graph. If the graph has a uniform structure where the maximum number of edges for each vertex is constant, the running time of breadth-first search is $O(n)$. This is the case in a graph representing a grid with 4-way connectivity.

This method is the simplest graph algorithm which will give optimal paths provided all the edge costs are equal. Since this is not a good general assumption, a more general algorithm is needed to allow for different edge costs.

2.4.2 Dijkstra's algorithm

Dijkstra's algorithm can be seen as an extension of BFS to allow for different edge costs (see Appendix A.2). It works by maintaining a cost function $g(v)$ which is an estimate cost from the starting vertex to the vertex v. The function is initialised as zero for the start vertex and infinity for all other vertices. Vertices are visited in the ascending order of their g values. When a vertex is visited, it is marked as visited and the g values of all its unvisited neighbours are updated if the g value of the vertex plus the edge cost to the neighbour is less than the g value of the neighbour. Once a vertex is marked as visited, its g value contains the exact cost from the starting vertex to that vertex. If the goal vertex is visited then a path is found, otherwise, if there are no more vertices to visit then there is no path. The actual path can be obtained by maintaining back pointers which indicate for each vertex the vertex that expanded it.

This algorithm visits each vertex connected to the start vertex once and will look at each of its neighbours until the goal vertex is visited or all the vertices have been visited. With this algorithm, a priority queue is used to order the vertices. This queue can be implemented as a heap which will allow for $O(\log n)$ insertions and removals. This gives an asymptotic running time of $O((n + m) \log n)$, where n is the number of vertices and m the number of edges. As in the case of breadth-first search, the asymptotic running improves to $O(n \log n)$ if the graph has a uniform structure.

This chapter shows how to model the robot and the environment. It also covers how robots can maintain a map of their environment as well as their position by using one of a number of

SLAM methods. The path planning problem has been defined and two graph based solutions were covered. The next chapter investigates a number of solutions to path planning problem.

Chapter 3

Literature overview

This chapter looks at a number of path planning algorithms which will be used in the remainder of this thesis. This is followed by a study of related work that compares path planners.

3.1 Algorithms

3.1.1 Bug algorithm

Traditionally the Bug algorithm is designed for use on a robot with sensors which have a limited range [17]. Figure 3.1 shows a robot guided by the Bug algorithm. The algorithm moves the robot directly towards the goal. The obstacles are assumed to be polygonal in shape and either convex or concave. When an obstacle obstructs the robot's path, the robot must move around the edge of the obstacle until it reaches another point on its original path to the goal. Afterwards the robot continues towards the goal. If the robot travels completely around the obstacle without being able to move towards the goal, a path to the goal does not exist. This version of the Bug algorithm is referred to as the Bug2 algorithm. The Bug1 algorithm is similar, but it requires the robot to move completely around an obstacle before it decides how to proceed to the goal. Neither of these Bug algorithm variations provide optimal paths. It can be shown that if a path exists, that these two Bug algorithms will find one, otherwise, they will detect that no path exists [22].

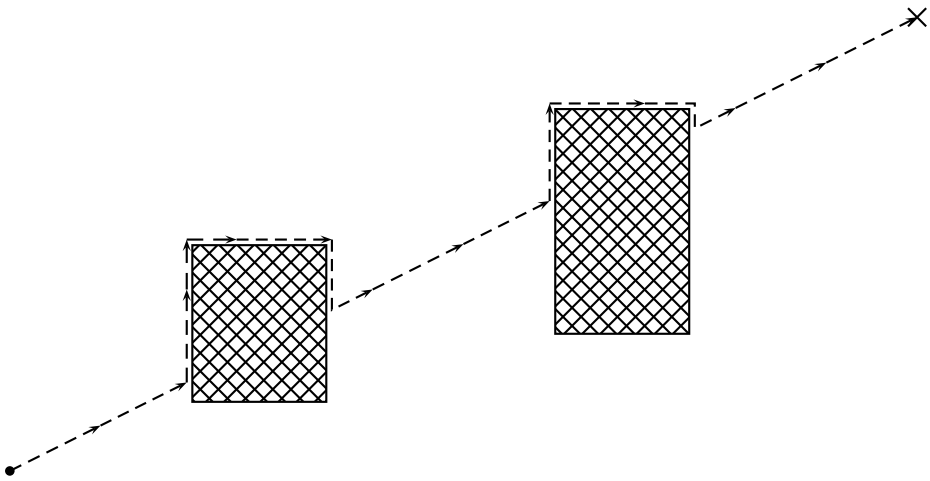

Figure 3.1: A robot guided by the Bug algorithm

The Bug2 algorithm can be improved in terms of finding shorter paths. The Tangent Bug algorithm, which is presented in [17], extends the Bug algorithm by using the sensor information within a given range of the robot. The Tangent Bug algorithm requires a robot with range sensors all around it. The algorithm operates in two modes, namely, the motion to target mode and boundary-following mode. In both modes, the algorithm models the obstacles within its sensor range as thin walls. Therefore, the algorithm does not model the depth of an obstacle it encounters, only that there is an obstruction. These walls are constructed by detecting discontinuities in the sensor readings. For instance, if the range changes gradually and suddenly increases to the maximum range, when considering sensor readings within a given angular interval, the angle for which the sudden increase occurs corresponds to the vertex of an obstacle. Using the sensor information, the algorithm constructs a local tangent graph from the obstacles. The local tangent graph contains vertices on the edges of obstacles and edges from the robot's position to some of these vertices.

In the motion-to-target mode, the robot moves in a straight line towards the goal or towards a vertex of an obstacle for which the path through that vertex is the shortest know path. If the robot has to move away from the goal, the boundary-following mode is initiated.

When the robot follows an obstacle boundary, the algorithm uses the local tangent graph to do so. While moving around the obstacle's boundary, the value d_{min} is updated to be the shortest distance between any point on the obstacle and the goal. The robot stops moving along an

obstacle's boundary once it finds a point on the local tangent graph which is closer to the goal than d_{min}. If such a point is found, the robot moves to this point and resumes the motion to target mode once the robot is closer to the goal than d_{min}.

Figure 3.2 shows a robot which is controlled by the Tangent Bug algorithm. The dotted circles show the sensor range of the robot. In this figure, one can see how the robot initially moves towards the goal. Once the first obstacle in its path comes within sensor range, the robot avoids it and moves towards the edge of it. Once free from the obstacle it moves directly towards the goal again until the process is repeated with the second obstacle. When compared with the Bug algorithm in Figure 3.1, the Tangent Bug algorithm gives a path which is much closer to the optimal path.

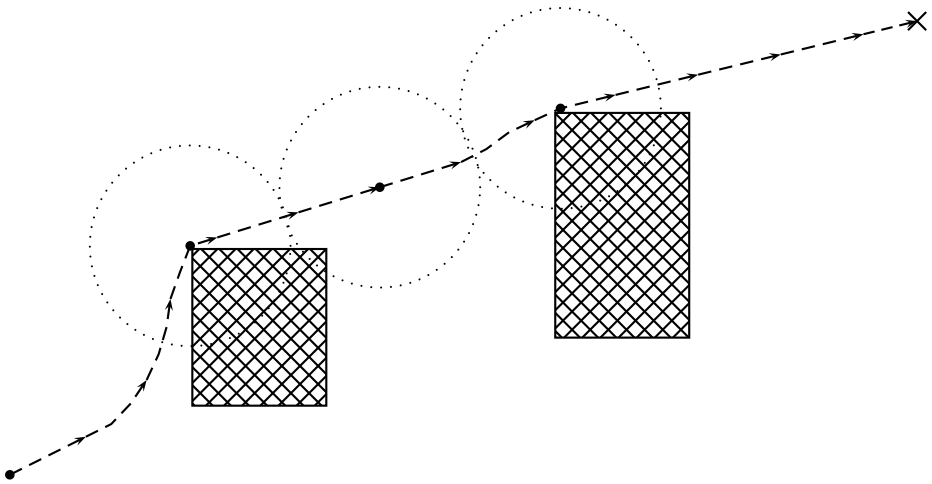

Figure 3.2: A robot guided by the Tangent Bug algorithm

Although the Bug2 and Tangent Bug algorithms can provide paths without much computation, they are designed to work in environments with stationary obstacles. A new algorithm which shall be referred to as the Dynamic Bug algorithm was developed to optimise the Tangent Bug algorithm for dynamic environments. The Dynamic Bug algorithm is discussed in Section 5.1.

3.1.2 Potential functions

The potential functions method of obtaining a path is a rather simple and elegant solution [12]. In this method, a potential function is used to create a potential field in which the goal is an attractive point and the obstacles are repulsive points. To achieve this effect, the total potential

function is constructed by summing potential functions for the goal and all the obstacles. The total potential function is given as

$$U(p) = \sum_{i=0}^{n} U_i(p)$$

where p is any point in the environment, U_0 is the potential function for the goal and U_i for $i \in [1, n]$ are the potential functions for the n obstacles.

The direction of movement of the robot is determined by using the gradient descent method on the potential function. The gradient descent method gives the direction to move in as

$$\dot{p} = -\frac{\partial U(p)}{\partial p} = -\sum_{i=0}^{n} \frac{\partial U_i(p)}{\partial p}$$

To use the gradient descent method, the potential function U must be smooth. If it is not, then the robot's path may oscillate around points which are not smooth.

Since a continuous vector field is used to plan the robot's path, this can be used to control the robot's drive system directly. This is not possible with planners which give discrete solutions that must be interpolated to provide a smooth path.

Example

To illustrate a working example of the potential functions method two simple potential functions are used. A linear function is used for the goal while a gaussian function is used for the obstacles. These functions are given by

$$\begin{aligned} U_0(p) &= k||p - p_0|| \\ U_i(p) &= me^{-||R_i(p-p_i)||^2} \text{ for i} \in [1, \text{n}] \end{aligned}$$

where p_0 is the goal's position and p_i for $i \in [1, n]$ are the positions of the obstacles. The k value is used to scale the gradient of descent into the goal. The obstacle sizes are controlled using the diagonal matrix R_i. The diagonal entries of R_i inversely scale the size of the obstacle on each axis and, therefore, the area which the obstacle covers. The height of the obstacles is set by m which controls how the potential functions of the obstacles affects the vector field.

To use the gradient descent method on the potential function, the first-order partial derivative is needed. The derivative of the goal and obstacle functions were calculated to be

$$\begin{aligned} \frac{\partial U_0(p)}{\partial p} &= k\frac{p-p_0}{||p-p_0||} \\ \frac{\partial U_i(p)}{\partial p} &= -2me^{-||R_i(p-p_i)||^2}R_i(p-p_i) \text{ for i} \in [1,\text{n}] \end{aligned}$$

The example shown in Figure 3.3 used the above potential functions in an environment with two obstacles. In the example, the potential function's surface is shown and the minimum at the goal as well as the two obstacle peaks can be seen. The robot's starting position is $(5,5)$ and its path shows how the potential functions method guides it towards the goal while avoiding the obstacles. An immediate observation which can be made is that the path is not optimal. Since the potential function can only affect the robot's path in close proximity to it, the robot will only start avoiding the obstacle when it is close to it. This is seen in the example at the start where the path moves directly towards the obstacle before it moves to the right to avoid it.

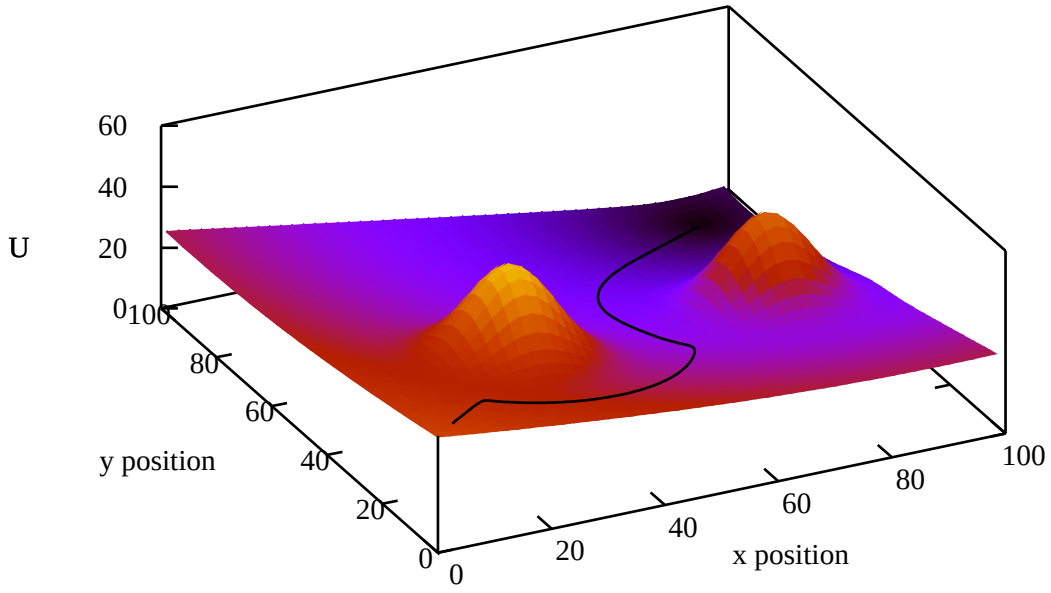

Figure 3.3: A potential function example using a linear function for the goal, gaussian functions for the obstacles and using the gradient descent method for the path. The parameters used for the functions are $k = 0.3$, $m = 30$ and $R_0 = R_1 = [0.1, 0; 0, 0.1]$.

Problems

One major problem with potential functions is the possible and likely existence of local minima. This problem has been solved in many ways. The most efficient solution seems to be to use special potential functions, called navigation functions [27]. Navigation functions are defined to only have one local minimum, the goal configuration. In [27] the potential function approach is also generalised so that it can be applied to a space of any dimension and to a robot with an arbitrary number of degrees of freedom.

A problem with using the gradient descent method is that it may produce paths with many oscillations. This occurs for example when the goal is close to an obstacle and the robot is on the opposite side of the obstacle, or when the robot is moving in a narrow gap between two obstacles. Figure 3.4 shows an example of a potential field with two closely spaced elongated obstacles. The robot's starting position is at the bottom left and the goal position is on the top right. The path produced by using the gradient descent method causes the robot to oscillate between the obstacles. The potential field between the obstacles has a small gradient towards the goal and a large gradient towards the midpoint between the obstacles. For the robot to follow the vector field between the obstacles, it has to move in extremely small increments and it has to use high-precision arithmetic. Since the robot must move in relatively small increments and since floating point arithmetic is not exact, the robot is not able to move precisely between the obstacles. This will cause the robot to oscillate between the two sides of the centre line between the obstacles.

The oscillation problem can be solved by using a modified Newton's method [26], as opposed to the gradient descent method. The modified Newton's method gives the direction to move in as

$$\dot{p} = -B(p)\frac{\partial U(p)}{\partial p}$$

where $B(p)$ is a positive-definite matrix and is defined as $B(p) = (\varepsilon I + H)^{-1}$, $\varepsilon = \varepsilon(p) \geq 0$, I is the identity matrix and H is the Hessian matrix of $U(p)$. The Hessian matrix is the second order partial derivative of the potential function. The value for ε is the smallest value which makes the matrix $\varepsilon I + H$ positive-definite and, therefore, invertible with its inverse also being positive-definite. For low dimensional environments, $B(p)$ can be calculated without a large

computational overhead.

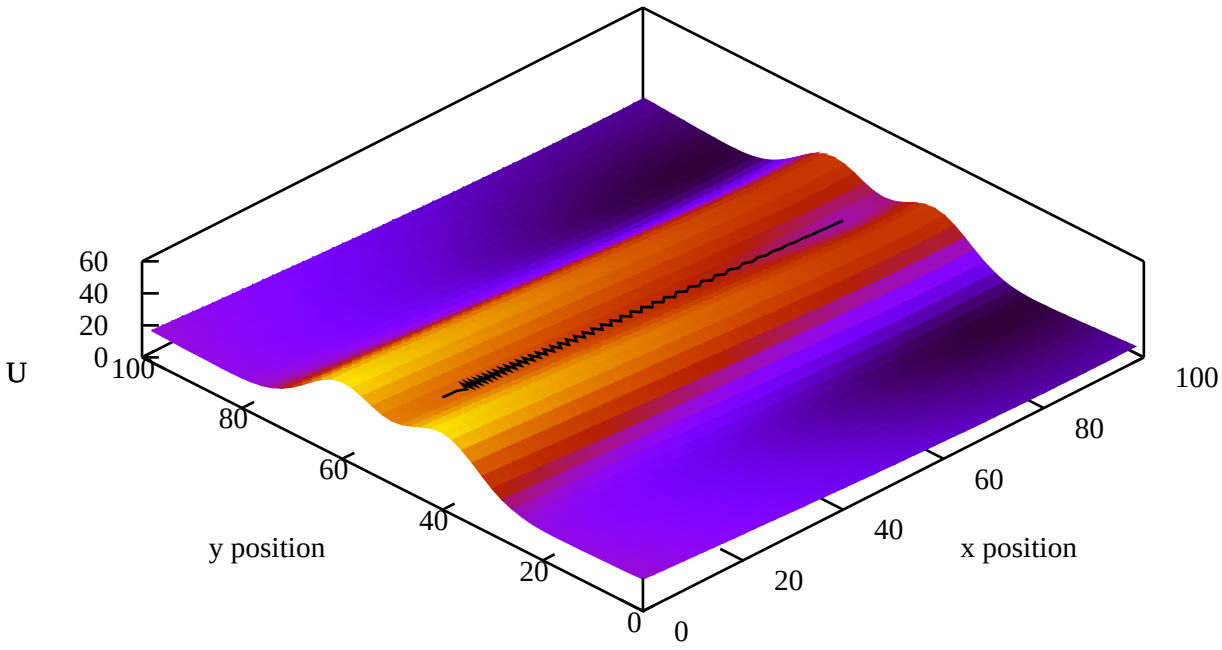

Figure 3.4: An example showing the oscillations of a robot between two obstacles in close proximity. The functions used are the same as in Figure 3.3 with the parameters given by $k = 0.2$, $m = 30$ and $R_0 = R_1 = [0.001, 0; 0, 0.1]$.

Certain problems can make the goal configuration unreachable. When the goal is placed close to an obstacle, the repulsive force of the obstacle outweighs the attractive force of the goal and, therefore, the goal is no longer the global minimum. New repulsive potential functions were developed in [12], where the distance between the robot and the goal is taken into consideration. These potential functions ensure that the goal can be reached and that it is the global minimum of the total potential function.

Although the potential functions method is normally computationally more efficient than other methods, the paths generated are suboptimal. Thus, we conclude that although the various problems encountered when using potential functions can be solved individually, it is often cumbersome to provide a general solution for all circumstances.

3.1.3 A*

The A* algorithm was first published in [14]. It makes use of a heuristic function to determine the order in which to process the graph vertices. A heuristic is additional information that can be provided about the structure of the graph. For instance, if Euclidean coordinates for each vertex is known, this information can be used by defining the heuristic function to be the Euclidean distance function. The heuristic function can be chosen as any underestimate for the exact distance between any two vertices. The closer the heuristic function is to the exact distance, the more the algorithm improves in terms of processing fewer vertices. In fact, [14] not only proves that A* produces optimal paths, but also that A* determines shortest paths by processing the fewest possible graph vertices for a given heuristic.

The article also shows that if the heuristic overestimates shortest path distances, then A* no longer guarantees optimal paths, but paths might be found quicker. In [14] the heuristic function is also required to be consistent. The definition of a consistent heuristic is given in Section 4.1. The consistency assumption is unnecessary and in [15] it is shown that one can use a heuristic function that does not obey the consistency assumption. If a heuristic is used that is not consistent, A* must be modified to re-open closed vertices if a shorter path is found from the initial vertex to a given closed vertex.

The A* algorithm makes use of data structures similar to those used in Dijkstra's algorithm. It uses an estimated cost function g, a labelling function t, a back pointer function b as well as a priority queue called the open list. In addition to these, it uses a heuristic function h which gives the heuristic distance between a given vertex and the goal vertex. The priority queue is ordered by the function $f(v)$ where $f(v) = g(v) + h(v)$ for each vertex $v \in V$.

Initially, all vertices are labelled as *NEW* and all other vertex functions are undefined. The algorithm is initialised by labelling the starting vertex as *OPEN*, its cost (g) is set to zero and it is inserted into the priority queue. The algorithm then processes the priority queue until it is empty or the goal vertex is labelled as *CLOSED*. When a vertex is removed from the priority queue, the cost function of all neighbours labelled as *NEW* or *OPEN* are considered. For *NEW* neighbours, the cost is set to be the cost of the current vertex plus the edge cost to the neighbour. The same calculation is used for *OPEN* neighbours, but this calculated value

is only used to update the cost of the neighbour if it improves on the current cost. For *NEW* neighbours, and neighbours with improved cost values, the back pointers are set to the current vertex, they are labelled as *OPEN* and are (re)inserted into the priority queue.

The A* algorithm is given in Appendix A.5 and is very similar to Dijkstra's algorithm. In fact, if the heuristic function is set to zero it is equivalent to Dijkstra's algorithm.

The worst case running time of A* is the same as Dijkstra's algorithm, but if the chosen heuristic is good, A* will process much fewer vertices. For a graph obtained from a grid structure, Dijkstra's algorithm will always expand the vertices in the order of their distance from the start vertex while A* will first expand the vertices on the straight line from the start to the goal and then expand the vertices neighbouring this line. Figure 3.5 gives an illustrative example of this with each square a graph vertex that is connected to its eight neighbours. The dot and cross indicate the start and goal vertices respectively. The dark grey squares are the vertices which have been processed and are closed while the light grey squares are vertices which are in the priority queue and are still open. This example shows that the A* algorithm processes vertices directly towards the goal since it uses extra information about the location of the goal. On the other hand, Dijkstra's algorithm has no such information and must process vertices in all directions.

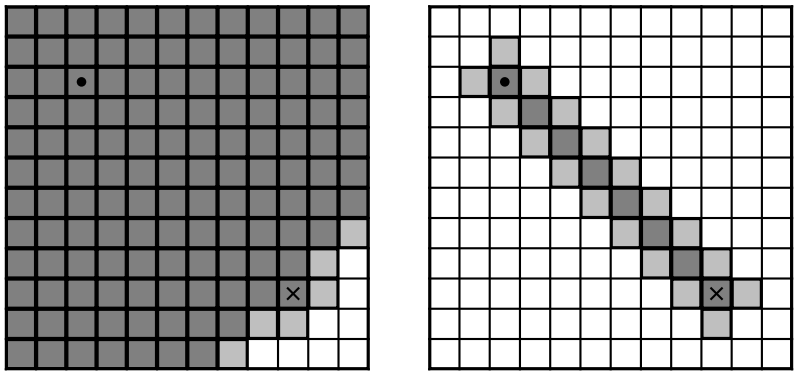

Figure 3.5: Dijkstra's algorithm (left) processes grid cells radially while the A* algorithm (right) processes grid cells in the direction of the goal if a good heuristic is used. The dark cells are cells which have been processed while the lighter cells are ones which are on the open list and are scheduled to be processed.

As previously discussed, an important consideration is the direction in which the algorithm processes its vertices. For this reason, the A* algorithm with the start and goal vertices

swapped will also be investigated to determine the effect of processing from the goal vertex as opposed to processing from the start vertex. When A* is used, but the vertices are processed from the goal vertex, we shall refer to the algorithm as Reversed A*. An immediate advantage of Reversed A* is that the back pointers will now point in the direction of the robot's motion. Therefore, after the algorithm has finished, the complete path does not have to be explored to make the first step. When the robot requires the next move to make, the Reversed A* algorithm simply returns the back pointer to the current vertex. One possible side effect of reversing the A* algorithm is that the path distances may now change as illustrated in Figure 2.2. This will be further investigated in Section 6.2.

3.1.4 D*

The D* algorithm was introduced in [29] and has been used in many applications [7, 31]. Its name is derived from "Dynamic A*", but the algorithm does not use any heuristic information. It is actually an extension of Dijkstra's algorithm which allows changes in the environment.

One major difference between this algorithm and A*, is that it starts processing vertices from the goal vertex, not the start vertex. The most important feature of this algorithm is that if a change in the environment occurs, the altered vertices are revisited and the change is propagated outwards, often only processing a few vertices. This implies that the whole graph is not explored every time a new obstacle is discovered or an obstacle moves within the environment.

Due to this algorithm's complicated nature, a thorough investigation was done and is provided in Appendix B. This includes the algorithm's proofs and illustrated examples to help aid in the understanding of various aspects of the algorithm.

Similar to the A* algorithm, D* makes use of a current cost function g, a vertex tag function t, and an open list. It also defines a previous cost function p which is used to maintain the order in which the vertices in the open list are processed after a change has occurred. The key function $k(v) = \min(g(v), p(v))$ is used to determine this order. The previous cost function p can be integrated into the key function k and is, therefore, redundant. This has been done for the D* algorithm given in Appendix A.6.

3.1.5 Focused D*

As previously discussed, the D* algorithm does not use heuristics as the A* algorithm does, and the author of D* extended it to do so in [30], naming the new algorithm Focused D*. The Focused D* algorithm keeps the same notation as the D* algorithm, but it does not maintain the previous cost values (p) and instead uses the open list key value (k) to store the same information. The article maintains the D* notation of having the symbol h refer to the current cost of a vertex and introduces the symbol g as the heuristic cost. This notation conflicts with the notation used in the A* paper [14]. To avoid confusion, the A* notation will be used.

The Focused D* algorithm is almost identical to the D* algorithm: the most notable difference is how the open list is sorted. The vertices in the open list are sorted by $f(v) = k(v) + h(v) + f_b$ where f_b is the bias of the robot when the vertex was inserted into the open list. This bias is initially zero and is increased by the heuristic cost between the robot's previous and current positions whenever it moves. Using this bias solves the problem of the heuristic values becoming invalid whenever the robot moves. In addition to this, if a vertex is removed from the open list and if the f value it was inserted with is greater than its current f value, i.e., it is being processed too early, then it is reinserted into the open list. This preserves the order in which the vertices must be processed according to the robot's current position.

Figure 3.6 compares the order in which D* and Focused D* process their vertices. The figure consists of three plots of the state of the algorithms. The environment used consists of two obstacles with a gate between them which is initially open. The first graph gives the initial state of the environment where both algorithms will construct a path between the two obstacles. All the vertices in the graph below the dashed lines will lead to the goal on the outside of the obstacles. The D* algorithm will have to expand almost all of the vertices in the graph to find its path. However, since the Focused D* algorithm uses heuristics, only the vertices in a straight line from the robot to the goal will have to be expanded.

The second graph shows how the D* algorithm adapts to the gate closing. The lighter region represents all the vertices which are in a raised state (their cost from the goal has increased) and the darker region represents all the vertices which are in a lowered state (their cost from the goal has decreased). All the vertices in a lowered state were all previously in a raised state.

The progression of the raised states forms a wave that starts at the closed gate and expands upwards. The lowered states also form a wave that starts at the outer corners of the obstacles and remains behind the raised state wave as it moves upwards.

The third graph shows how the Focused D* algorithm adapts to the gate closing. The raised state wave starts at the gate and extends upwards, but it does not extend past the outer edges of the obstacles. The lowered state wave again starts at the corners of the obstacles and then moves almost directly towards the robot's position. Comparing this to the D* algorithm, one can see that many fewer vertices need to be visited for both the raised and lowered waves.

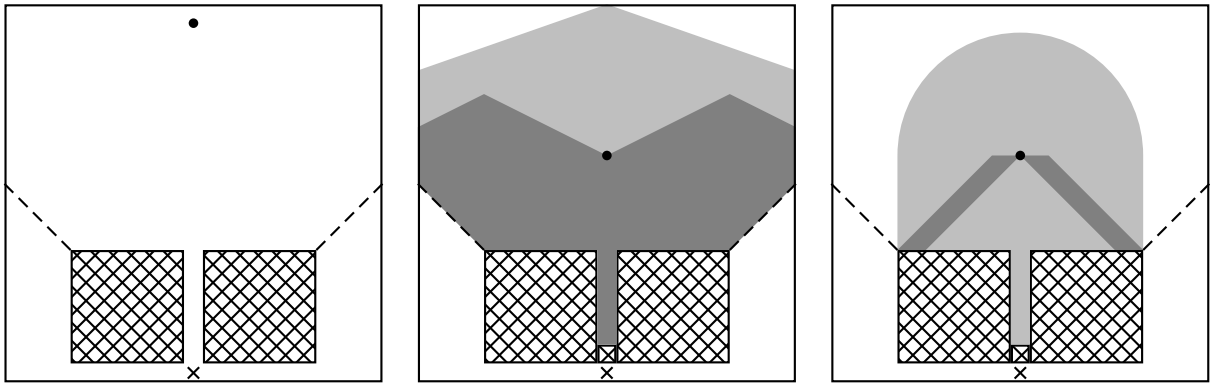

Figure 3.6: A comparison between D* and Focused D* [30]. The initial state (left) with two obstacles where both algorithms will plan a path directly towards the goal. After the robot has moved a gate is closed between the obstacles. The D* algorithm (middle) and the Focused D* algorithm (right) have to process a number of vertices to react to this. The light regions are cells which are in a raised state while the dark region are cells which are in a lowered state.

3.1.6 D* Lite

A more recent algorithm which was developed to make use of both previous path calculations as well as heuristics is the Lifelong Planning A* (LPA*) algorithm [19]. This algorithm was developed by adapting the A* algorithm to reuse information if the environment changes. This algorithm's implementation is simple and easy to understand, but it is only for a fixed start and goal position and, therefore, cannot be used directly for robot path planning.

The D* Lite algorithm was developed from LPA* and functions much like D* as a path planner, however, the actual algorithm is quite different [18]. The D* Lite algorithm is given in Appendix A.7. An optimised version of D* Lite is also presented in [18] which is the version

implemented and used in Chapter 6.

The D* Lite algorithm has been used in a number of applications. The most notable is its use in the Field D* algorithm [11] which was uploaded to NASA's Mars Exploration Rovers Spirit and Opportunity in 2006 [32]. This new algorithm allows the rovers to behave more autonomously and to plan paths around obstacles.

Similar to the D* algorithm, the D* Lite algorithm makes use of an open list which is an ordered list of vertices that need to be processed. Initially, the open list contains only the goal vertex and vertices are processed from the list, in order, until the termination conditions are met. The D* Lite algorithm uses the common definitions for the current cost of a vertex (g) and the heuristic cost of a vertex (h). It also defines a look ahead value *rhs* which is the next value for the current cost. The open list is sorted by the key vector (α, β) where $\alpha = min(g(v), rhs(v)) + h(v) + k_m$, $\beta = min(g(v), rhs(v))$, and k_m is a constant value determined by the robot's movement and is similar to Focused D*'s robot bias value. The first element in the key vector is used to order the vertices and ties are broken using the second element.

Whenever the robot moves, the k_m value is increased by the heuristic distance between the robot's previous and current positions. Since the k_m value changes, once a vertex is processed its actual key value may be different from the one it was inserted into the open list with. If this is the case, and the key it was inserted with is greater than its actual key, then the vertex is reinserted into the open list. This is done to preserve the correct order in which the vertices must be processed at the new location of the robot.

3.2 Related work

There are a number of complete books that present and explain various path planning algorithms. *The Principles of Robot Motion* by Choset *et al* [6] covers many algorithms including the simple Bug algorithm. It also covers other topics which are required for a complete solution such as the configuration space, localisation, mapping, and trajectory planning. It does not, however, give any comparisons between the algorithms. This is also the case for *Planning Algorithms* by LaValle [20] which is quite detailed and verbose.

Some articles present new algorithms and compare them with previous ones, but the comparisons are not always thorough. In the D* Lite article by Koenig and Likhachev [18] the D* Lite algorithm is compared to the D* algorithm. However, only the number of vertex accesses, expansions, and heap changes are compared. Even though the true processing time of each algorithm is related to these factors, the extra constant time which is associated with them is still unknown. For instance, in some cases the D* algorithm expands significantly fewer vertices than the A* algorithm, but each D* vertex expansion has a much higher computational cost than an A* vertex expansion.

Haro and Torres [13] directly compare different path planning algorithms using their actual running time. The paper claims to be the first paper to perform such a comparison. The authors consider three algorithms, namely, the Bug algorithm, the Potential Fields algorithm, and the A* algorithm with multi-resolution grids. These algorithms are compared in only three examples and the actual processing time and path distances are given. Although this gives a good indication of the differences in the running time, the limited set of examples used does not give a true indication of how these algorithms will perform in a truly dynamic environment. In addition to this, the way in which the A* algorithm is used is far from optimal. In this article, the complete A* algorithm is run for each step the robot takes regardless of whether the environment changes or not. It is also not clear if the time taken to set up the multi-resolution grid is included in the measured running time of the A* algorithm.

This chapter has identified and discussed a number of path planning algorithms and investigated their strengths and weaknesses. Related work was investigated but none was found to compare path planning algorithms in many different dynamic scenarios by using their true computation time. Since many path planning algorithms make use of graphs, the next chapter provides a detailed investigation into various graph representations of the robot's environment.

Chapter 4

Graph representations

The placing of the graph vertices and the connections between these vertices is determined by how the graph represents the environment. The grid, visibility map, quad tree, and directional grid representations are investigated in this chapter. Different distance heuristics and their properties are presented and discussed. In addition to this, various other aspects, such as how the robot actually moves between graph vertices, are discussed.

4.1 Grid representation

As previously discussed, an environment can be approximated by a grid. Each cell in the grid will be represented by a vertex in the graph and the cell's neighbours will have appropriate edges within the graph. Defining the neighbours of each cell is an important factor for both accuracy and speed. If a neighbourhood is too small, then the algorithm will have a limited range of motion, and if a neighbourhood is too large, then the algorithm will take too long processing all the neighbours for each cell. The three common classifications for neighbourhoods are illustrated in Figure 4.1. The von Neumann neighbourhood (left) includes only those cells which are directly North, South, East and West of a given cell, while the Moore neighbourhood (middle) includes the diagonal neighbours. The extended Moore neighbourhood (right) includes the Moore neighbourhood and their Moore neighbours. The centre of a neighbourhood is the cell from which the neighbourhood is constructed. The radius r of a neighbourhood is defined

such that the number of rows or columns in a neighbourhood is equal to $2r + 1$. For example both the von Neumann and Moore neighbourhoods have a radius of one while the extended Moore neighbourhood has a radius of two. The extended Moore neighbourhood can be enlarged further by increasing its radius.

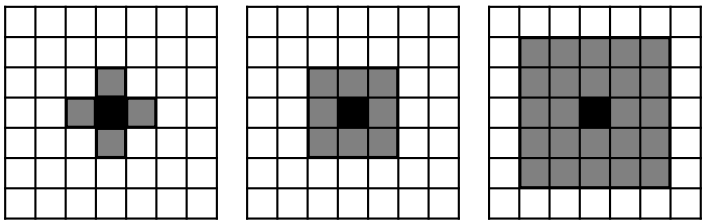

Figure 4.1: The von Neumann (left), Moore (middle) and extended Moore (right) neighbourhoods

Increasing the radius of the extended Moore neighbourhood will allow more accurate paths, however, the number of cells in the neighbourhood increases quadratically as the neighbourhood radius increases. In addition to this higher cost, problems can arise when cells in the middle of the neighbourhood are occupied while the outer cells remain unoccupied. Figure 4.2 illustrates this problem. The black cell in the figure represents the cell occupied by the robot. The grey cells represent the robot's neighbourhood. The crosshatched cells are occupied by an obstacle. In the figure, the robot moves along the edge of an obstacle until it reaches the obstacle's corner. Since the outer corner of the robot's neighbourhood is unoccupied, it will attempt to move through the obstacle. To solve this problem, a hierarchy could be formed where the neighbours closer to the centre of the neighbourhood determine if the outer neighbours are free.

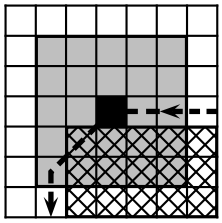

Figure 4.2: With a large grid neighbourhood (grey cells) it is possible for an obstacle to occupy a cell which is between the robot (black cell) and one of its unoccupied neighbours as illustrated

Once a neighbourhood is chosen, the cost of moving between the grid cells must be defined. Many academic papers use a cost of one between all the unoccupied neighbours and a high cost for the occupied neighbours [19, 29]. This may be a simple solution, but it is a poor model

of a real environment, since the diagonal distance between neighbouring cells is greater than the horizontal or vertical distance. A better solution is to use the Euclidean distance between the neighbours. In a Moore neighbourhood, this would make the cost of travelling between unoccupied horizontal and vertical neighbours 1, and diagonal neighbours $\sqrt{2}$.

Since the Moore neighbourhood is a good trade-off between accuracy and simplicity (and therefore speed), it will be used as the grid neighbourhood for the remainder of this thesis.

Heuristics

Some algorithms make use of a heuristic cost which is an approximate distance between the vertices of the graph. In general, the heuristic cost must be non-negative, an underestimate of the exact distance between the cells and in some cases consistent. A heuristic cost $h(u, v)$ between two cells $u, v \in V$ is consistent if it obeys the triangle inequality $h(u, v) \leq h(u, w) + h(w, v)$ for all vertices $u, v, w \in V$. In a grid representation, one of the distances in Table 4.1 can be used for the heuristic distance. For a von Neumann neighbourhood, the Manhattan distance gives the exact minimum distance between two cells. The Diagonal Manhattan distance, which is used in the Lifelong Planning A* algorithm [19(pp. 2)], can be used for Moore neighbourhoods. The Euclidean distance can also be used, but it is computationally expensive and using it can lead to rounding errors that can make the heuristic inconsistent. The Moore distance was calculated to be the shortest possible distance to move from one position to another within a grid using a Moore neighbourhood. The maximum difference minus the minimum difference gives the distance which will be travelled between horizontally and vertically adjacent neighbours. The distance travelled between diagonal neighbours is given by $\sqrt{2}$ times the minimum difference. For example, to travel from (0, 0) to (8, 3) using a Moore neighbourhood, a robot must move five grid cells horizontally and three grid cells diagonally. The Moore heuristic distance for this movement is given as $8 + 3(\sqrt{2} - 1) = 5 + 3\sqrt{2}$, which is the exact distance. The Moore heuristic is used in [25] where it was called "octile distance". Although the Moore distance is consistent, it suffers from the same rounding errors as the Euclidean distance.

Metric	Formula
Euclidean	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
Manhattan	$\lvert x_1 - x_2\rvert + \lvert y_1 - y_2\rvert$
Diagonal Manhattan	$\max(\lvert x_1 - x_2\rvert, \lvert y_1 - y_2\rvert)$
Moore	$\max(\lvert x_1 - x_2\rvert, \lvert y_1 - y_2\rvert) + \min(\lvert x_1 - x_2\rvert, \lvert y_1 - y_2\rvert)(\sqrt{2} - 1)$

Table 4.1: A list of heuristic distances and their formulas

Problems

Various problems can arise when using a grid representation. Obstacles represented on the grid will always occupy more space than they do in reality. Figure 4.3 shows a light grey circular obstacle being represented on a grid. The dark grey regions are the extra space used to represent the obstacle on the given grid resolution. If any portion of the obstacle occupies the grid cell, the whole cell must be assumed to be occupied.

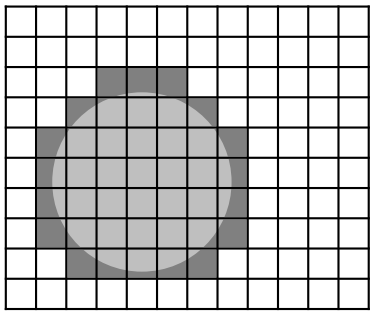

Figure 4.3: A circular obstacle represented by a grid. The dark grey region illustrates the extra space required to represent the obstacle on the grid.

Having grid cells that are larger than the obstacles can cause problems. Figure 4.4 gives an example with large grid cells and a relatively small obstacle. The light grey rectangle is the obstacle and the dark grey blocks are the grid cells which the obstacle covers. The dots represent a robot's possible position and the crosses are the target positions. The dashed line from a robot's position to its target position is the shortest path if only the grid information is taken into consideration. In this example, the obstacle is only a single row high and the occupied grid cells have a much higher cost than the unoccupied ones. The algorithm guiding the robot is only aware that an obstacle occupies part of the grid cell which the robot is in. For the case where the robot is below the obstacle, if the algorithm chooses the shortest path,

it will guide the robot to move vertically upwards since it does not know the exact robot or obstacle positions within the grid cell, and in doing so, will make the robot collide with the obstacle. This problem can be easily avoided by ensuring that any obstacle occupies at least two rows and two columns within the grid. To illustrate this solution, consider if the robot was just to the right of the obstacle and its goal position was in the cell to the left of the obstacle. It will then be guided outside of the occupied cell first either to the top or to the bottom before moving to the goal without colliding with the obstacle.

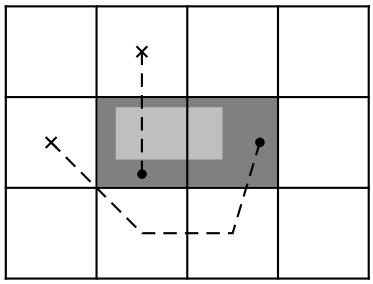

Figure 4.4: Two robots which are on an occupied grid cell which must plan paths to the opposite side of a relatively small obstacle

Further problems occur when two or more obstacles occupy the same cell as the robot. Figure 4.5 shows two examples where the robot moves towards its goal, but it is intercepted by two obstacles which then occupy the same grid cell as the robot. In the figure, the left grid of each example is the initial state and the right grid is the state after the obstacles have intercepted the robot. In the top example, the obstacle's vertical edges occupy the same cells and the robot is caught between them. The shortest path for the robot is to continue between the obstacles towards the goal. If the robot was in the cell below its current one, the shortest path would be to move down and then travel around the obstacles. In situations like these the robot will generally have to rely on its sensors and stop if it gets too close to an obstacle.

In the bottom example, the corners of two obstacles occupy the same cell as the robot. The shortest path takes the robot through the actual obstacles. Here the best solution would be to move back and plan a new path from a previous position as illustrated.

The problems discussed above only arise when a robot is in an occupied cell. If it is, and both the next and previous cell on the robot's path are unoccupied, then the robot could collide with an obstacle if it continues on a previously planned shortest path. In such cases, the robot

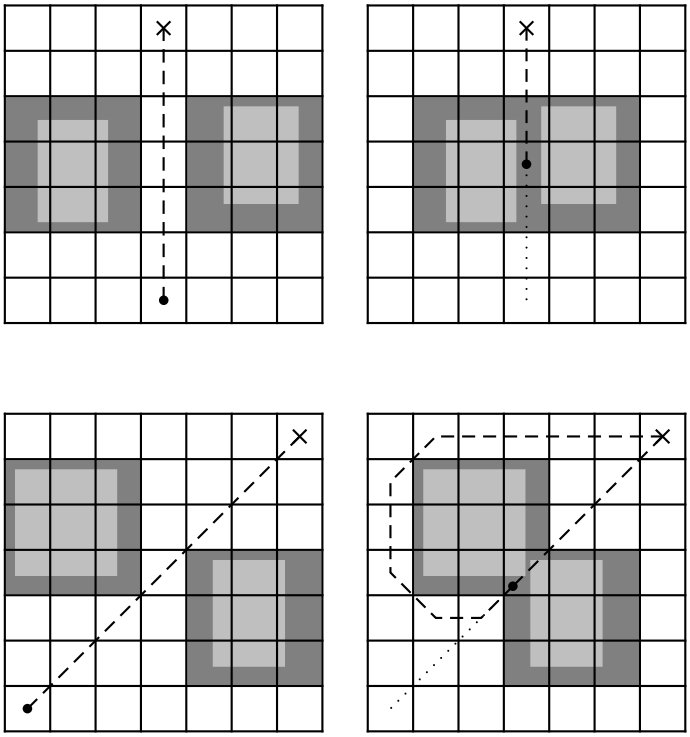

Figure 4.5: Two examples which show the initial path for a robot (left) and the possible paths after the obstacles intercept the robot (right)

should move back to the previous cell it was on and recalculate the shortest path from there. This provides a general solution to both the small grid size and obstacle overlapping problems. Another solution would be to adopt a hybrid approach and use the robot's sensors directly when such problems arise.

Movement between neighbouring cells

One important aspect to consider with the grid representation is how the robot actually moves between the grid cells. One would assume the easiest method would be to move between the midpoints of each grid cell. Although this would ensure that the robot will only occupy cells on its path, even during diagonal movement, this solution is not ideal. Figure 4.6 shows three different strategies which can be used when moving between cells. In the example on the left, the robot moves between the midpoints of the cells on its path which shall be referred to as the cell midpoint strategy. At the start of the robot's path, it has to move to the midpoint of its starting cell and then it has to turn almost completely around to get to the next cell. In the

example in the middle, the robot moves to the midpoint between the current cell it occupies and the next cell on its path. This midpoint will lie on the edge between the two cells. This method shall be referred to as the edge midpoint strategy. In the example on the right, the robot moves to the closest point of the next cell on its path. This method shall be referred to as the closest point strategy. The edge midpoint strategy produces a slightly shorter path with fewer changes in direction than the cell midpoint strategy. Although the closest point strategy will always produce the shortest path, there is an extra computational cost of choosing the closest point on the next cell. In addition to this, the paths it produces have the same sharp turns as the cell midpoint strategy.

The cell midpoint strategy could be improved upon by using the edge midpoint strategy for the first and last cells on the robot's path. Since the closest point strategy will give the shortest path, this strategy will be used as the movement strategy for grids.

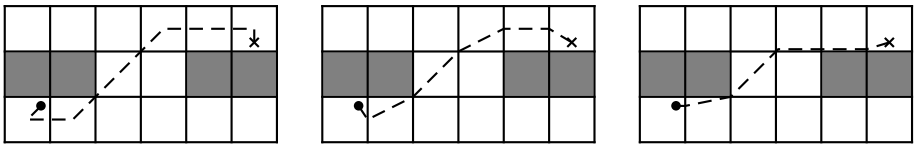

Figure 4.6: Movement between grid cells. The left figure shows the cell midpoint strategy where the robot moves between the midpoints of the individual cells on its path. The middle figure shows the edge midpoint strategy where the robot moves between the midpoints of two adjacent cells on its path. The right figure shows the closest point strategy where the robot moves to the closest point of the next cell on its path.

A method to produce smooth continuous paths is to use interpolation on the planned shortest paths. Since a grid restricts the possible paths, a smooth path obtained by smoothing a grid based path is not necessarily optimal in terms of length. Figure 4.7 shows an example where the path obtained when using a grid differs from the optimal path. Even if the grid path is to be interpolated, the result does not produce the optimal path. To overcome this problem, the Field D* algorithm [11] integrates the interpolation step into the path planner. This is done by effectively allowing the transition from a given cell to move to any point on the edge between the current and a neighbouring cell.

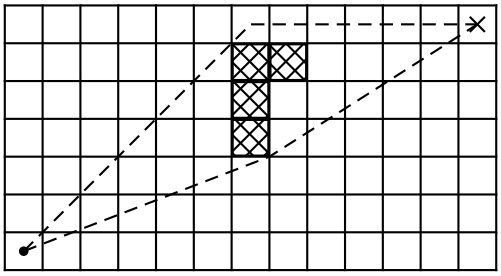

Figure 4.7: A comparison between the optimal (bottom) path and the path obtained using a grid (top)

4.2 Visibility maps

A visibility map is a graph in which vertices are connected if and only if there is an unobstructed line of sight between them. The cost of the edges in the graph are given as the physical distance between vertices. A simple form of a visibility map is where the vertices of the graph are placed on the corners of the obstacles as well as on the starting and goal positions. This type of visibility map is generally referred to as a visibility graph [21] and an example is provided in Figure 4.8. For visibility graphs, we assume that the obstacles in the environment are polygons.

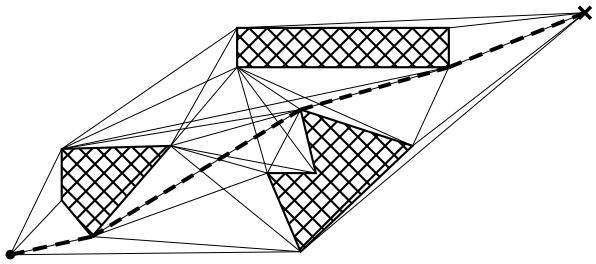

Figure 4.8: A visibility graph for an environment with three polygonal obstacles

To obtain the visibility graph, the visibility between every pair of vertices in the graph must be calculated. To simplify this process, each vertex is taken and the visibility between that vertex and every other vertex is calculated. A rotational plane sweep algorithm is used to calculate the visibility of a vertex. This reduces the number of polygon edges to consider when testing the visibility between two vertices.

The rotational plane sweep algorithm uses a line which starts at a vertex v in the direction of the positive x-axis. Initially, all the polygon edges which intersect this line are determined and placed in a list S. All the graph vertices w, with $w \neq v$, are placed in an ordered queue

E and are sorted in ascending order by the non-negative angle between the positive x-axis and the line from v to w. Each vertex in E is processed in order. When a vertex w is processed, only the polygon edges in S need to be examined to determine if w is visible from v. After the visibility from v to w is determined and before determining the visibility from v to w_{new}, any polygon edge adjacent to w which are in S are removed from S, and the adjacent edges not in S are added to S. By doing so, we ensure that w_{new} is not visible from v if and only if the line from v to w_{new} intersects an edge in S. Appendix A.4 gives a detailed description of this algorithm as well as the pseudocode for its implementation.

To avoid moving the robot too close to an obstacle, the vertices in the visibility graph can be moved further away from their respective obstacles to produce a border around the obstacle. This may, however, produce obstacle polygons that intersect each other as well as the robot's position. If the robot's position is inside an obstacle polygon, the vertex associated with that position will have no edges in the visibility graph and, therefore, no path exists between the starting and goal vertices. To correct this problem, two edges should be created from the position of the robot inside the obstacle to the end points of the closest obstacle edge.

The rotational plane sweep algorithm gives all the vertices which are visible from a single vertex. By starting with just the vertices in the visibility graph and no edges, and using the A* algorithm, only the visibility of the vertices which the A* algorithm processes need to be calculated. Figure 4.9 shows a visibility graph which was calculated in this way. When comparing this figure to Figure 4.8, one can see that it has fewer edges. Combining visibility graphs with A* is thus computationally less expensive than the traditional visibility graph approach when calculating shortest paths. A visibility graph which has been combined with the A* algorithm shall be referred to as the visibility graph algorithm.

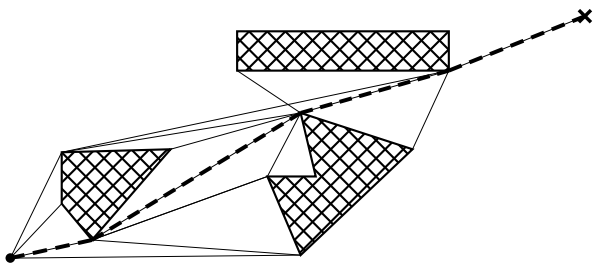

Figure 4.9: A visibility graph for an environment with three polygonal obstacles which is constructed using the A* algorithm

A visibility graph contains fewer vertices and edges than a grid representation. If there are n vertices in the visibility graph, there are at most $n(n-1)/2$ edges. For a 10-by-10 grid with 8 way connectivity, there are 100 vertices and 342 edges, while a visibility graph with 6 rectangular obstacles will only require 34 vertices and at most 288 edges. Thus, the visibility graph has significantly fewer vertices and edges and would, therefore, require less processing when planning a path. The visibility graph will also allow for more accurate paths due to the fact that obstacle positions and shapes are not as crudely approximated as in the case of grid representations. One downside to using visibility graphs is that the time required to calculate this graph might be more than the time required for maintaining a grid representation of the environment. The feasibility of visibility graphs will be investigated in Chapter 6.

4.3 Quad trees

A popular approach used to reduce the number of cells within a grid is to use a multi-resolution grid, such as a quad tree. A quad tree represents the environment by splitting it into four cells, each of which is a node under the root node of the quad tree. Each of these cells can also be split into four cells by creating four child nodes. This is repeated until a desired depth is reached. The decision of whether to split a cell or not, is determined by whether or not the children of a given cell would be different from each other if we split the cell under consideration. If the four children are identical, then they can be represented by the single parent cell. Quad trees allow for a high resolution around the edges of obstacles while the rest of the graph remains at a relatively low resolution.

Figure 4.10 shows a quad tree representation of an environment with a circular obstacle. This quad tree splits its cells up to 5 times and, therefore, has the same resolution as a 32-by-32 grid. There are, however, only 112 leaf nodes in the quad tree, while an equivalent resolution grid has 1024 cells. In this example, one can see how the quad tree splits the environment so that the number of nodes in the tree is minimised while maintaining a high accuracy.

The quad tree partitions do not necessarily have to divide a node into equal parts, nor is it necessary for the partition lines to be perpendicular. The partition lines can also be positioned anywhere within the node, provided they divide the node's space into four parts. By doing

so, an optimal division can be found in which the number of nodes in the tree are minimised. Figure 4.11 shows the same environment as the previous quad tree example, but variable partitions are used. In this example there are only 46 leaf nodes in the quad tree even though both trees have the same depth. The first two partitions isolate the obstacle into a single cell. In the third partition the obstacle is divided into four equal cells. The fourth and fifth partitions were made so as to maximise the area of the child cell which is not occupied by the obstacle. Although using a variable partition can produce a tree with much fewer nodes, the computational cost of doing so often outweighs the possible benefit. In some cases though, such as static environments, the variable partitions can be precalculated and the additional computational cost might be negligible.

Similar to a grid representation, a neighbourhood must be defined for the quad tree cells. The quad tree neighbourhood, however, does not have a fixed number of cells. The equivalent von Neumann neighbourhood for quad trees contains the cells which have edges adjacent to the edges of the centre cell. The Moore neighbourhood contains any cell which touches the centre cell and diagonal neighbours are thus included. When a cell is split, the new children cells construct their neighbourhoods to include some or all of their siblings and a subset of the parent's neighbours. When a cell's children are rejoined, the cell's neighbourhood would include the set of all of its children's neighbours with the exception of the children themselves. The cost of travelling from one neighbour to another is the Euclidean distance between the centres of the two cells.

The robot can move between the cells of the quad tree in the same way as it does for a grid. Any of the three strategies given in Figure 4.6 can be applied to the quad tree cells. Since a single quad tree cell can span a quarter of the environment, any additional movement between cells is undesirable. The cell and edge midpoint strategies both require some additional movement to the centre of a cell or edge between cells. This extra movement could cause the robot to move large distances unnecessarily. Therefore, the closest point strategy is the only method of the three which should be used for the movement between quad tree cells.

In Section 6.5, quad trees and the equivalent resolution grid are compared to determine if, and under which circumstances, the computational gain in processing fewer cells outweighs the computational cost of maintaining the quad tree.

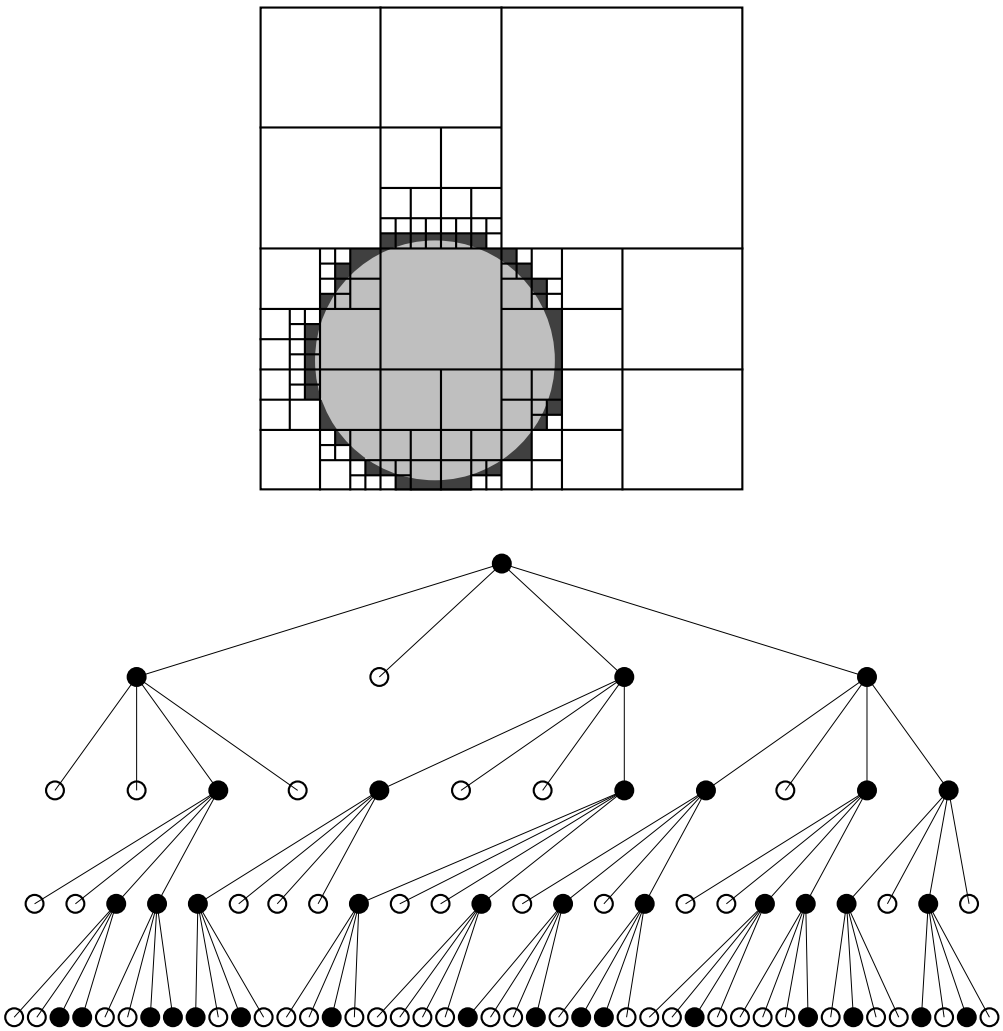

Figure 4.10: An environment (top) with a single circular obstacle divided by using a quad tree (bottom). Only the first four levels of the quad tree are given since the fifth level will only contain leaf nodes. The root node is at the top and the children for each node from left to right are the top left, top right, bottom right and bottom left nodes respectively. The white filled nodes are the leaf nodes in the tree.

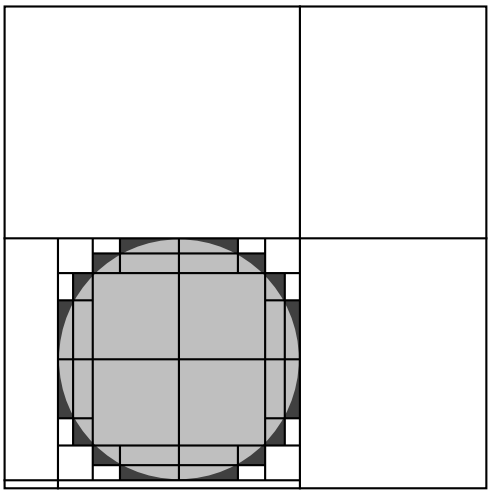

Figure 4.11: An environment with a single circular obstacle divided by using a quad tree with variable partitions

4.4 Directional representation

One important aspect of path planning algorithms is the amount of rotation on the paths they generate. To investigate this within a grid-like environment, a third dimension is added to a regular grid as the direction in which the robot is facing.

The three dimensional grid, which shall be referred to as a directional grid, is constructed from a traditional grid. The traditional grid is split into n layers, where n is the number of cells in the neighbourhood, excluding the centre cell of the neighbourhood. For each layer, the cells in that layer contain only a single neighbour from the original neighbourhood. For example, the first layer's cells will only contain the left neighbours while the second layer will contain the top neighbours. The layers are arranged such that the direction of a neighbour changes by $360°/n$ degrees from one layer to the next. Once arranged, each cell within a layer is linked to the corresponding cells in the two layers adjacent to it. The cost of traversing over this link is the cost of rotating from one neighbour to another. The first and last layers are also linked in this way so that the directional axis wraps around. With the completed directional grid, movement between the layers rotates the robot and movement within a layer moves the robot in a single direction.

The distance between two cells on a directional grid is not the physical distance between the position those cells represent. This means that, if a heuristic is used, it must be updated to include rotation.

Figure 4.12 shows an example of a 3-by-3 grid with a von Neumann neighbourhood which has been expanded into a directional grid. Each of the layers is indicated by the larger squares and the cells within the layers are the smaller squares. The two double arrows represent the links between each cell of one layer to each corresponding cell of the other. From the centre cell in the bottom left layer the only movement is to the right and moving to the adjacent layers rotates the robot to face either down or up.

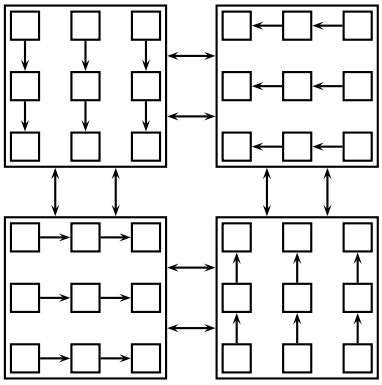

Figure 4.12: The four layers for a directional grid using a von Neumann neighbourhood

The advantage of using the directional grid is that it adds a cost to rotating the robot which should provide paths with fewer turns. The disadvantage of using the directional grid is that it increases the dimension of the grid and forces the path planning algorithm to process many more cells. Although each cell now has three neighbours at most, the total number of edges in the graph has increased to $2mn$, where m is the number of cells in the traditional grid and n is the number of cells in the neighbourhood. In addition to this, each time a change occurs in the environment, the affect on a dynamic algorithm is now multiplied by n since there are n cells which represent the same physical location in the environment.

The directional grid will be used as a case study to determine if it is possible to reduce the amount of rotation in the paths of an algorithm.

This chapter describes four graph representations: the grid, visibility map, quad tree, and directional grid. The grid, quad tree and directional grid all model the environment up to

a certain resolution. It is shown that by doing so, there are various problems that can arise from inaccurate models. The directional grid demonstrates that different objectives can be integrated into the graph representation.

Chapter 5

Dynamic algorithms

This chapter investigates four dynamic path planning algorithms. The algorithms should satisfy a number of requirements. Firstly, the path planning algorithms that we consider must deliver optimal or near optimal paths, that is, paths which minimise a given metric. Secondly, the algorithms must find these paths without excessive computation, since robots usually have limited computing power. We also consider how efficiently the algorithms adapt to changes in the environment.

5.1 Dynamic Bug algorithm

The Dynamic Bug algorithm was developed as part of this thesis to adapt the Tangent Bug algorithm for a dynamic environment. Like the Tangent Bug algorithm, the Dynamic Bug algorithm also uses two modes, namely, the motion-to-goal mode and the boundary-following mode. The most notable difference is that the Dynamic Bug algorithm does not need to observe all the obstacles around itself. In the motion-to-goal mode, if an obstacle is blocking the robot's path to the goal, the Dynamic Bug algorithm only looks as far as the left and right edges of the obstacle. Once both edges of the blocking obstacle have been found, the robot moves past the edge which gives the shortest path to the goal. As in the Tangent Bug algorithm, if the next position moves the robot away from the goal, the boundary-following mode is initiated. In the boundary-following mode, the Dynamic Bug algorithm will note the shortest distance between

any point on the obstacle it is following and the goal. Before moving around the obstacle, the algorithm first checks if there is any point to move to which is closer to the goal than any point on the obstacle. If there is, the robot moves towards this point and the boundary-following mode is terminated. If no such point is visible, the robot continues to move around the obstacle in the direction of the last motion-to-goal movement. Appendix A.3 describes the Dynamic Bug algorithm in detail.

Since the Dynamic Bug algorithm does not maintain any information about the environment, it is does not require any computation when the obstacles move. As with the other Bug algorithms, the Dynamic Bug algorithm is a suboptimal algorithm.

5.2 A*

Although the A* algorithm cannot handle changes within a graph, it can be adapted to perform as a dynamic algorithm. One way to achieve this is to replan the entire path every time the robot wants to move another step. This works well on robots with powerful processors which repeatedly run a sense, plan and act cycle. This method is also necessary for graph representations which depend on the robot's position, such as visibility graphs. Another approach is to run the A* algorithm again only when a change in the environment is detected.

These two methods allow the robot to follow an optimal path, but they are computationally for more expensive than required. From the way that the A* algorithm plans its paths, it follows that only when certain vertices in the graph are modified, that it is required to replan the path. The A* algorithm only uses vertices which were on the open list at some stage to determine the shortest path. Thus if a change occurs to an edge with the property that neither the source nor the destination of the edge were on the open list, then the current optimal path does not change and it is therefore not necessary to execute the algorithm again. If one considers changes which increase the weight of edges, another optimisation can be made. If the weight of an edge is raised, A* only needs to replan its path if this raised edge is on the remainder of the current optimal path to be traversed by the robot.

These new conditions on when to re-run A* could potentially produce a significant reduction

in the computational cost of using A* in a dynamic environment.

5.2.1 Suboptimal A*

More optimisations can be made to the A* algorithm if the restriction of path optimality is relaxed. One method is to use a heuristic which overestimates the exact distance. For instance, if the Manhattan heuristic is used on a Moore neighbourhood, it is no longer an underestimate of the exact distance. Another method is to ignore certain information about the environment. Using this method, a variant to the A* algorithm was developed for this thesis which shall be referred to as the Blind A* algorithm.

The Blind A* algorithm functions exactly as the A* algorithm does, but it only re-runs the A* algorithm if the current path within a certain range is blocked. In a dynamic environment, this choice of when to replan the path is favourable since an obstruction far away from the robot may no longer obstruct the robot once it is closer. The range in which the algorithm will replan its path is chosen to be relatively small, but large enough so that the robot is able to move around obstacles.

Since the Blind A* algorithm ignores possible path obstructions until it is closer to them, it is expected that the paths which this algorithm produces would be on average longer than the A* algorithm. Chapter 6 will investigate how much longer the Blind A* paths are than those of the A* algorithm. It will also determine the computational cost benefit of this suboptimal algorithm.

5.3 D*

To illustrate how the D* algorithm adapts to changes in the environment, a detailed example is given next. Figure 5.1 shows the effect at various stages of running D* on a 4-by-4 grid with 8-way connectivity. The figure contains four grids which are snapshots of the algorithm at different times. The white and grey grid cells are the clear and blocked cells respectively. The cost of moving from one open cell to another is 1 if the cells are horizontal or vertical neighbours and 1.4 if the cells are diagonal neighbours. To simplify the example, it is assumed

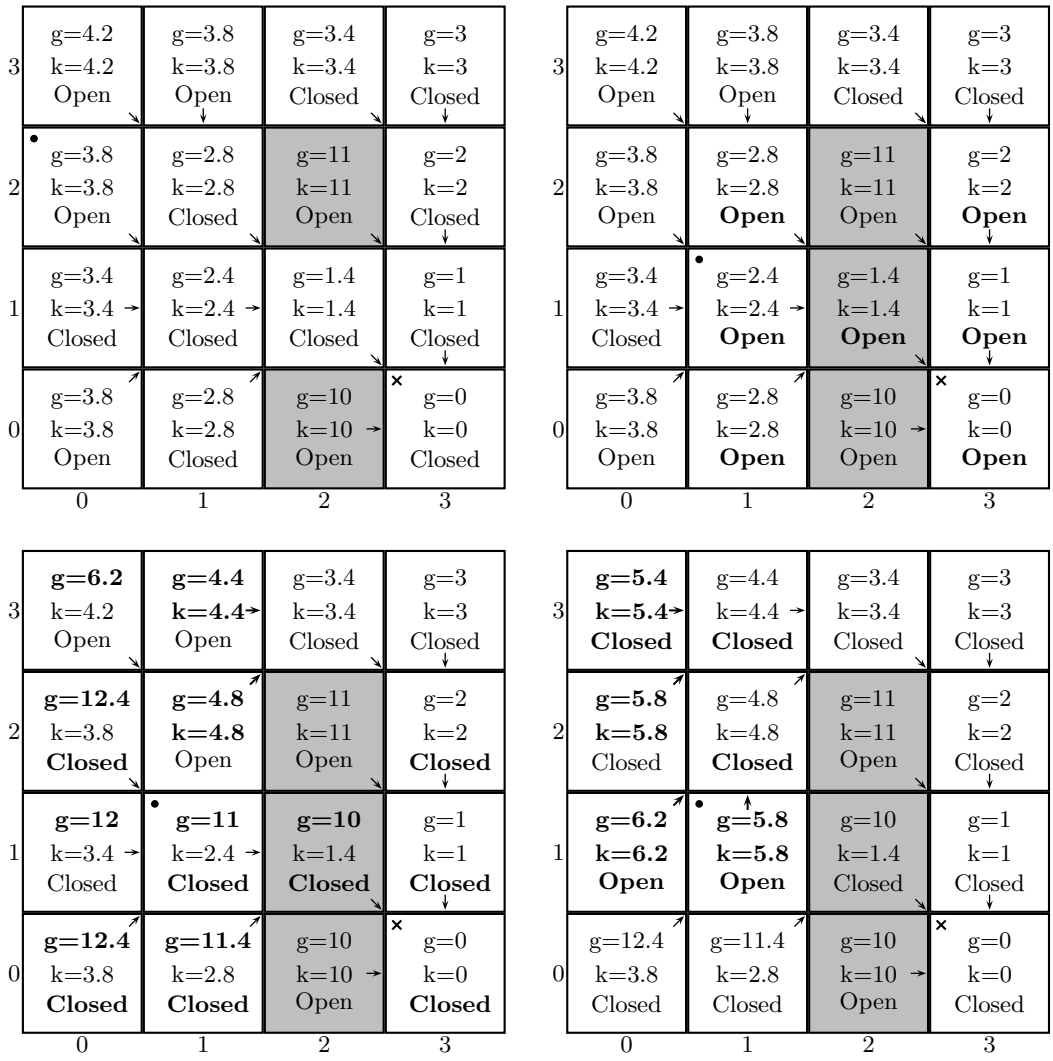

Figure 5.1: An illustrated D* example

that the cost of moving vertically, horizontally or diagonally from a closed cell to any of its neighbours is 10. Each cell indicates the direction of its back pointer with a small arrow on the edge of the cell. The cells also contain the current cost value (g) and the priority queue ordering value (k) as well as a value to indicate if the cell is open or closed. The cell which represents the robot's position contains a dot and the goal cell contains a cross. In the description below, a specific cell will be referenced to by its location in the grid, e.g. a cell at row r and column c is given by (r, c). Bold values in the grids indicate a change from the previous grid.

The first grid contains the state of the algorithm after the initial path is found. D* starts by placing the goal $(0, 3)$ on its open list. The goal is then processed which adds all of its

neighbours, $(0,2)$, $(1,2)$, and $(1,3)$, to the open list after updating their current cost values. The next cell to be processed is $(1,3)$ since it has the lowest k value. The cells on the open list are processed in a similar way until the g value of the robot's position cell is less than or equal to the smallest key in the queue. The first grid shows the state of D* after this occurs and a complete and optimal path is found. The behaviour of D* up to now is identical to Dijkstra's algorithm.

The second grid is the state of the D* algorithm after a change in the grid occurs. The cell $(1,2)$ is now blocked and it and all of its neighbours are re-opened as a result. The robot has also moved along its original path to $(1,1)$.

The third grid shows what happens after the cells have been processed. By referring to the second grid, we describe the actions of D* in response to the closing of cell $(1,2)$. The first cell to be processed after the change is the goal cell $(0,3)$. Since the cost from $(1,2)$ along its back pointer is now 10, its g value is updated appropriately. Cell $(1,3)$ is processed without any changes after which cell $(1,2)$ is processed. This causes the cost change to be propagated along the back pointers to cells $(0,1)$, $(1,1)$ and $(2,1)$. It is important to note that when the cost increase is propagated, only the g values of the cells are changed while the k values remain the same to preserve the order in which cells are processed. Cells $(2,3)$, $(1,1)$, $(2,1)$, $(0,1)$, $(1,0)$, $(0,0)$, $(2,0)$ and $(3,1)$ are processed in order with similar actions as before with the exception of the cells $(2,1)$ and $(3,1)$. These two cells each have $(3,2)$ as a neighbour which has a better cost path through it. Both their back pointers and g values are updated before processing any of their neighbours. When processing the neighbours of cells $(2,1)$ and $(3,1)$, it is found that these two cells can reduce the cost of their neighbours, but cells $(2,1)$ and $(3,1)$ currently have a higher cost than key value. If they modify their neighbours now this could cause path cycles, therefore, their k value is assigned to their g value and they are placed on the open list again. This gives us the state of the third grid.

The fourth grid shows D* after the new path has been found. Cells $(2,1)$ and $(3,1)$ have been processed again and they have modified their neighbours to go through them since they have a better cost path.

This example shows how the D* algorithm propagates changes through a graph. If there is a

change in the cost of a vertex, this change is propagated along the back pointers. If a change raises a vertex, i.e., causes its g value to be larger than its k value, then neighbouring vertices are also raised until a better cost path is found. Once a better cost path is found, the g values of vertices are lowered again. This raise-lower cycle implies that each vertex in the cycle will be processed twice as shown in Figure 3.6.

A serious problem with the D* algorithm occurs when an obstacle occupies the same vertex as the robot. The termination condition for the algorithm is that the vertex which the robot is on must have a g value less than or equal to the smallest k value of all the vertices on the open list. If the vertex which the robot occupies is blocked, the cost of moving to or from that vertex will be high and, therefore, the robot's g value will also be high. This will cause almost all of the vertices in the graph to be processed before the algorithm can terminate.

5.4 D* Lite

The D* Lite algorithm is explained next in terms of a detailed example. Figure 5.2 contains six snapshots of the D* Lite algorithm running on a 4-by-4 grid. This example uses the same notation as the D* example in Figure 5.1. Initially, there are only two blocked cells, but a third is blocked after the robot has started moving. The robot's location on the grid is indicated by a dot and the goal cell is indicated by a cross. The g, *rhs* (denoted by r), and h values are given in each cell. Although the D* Lite algorithm does not specifically maintain back pointers, the arrows in each cell indicate the neighbouring cell which has the smallest cost path through it. The heuristic used in the grid is the Diagonal Manhattan distance (see Table 4.1).

The first grid shows the state once the initial path has been planned. This is identical to how the A* algorithm would plan a path. Initially, each cell in the grid has their g and *rhs* values set to ∞ and the goal cell, $(0, 3)$, has its *rhs* value initialised to 0 and it is placed on the open list. When $(0, 3)$ is processed, its g value is set to its *rhs* value and all its neighbours are updated. When a vertex v is updated, $rhs(v)$ is set to the smallest of all values $g(w) + c(v, w)$, for all the neighbours w of v, and v is placed onto the open list if $g(v) \neq rhs(v)$. The next cells to be processed are $(1, 2)$, $(1, 1)$, $(2, 1)$ and $(2, 0)$. This is done in the same way as for $(0, 3)$. Now the calculation of the initial path is complete, since $g(v_S) = rhs(v_S)$, and since the starting

cell's key is less than the smallest key in the open list. Since the D* Lite algorithm does not use back pointers, the path from any cell v is towards the neighbour w which has the smallest $g(w) + c(v, w)$ value.

The robot now moves from $(2, 0)$ to the next cell on its path, $(1, 1)$, as shown in the second grid. Since the robot has moved, all the h values in this grid have been updated to the correct values relative to the robot's new position. In addition to this, the heuristic distance between the robot's old position and its current position, denoted by k_m, is increased to 1. After the robot has moved, the grid changes and $(1, 2)$ is now blocked. The D* Lite algorithm adapts to this change by updating each cell adjacent to the changed cell in a similar fashion to the way that the neighbours were updated in the first grid. Once all of the cells have been updated, the state is that of the second grid (top right grid). The cell $(1, 2)$ now has a *rhs* value of 10 since the cost of travelling to any of its neighbours is now 10 and the neighbour with the smallest g value is the goal. All of the cells adjacent to $(1, 2)$ are updated in a similar way. An unusual side effect of this update is that a cycle is created. The cell $(1, 1)$ has updated its *rhs* value and, therefore, its back pointer from $(1, 2)$ which has in turn updated its *rhs* value from $(1, 1)$. In addition to this, all of the cells which contained paths through the newly blocked cell $(1, 2)$ are now cut off from the goal cell.

The third grid shows the effect of $(1, 2)$ being processed. Since its g value is not greater than its *rhs* value it is not processed as before. Instead its g value is set to ∞ before all of its neighbours are updated. It is then updated in the same way as its neighbours were. By setting its g value to ∞ all of the cells which had their *rhs* updated through $(1, 2)$ will now have to set their *rhs* values by using another neighbouring cell. If all of the neighbouring cells have infinite g values the cell's *rhs* value would be set to ∞ and it would be removed from the open list which is exactly what happens with the cell $(2, 3)$.

In the fourth grid the algorithm develops further. Firstly, the cell $(1, 1)$ is processed similarly to how $(1, 2)$ was processed in the third grid. Next, $(1, 3)$ is processed as the cells were processed in the first grid. Finally, $(2, 1)$ is processed as $(1, 1)$ was. Now the group of cells which are cut off from the goal is being reduced as each cell in that group is being processed. The only cell left with a non-infinite g value is $(2, 0)$.

In the fifth grid $(2, 3)$ is processed followed by $(2, 0)$. The problem of having a group of cells which is cut off from the goal is now eradicated.

The final grid shows the state after the path from cell $(1, 1)$ to the goal is found. The cells $(3, 3)$, $(3, 2)$, $(2, 1)$ and $(1, 1)$ are processed in this given order.

This example illustrates how the D* Lite algorithm obtains a path after a change occurs. One can also see that each cell with a non-infinite g value will have neighbours with non-infinite *rhs* values. Another characteristic of the algorithm is how a raise in a grid cell is handled. Much like D* and Focused D*, D* Lite produces a raised state wave which propagates along back pointers to update cells which now have a higher cost. The main difference is that D* Lite does not attempt to update the cells with the new greater cost, but instead resets the cells cost to their initial value of infinity. The D* Lite algorithm will then process the cells until the lower cost path is found.

This chapter discussed how the Bug and A* algorithms can be adapted to work in a dynamic environment. With detailed examples, the way in which the D* and D* Lite algorithms adapt to changes was observed. The next chapter discusses the performance of these dynamic algorithms in terms of their path distance and computational cost.

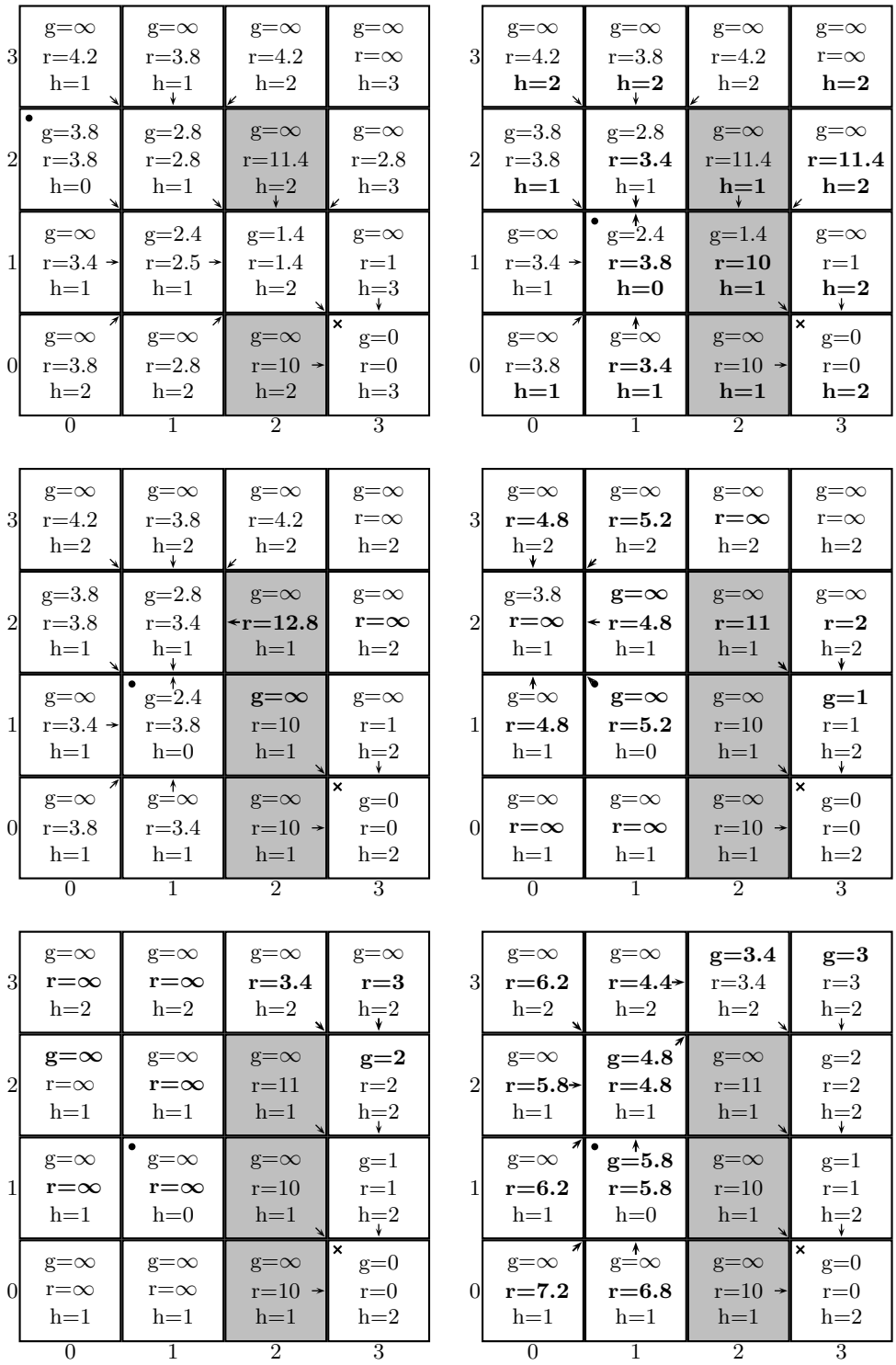

Figure 5.2: An illustrated D* Lite example

Chapter 6

Experimental Results

To evaluate the various path planning algorithms, they need to be implemented and compared with one another. One way of doing this is to implement the algorithms on a physical robot and evaluate their performance in various scenarios. Although this would show the performance in the real world, it is not ideal. The cost and effort involved in building or purchasing a physical robot can be substantial. There is also the cost of building the different evaluation scenarios which may include mechanical components to ensure that all runs are identical. Furthermore, the evaluation of the algorithms may be time consuming. Physical measurements have to be taken and scenarios have to be repeated many times.

For these reasons the path planning algorithms in this thesis are evaluated within a simulator. This allows the algorithms to be run thousands of times in many different scenarios.

6.1 Simulation method

To compare various aspects of path planning algorithms a simulation system is required. There are a number of robot simulation systems freely available. Certain simulators, such as EyeSim [5] and TeamBots [4], allow the simulation code to be run on the physical robot as well. These types of simulators require low-level control and sensing systems to move the robot around and detect its environment. This takes the focus of the system away from the goal of this thesis, which is to evaluate path planning algorithms. In addition to this, the EyeSim

simulator does not allow the user to access information about the robot, such as its position, which is essential when evaluating the simulation. The Sinbad [16] simulator was developed with a different goal in mind. It is implemented in Java and is more user-friendly. This does, however, make it less applicable to the real world. Using a Java-based system makes the timing of the algorithms less accurate and does not allow for certain optimisations which are only possible in languages such as C or C++.

Since no simulator was found which meets the requirements for this thesis, one was developed. The requirements for the simulator are discussed next. This is followed by a description of the system implemented to meet the given requirements.

6.1.1 Requirements

The simulator must simulate a number of autonomous robots which traverse a bounded rectangular environment containing a number of obstacles.

Each of the robots is modelled as a point-size robot and given the same start and goal positions. The robots must be able to use different path planning algorithms or even the same algorithm with different configurations. The robots must be implemented so that they have no direct effect on each other, that is one robot cannot block another, however, since an obstacle cannot move over a robot, robots may block obstacles which could have an effect on another robot.

Obstacles within the simulation are rectangular and unable to rotate. The obstacles move without colliding with other obstacles and the robots, and the obstacles are not allowed to move over the goal position.

The goal of the simulation is to provide an average running time for the different algorithms as well as each robot's average path length. The focus of the simulator is path planning and, therefore, it does not have to simulate the physical dynamics of a robot.

6.1.2 Design

Obstacles are represented by a position, width, and height. The obstacles continuously move to random locations within the environment. If an obstacle either reaches its target location

or it is blocked, it chooses another random location to move to.

The method each robot uses to represent the environment can be different. The choice of environment model must be compatible with the robot's algorithm. For instance, the A* algorithm uses a graph model of the environment.

The system simulates the environment by moving both the robots and the obstacles by small increments. Since the robots and obstacles may not collide with each other, the robots have no direct effect on one another.

The system runs a number of simulations with the same robot configurations and computes the average of the results. Since each simulation runs independently of the others, the total number of simulations are divided among a number of threads and run concurrently.

Modern processors are able to cache certain parts of programs which are run frequently. It is thus highly likely that one or more robots have an advantage. This occurs often with one of the robots in a simulation. The processor's choice of which robot to cache is not predictable. To avoid this unfair advantage, each robot configuration is simulated more than once and the robot data with the worst performance is used in the results.

The general layout of the simulator is given in Figure 6.1. The simulation environment contains a number of robots. Each robot is indirectly connected to an algorithm (A*, D*, ...) through an interface known as a mover. The mover facilitates the translation of the real world coordinates it receives from the robot into the environment model's native representation. The mover also queries the algorithm for the next move to make and then translates the environment model's representation of the next move back into the real world coordinates for the robot. For a grid environment representation, the mover must convert the real world position into the grid cell position and pass this on to the algorithm.

The algorithm will respond with the next grid cell position to go to and the mover must then use one of the strategies given in Figure 4.6 to determine the real world coordinate to give to the robot.

The algorithm may require the use of a heuristic function which can be any one of the heuristic subcomponents listed in Table 4.1 as well as the zero heuristic which returns zero.

The environment model has only one implementation, namely, the graph environment model. Any number of environment models can be implemented, but a corresponding mover and path planning algorithm are required.

Algorithms determine the layout of the environment through their appropriate environment model. Changes within the environment are detected by the environment model and the algorithms using that model are informed of the change.

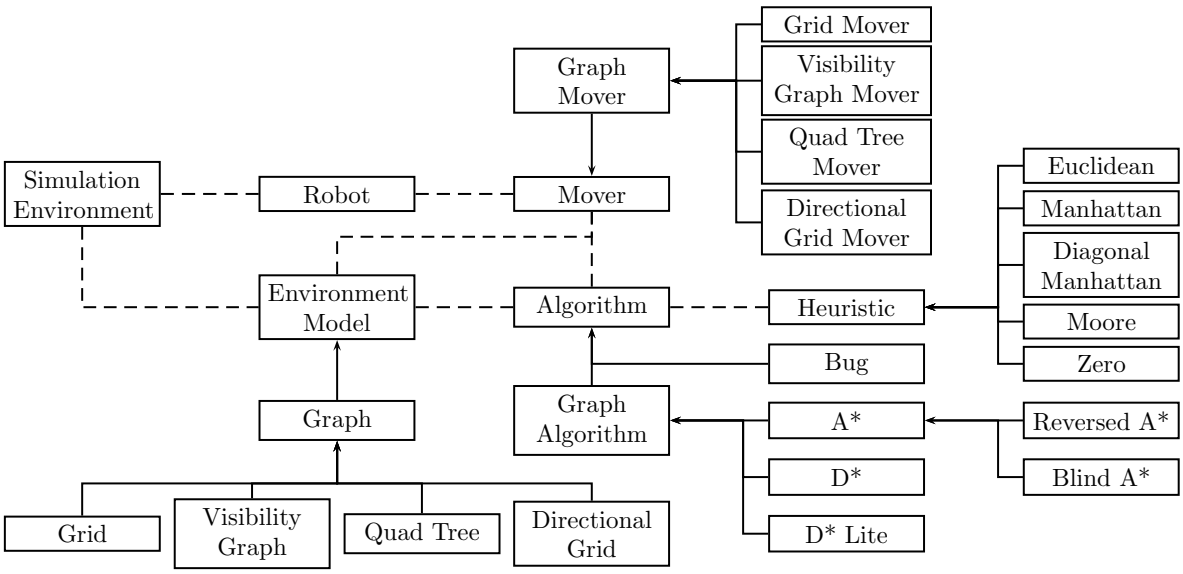

Figure 6.1: The layout of the simulator. The dashed lines indicate a relationship between the components while the solid lines indicate a component hierarchy.

The robots and obstacles are updated by allowing them to move for a fixed period of time. The robot plans its path by querying its algorithm for the next position to move to. If the robot reaches its next position before its time slice is up, the mover will query the algorithm again for the next position to move to.

6.1.3 Results

The simulator records a number of variables within the environment. Each recorded variable is aggregated across 8000 simulations. The list of variables used is given in Table 6.1. For the simulation results, two units are defined. As a distance measure, u is used. To measure the computational time, the unit mc is used which are a million CPU cycles and is measured using

the Intel RDTSC instruction. In all the simulations, the environment was $1000u$ by $1000u$ in size.

Metric	Explanation
Path Distance	The distance the robot travelled to reach the goal. This is directly related to the length of the paths the algorithm produces.
Environment Model Time	The time it takes for the environment model to be updated based on the changes in the environment.
Path Planning Algorithm Time	The average time the algorithm takes to plan a path for an entire time slice. This includes the overhead incurred by the environment model when moving between vertices.
Total Time	The total computational time required to move the robot a certain distance. This is basically the sum of the environment model time and the path planning algorithm time.
Vertex Expansions	The number of times the algorithm expanded vertices. For algorithms which do not expand vertices, this value is defined as zero.

Table 6.1: Variables used to evaluate simulations

In the simulation results, the algorithms and graph representations used are abbreviated. Table 6.2 lists all these abbreviations.

Abbreviation	Full name
Bug	Dynamic Bug
Dirgrid	Directional grid
Quad	Quad tree
RevA*	Reversed A*
VisGraph	Visibility graph using A*

Table 6.2: Abbreviations used in the results

6.2 Graph processing direction

As theorised in Section 3.1.3, the direction in which an algorithm processes its vertices may have an impact on the paths that the robot takes. To investigate the impact of the processing direction, the A* algorithm and the Reversed A* algorithm were simulated together for a number of different grid sizes. The A* algorithm processes its vertices from the start to the goal while the Reversed A* algorithm processes its vertices in the opposite direction. In the results

of the simulation, the actual computational cost of both algorithms were almost identical, although the Reversed A* algorithm performs slightly better than the A* algorithm as the grid size increases. Figure 6.2 shows the path lengths for both the A* and Reversed A* algorithms. This figure clearly shows that the A* algorithm produces shorter paths than its reversed counterpart. The difference in path length varies, but it is about 1% on average. These results clearly show that the direction in which the vertices are processed have an impact on the path length of the A* algorithm.

Another factor which plays a role in the performance of a path planning algorithm is the amount of rotation in its paths. The rotation of a path is the total change in direction which occurs on that path. Figure 6.3 shows the rotation made by a robot running the A* and Reversed A* algorithms. This figure clearly shows that the A* algorithm requires a robot to perform fewer rotations than the Reversed A* algorithm.

To investigate if the amount of rotation a robot is required to make can be reduced when using the A* or Reversed A* algorithm, they were both simulated using a directional grid. Table 6.3 shows the results of a simulation which used both the algorithms running on a grid as well as a directional grid. The zero heuristic was used for the directional grid, since it provides a fair comparison with a grid using the zero heuristic. A grid using the diagonal heuristic was also included for comparison purposes.

The A* algorithm using the zero heuristic shows a slight increase in path length when using the diagonal grid, but it produces paths that require less rotation compared with paths produced by the grid method. This is also true for the Reversed A* algorithm using the zero heuristic. The A* algorithm using the diagonal heuristic produces much shorter path distances as well as fewer rotations. The Reversed A* algorithm using the diagonal heuristic on the other hand has more rotations than when it uses the directional grid. These results show that when the A* algorithm is using an accurate heuristic, its choice of possible optimal paths is better than when it is using a less accurate heuristic. The Reversed A* algorithm on the other hand seems to make bad choices from the possible optimal paths regardless of the heuristic used. For this algorithm, the directional grid seems to reduce the amount of rotation somewhat.

Both the path distance and robot rotation results show that for the A* algorithm, it is better to

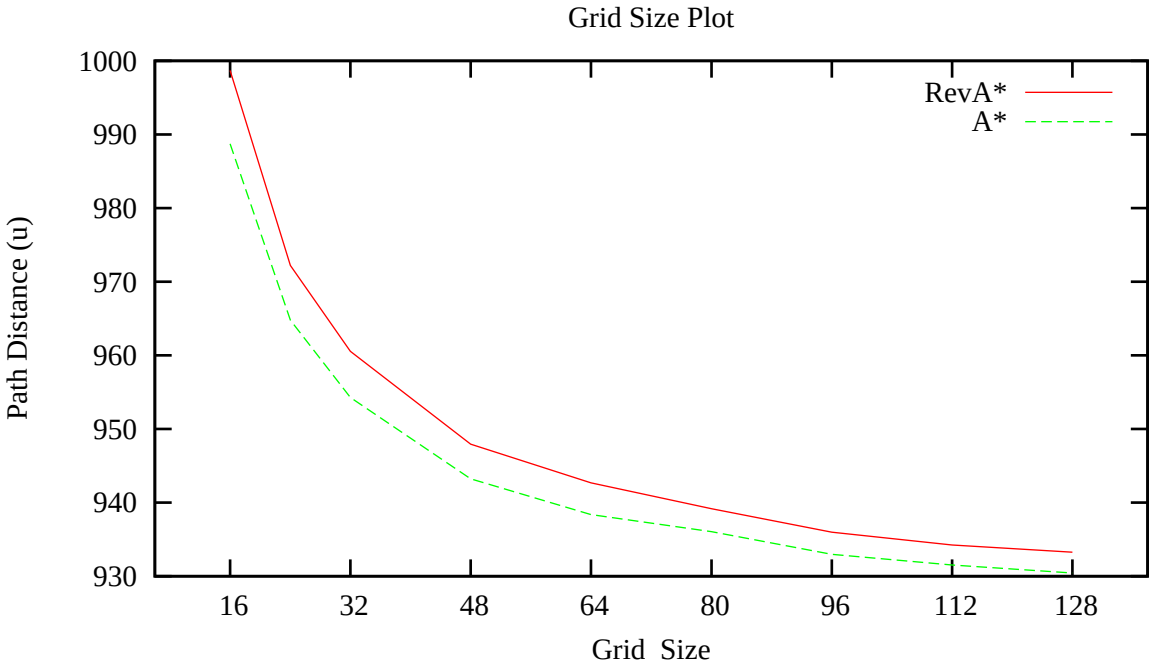

Figure 6.2: The path length for the A* algorithm processed forwards and in reverse

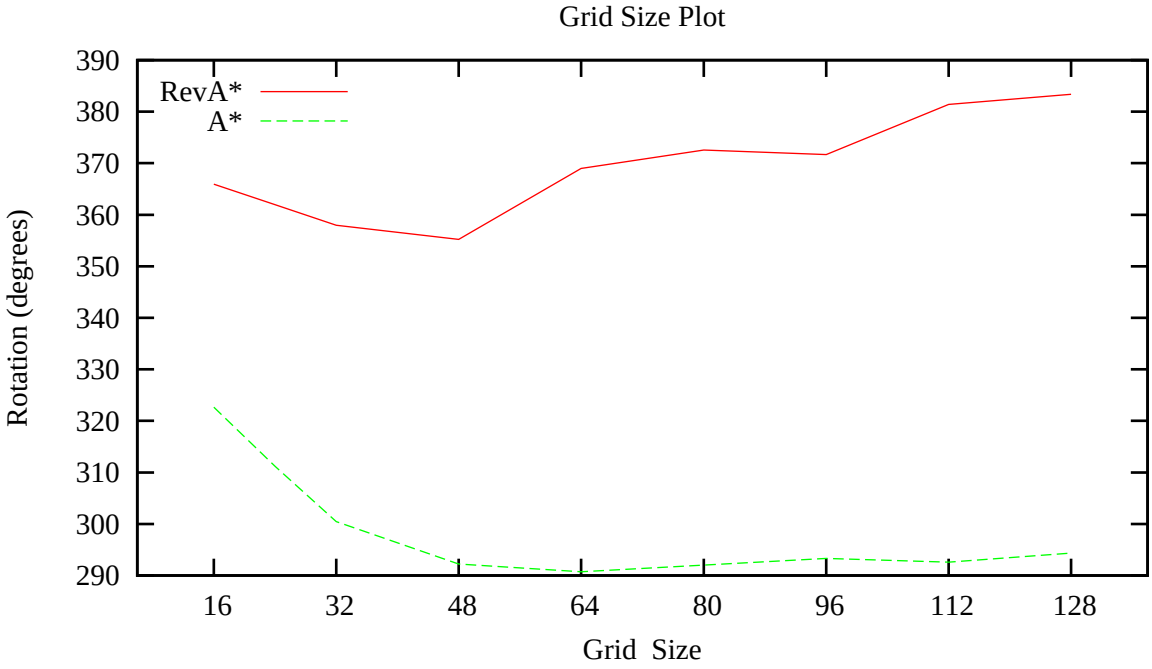

Figure 6.3: The degrees rotated by a robot for the A* algorithm processed forwards and in reverse

Algorithm	Graph	Heuristic	Path Distance (u)	Rotation (°)
A*	grid	Zero	1003.04	501.44
A*	dirgrid	Zero	1006.09	450.03
A*	grid	Diagonal Manhattan	980.33	420.65
RevA*	grid	Zero	991.14	498.87
RevA*	dirgrid	Zero	998.07	480.19
RevA*	grid	Diagonal Manhattan	991.24	500.68

Table 6.3: Simulation results for the A* and Reversed A* algorithms using a 32-by-32 grid and directional grid representations

process the vertices from the robot's position towards the goal. Consider how the A* algorithm plans its paths when processing vertices from the start towards the goal. When using a good heuristic, it will plan its path so that the distance from the goal is decreased as fast as possible and only move around obstacles once it is closer to them. This strategy appears to not only produce shorter paths in a dynamic environment, but also paths with fewer rotations.

6.3 Heuristic

The heuristic information used in the A* algorithm allows it to compute its paths with fewer vertex expansions than the Dijkstra's algorithm. Therefore, it is important that the choice in heuristic is made carefully. This section investigates a number of different heuristics. The zero heuristic is used as a base line for the results. Of the heuristics used, all are consistent and underestimate the true distance, with the exception of the Manhattan distance function. When using the Moore neighbourhood, the Manhattan distance function overestimates the exact distance. The heuristics were tested using the A* algorithm on a 32-by-32 grid representation. Two factors for each heuristic are used to evaluate their performance, namely, the distance the robot moved and the number of vertices the A* algorithm expands to calculate its path.

Table 6.4 gives the results for the heuristic comparisons. As expected the zero heuristic expands the most vertices. The path distance when using the zero heuristic is also larger than any of the other heuristics. The A* algorithm is proven to give an optimal path if a consistent heuristic is used. Since the path length of the zero heuristic is more than the others, the A*

algorithm makes a bad choice of the possible optimal grid paths when using the zero heuristic. The Euclidean heuristic on the other hand produces the shortest paths. This means that the A* algorithm makes a good choice from the possible optimal grid paths when using the Euclidean heuristic. The Manhattan heuristic is not an underestimate and, therefore, the path distance is expected to be longer and the number of vertex expansions to be fewer than the other methods [14]. This is true with the exception of the zero heuristic's path distance. The Diagonal Manhattan heuristic and the Moore heuristic have roughly the same path distances, but the Moore heuristic expands fewer than half the number of vertices the diagonal heuristic expands.

Of all the consistent heuristics tested, the Moore heuristic performs the best since it expands the fewest vertices and gives a path length which is comparable to the Euclidean path length. However, the Moore heuristic can suffer from rounding errors and as a result is not safe to use with certain algorithms such as the D* Lite algorithm. Therefore, the diagonal heuristic will be used in the remainder of the simulations, since it produces paths with similar lengths, although by using it more vertices are expanded.

	Path Distance (u)	Vertex Expansions
Zero	1000.15	15091.80
Euclidean	974.64	1490.79
Manhattan	982.47	511.24
Diagonal Manhattan	977.57	2478.95
Moore	978.13	1047.19

Table 6.4: Experimental results for different heuristics

6.4 Environment model

One key aspect which influences the performance of an algorithm is the way it models the environment. In this section, various environment representations will be compared and their performances evaluated. The Bug algorithm is the only algorithm used which does not use a graph as its environment model, and in fact it does not maintain an environment model at all. For all the graph environment models, the A* algorithm will be used. A 16-by-16 grid was used

for the grid and directional grid models while the quad tree model used a depth of 4. Note that these two have equivalent resolutions.

Table 6.5 shows the results obtained by simulating the different environment models together. In this simulation the visibility graph representation produces the shortest paths. The Bug algorithm produces the next shortest paths.

	Path Distance	Model Time Environment	Path Planning Algorithm Time	Total Time
	(u)	(mc)	(mc)	(mc)
bug	916.666	0	0.069	0.069
visgraph	879.209	0	0.165	0.165
grid_{16}	985.630	0.368	0.083	0.451
quad_4	1011.830	0.626	0.080	0.707
dirgrid_{16}	1001.980	1.221	0.903	2.124

Table 6.5: Experimental results for different graph representations

Following this are the grid, directional grid and quad tree graph representations, in this order. The results for the path lengths can easily be justified. Using visibility graphs actually gives the optimal path for the environment, since it uses the exact obstacle locations. The Bug algorithm also benefits from using the exact obstacle locations, but the algorithm does not always produce the optimal path with this information. The grid, directional grid and quad tree representations all approximate the obstacle locations by forcing them to fit on a cell of a given size. The directional grid is expected to produce longer paths than the grid since this method will minimise direction changes and, therefore, take longer paths.

An interesting result is that the quad tree approach gives the longest paths of all the representations. Even if the grid method is given the same cell movement strategy as the quad tree method, the quad tree paths are still marginally longer. The only plausible explanation for why this occurs is that an optimal path within a quad tree is actually longer than the optimal path in an equivalent resolution grid. Figure 6.4 shows a grid and quad tree representation of the same environment with a single stationary obstacle. The quad tree's path is longer than that of the grid because its cells are larger. Furthermore, the quad tree's state is the same for all depths of at least 2. This shows that although using a quad tree gives fewer cells to process, these cells are larger on average than the cells of an equivalent resolution grid, therefore, the

quad tree paths are longer. This problem can be reduced by forcing the quad tree to make the cell that the robot is on to be at the maximum depth, but this adds a high computational cost which negates the benefit of using a quad tree.

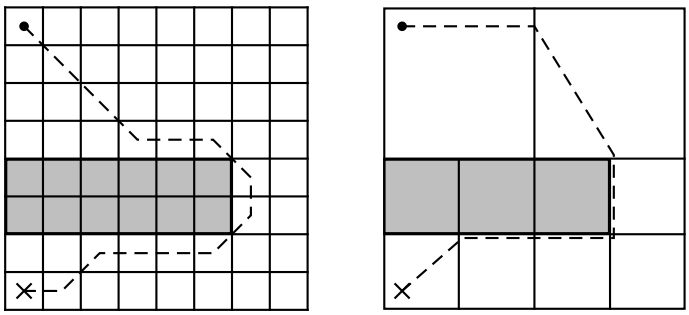

Figure 6.4: A comparison between the paths of a 8-by-8 grid (left) and a quad tree (right) with a depth of at least 2

For the environment model time, the Bug and visibility graph algorithms require virtually no computation time. This is because the Bug algorithm does not keep a model of the environment, but instead uses its sensor observations directly. The visibility graph calculations are integrated with the A* algorithm so that it models its environment while the algorithm is running and can use an incomplete environment model. The grid method requires the least amount of computation with the quad tree method requiring approximately double. The directional grid method requires the greatest amount of computation.

The path planning algorithm time shows whether or not the cost of maintaining the environment model pays off by allowing the path planning algorithm to run faster. The Bug algorithm requires the least computation even though it does not have an environment model to aid it. The visibility graph method requires roughly twice the amount of computation than the 16-by-16 grid. However, when looking at the total computation time, the grid method requires almost three times the computation time required by the visibility graph. An interesting result arises when comparing the path planning algorithm time of the grid to that of an equivalent resolution quad tree. The grid method expands 573.012 vertices on average while the quad tree only expands 210.139. Although the quad tree method expands fewer than half the vertices of the grid method, the path planning algorithm time of the quad tree is only a fraction less than that of the grid. There are only two factors which can contribute to the relatively high path planning algorithm time. The first is the cost of accessing a quad tree cell is computationally

more expensive than accessing a grid cell. Secondly, the computational cost to calculate the movement between the quad tree cells is more expensive than that of the grid cells.

The results discussed in this section lead to a number of questions. What happens when the grid and quad tree resolutions are increased? How does the number of obstacles, their size and their speed affect these solutions? The following sections will attempt to answer these questions.

6.5 Grid resolution

When using a grid representation of the environment, an important consideration is that the algorithm scales well when increasing the grid size. For this investigation, all the graph algorithms will be simulated using a number of different grid sizes. In addition to this, a quad tree representation will be used with the A* algorithm using the same resolutions as the grids where possible. To compare the grid methods with non-grid methods, the Bug algorithm and visibility graph representation with the A* algorithm are also included.

Figure 6.5 shows a graph of the average path length for each algorithm for a given grid size. The Blind A* algorithm always moves a little more than the rest of the grid algorithms since it does not compute a complete optimal path. On a 16-by-16 grid, it moves about 4.3% more than the A* algorithm and on a 128-by-128 grid the Blind A* algorithm moves only about 1.4% more than the A* algorithm. The rest of the grid algorithms move about the same distance for all grid sizes, except for the A* algorithm which moves a little bit less than the rest. This shows that even though the actual paths the graph algorithms take may be different, on average the path lengths are about the same even in a dynamic environment. The quad tree representation produces a path which is much worse than the grid representation, as was observed in the previous section.

The Bug algorithm's paths are shorter than the paths of the optimal grid algorithms up to a grid size of about 112-by-112. At larger grid sizes, the grid algorithms fare better. The path length when using a visibility graph is much less than all the tested grid sizes.

The average time each algorithm takes to plan its paths is given as a graph in Figure 6.6.

The Bug and visibility graph algorithms require the least amount of computation for almost all grid sizes. The D* algorithm performs quite well on small grid sizes, but as the grid size increases, the computational cost of the D* algorithm increases beyond that of the A* and D* Lite algorithms. The D* Lite algorithm performs better than the D* algorithm, but it still does not outperform the A* algorithm. The A* algorithm running on a quad tree performs much better than when running on a grid. In addition to this, the quad tree method scales well as the grid size increases.

Figure 6.7 shows the number of vertex expansions for each algorithm on the different grid sizes. The quad tree method requires very few expansions and this number grows at a much slower rate than the grid method. The Blind A* algorithm requires very few vertex expansions, but still more than the quad tree method. The D* Lite algorithm expands the fewest vertices when considering the optimal grid algorithms. The rate at which this increases as the grid size increases is also quite low. The A* algorithm expands many vertices, however, it expands less than the D* algorithm.

Finally the total time, which includes the cost of maintaining the environment mode, is given in Figure 6.8. The algorithms using a grid do not scale as well as the A* algorithm using a quad tree.

From these results, it is clear that the D* Lite algorithm expands fewer vertices than both the A* and D* algorithms. When looking at the actual computational cost, the D* Lite algorithm is better than the D* algorithm, but it does not perform better than the A* algorithm. The quad tree method scales quite well with the number of vertex expansions.

6.6 Obstacles

All the simulations discussed thus far have used a between two and four obstacles with side lengths between 100u and 250u. This section investigates the impact of the obstacles' size, number and speed. The obstacle speed will be given in u/t which is a unit of distance per a unit of time.

In the first series of simulations an increasing number of $35u$-by-$35u$ obstacles moving at a

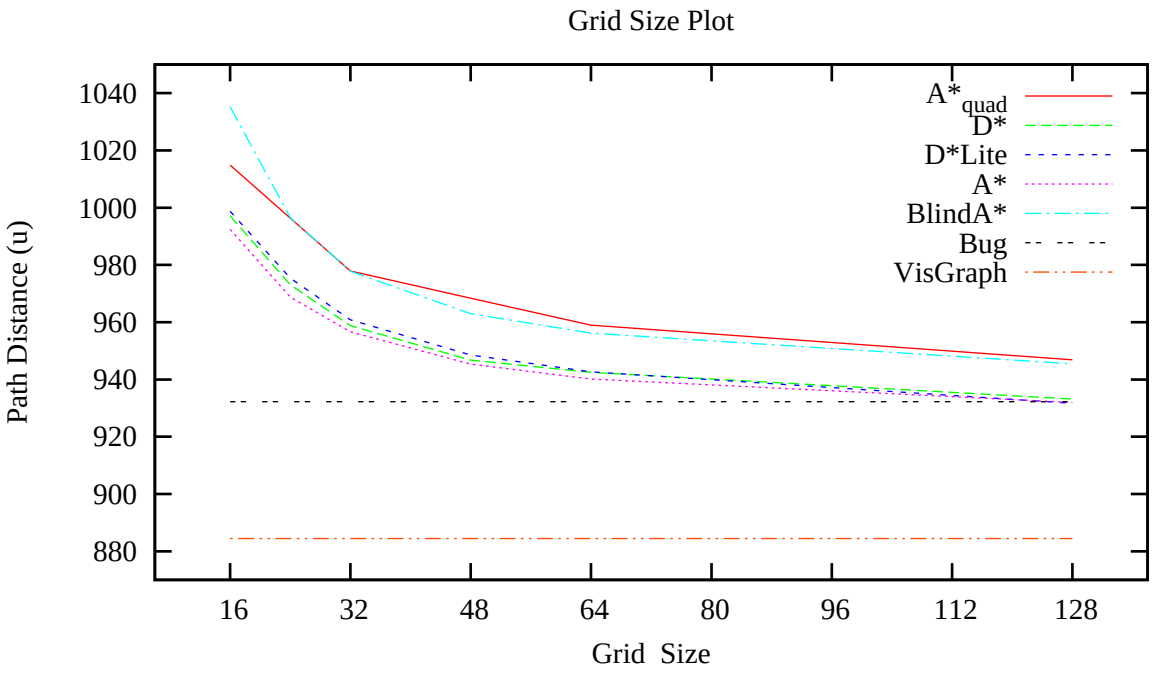

Figure 6.5: The path distance for various algorithms on different grid sizes

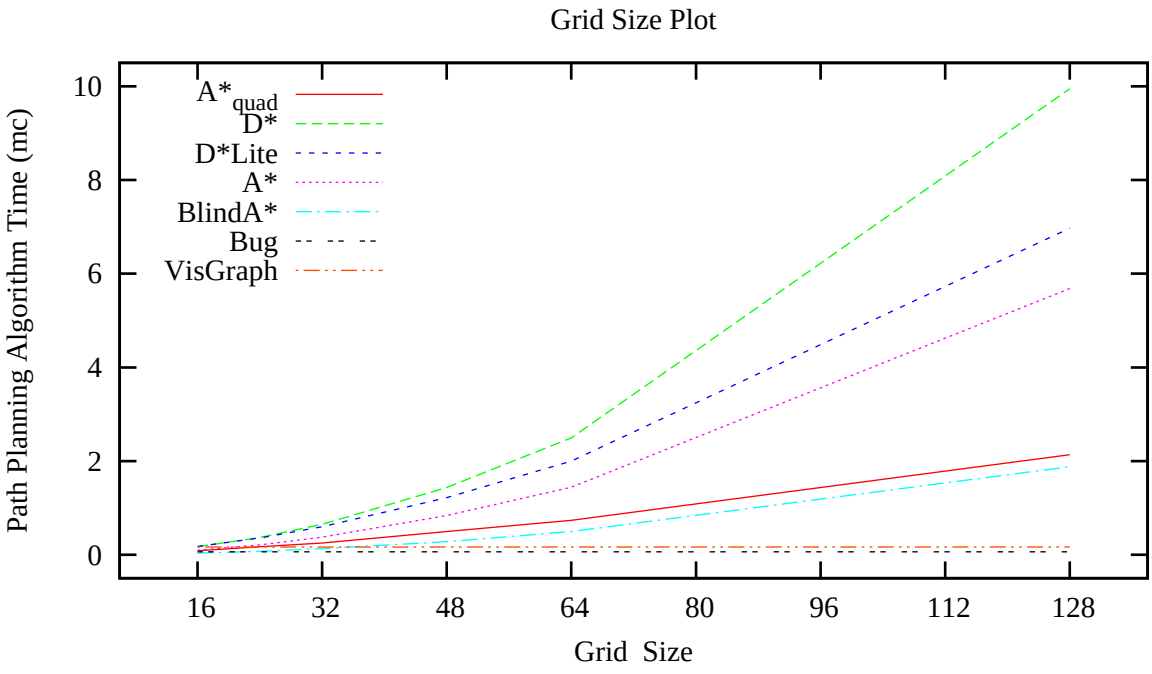

Figure 6.6: The path planning algorithm time for different grid sizes

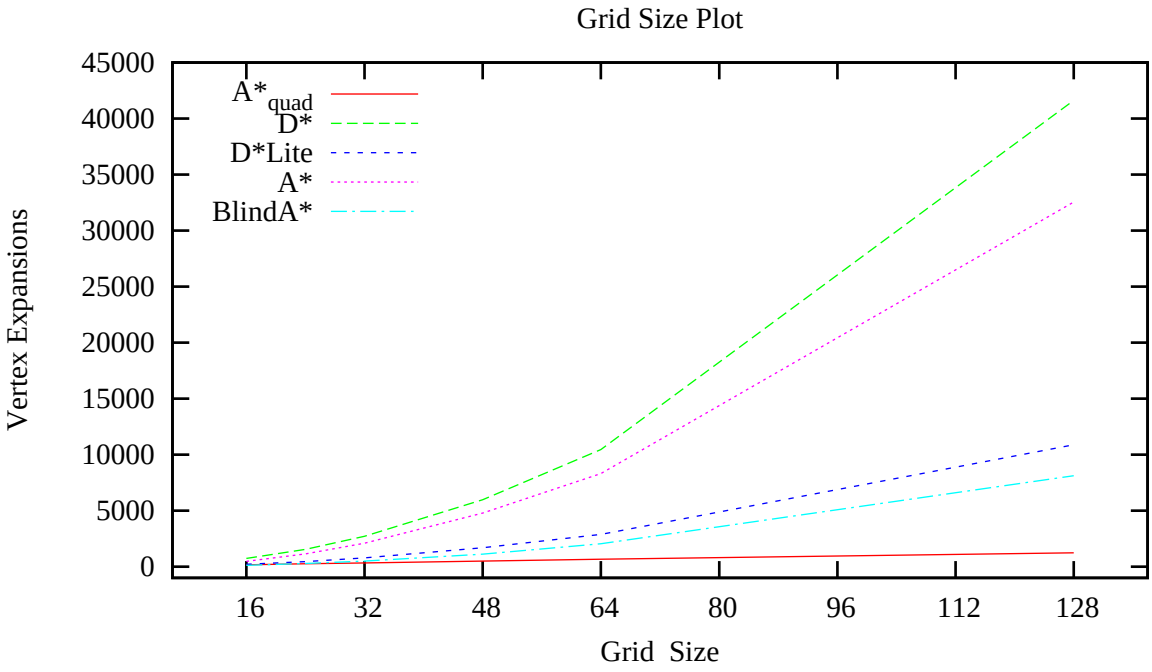

Figure 6.7: The path planning algorithm time for different grid sizes

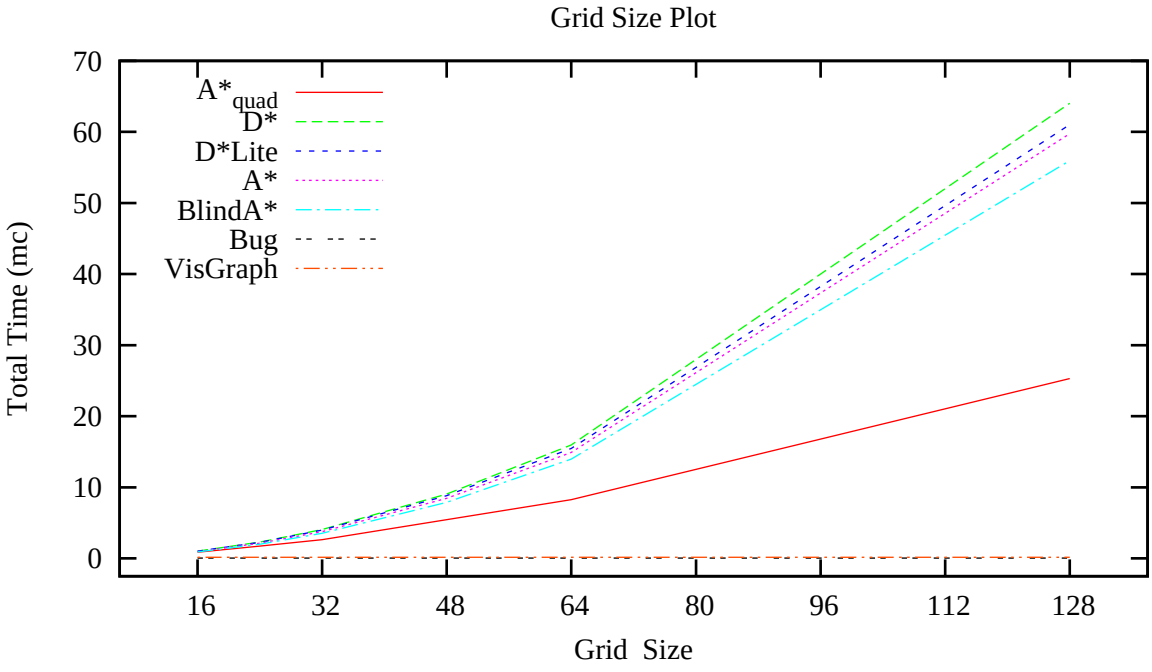

Figure 6.8: The total time for different grid sizes

speed of $1u/t$ were used. Figure 6.9 shows how the increasing number of obstacles impacts the path distance for the various algorithms. The visibility graph algorithm is almost unaffected by the increasing number of obstacles. The Bug algorithm on the other hand is the most affected algorithm and its path length increases exponentially as the number of obstacles increases. The path length when using a grid or quad tree algorithm increases as the number of obstacles increase, but not as quickly as the Bug algorithm does. In Figure 6.10 the total computational cost of each algorithm is given for the different number of obstacles. The quad tree representation does not respond well to the increasing number of obstacles. The A* and Blind A* algorithms react about the best and the Bug algorithm has the best performance over all. As expected, the computational cost for the visibility graph algorithm increases quite a bit, since its time complexity is dependant on the number of obstacles. Interestingly, the D* and D* Lite algorithms react worse to the increase in obstacles than the A* algorithm. This can be attributed to the fact that with more obstacles, there are more changes in the graphs which these algorithms must adapt to.

Next a series of simulations were run with two obstacles at a speed of $1u/t$ with an increasing obstacle sizes. The path distances for the different obstacle sizes are given in Figure 6.11. All the algorithms produce longer paths as the sizes of the obstacles increase. The paths of the Bug and Blind A* algorithms increase at a slightly higher rate than the rest of the algorithms. This result can be attributed to the fact that these are the only two suboptimal algorithms. Figure 6.12 shows the effect on the running time of the algorithms as the obstacles sizes increase. All the algorithms are generally unaffected by the increase in obstacle size with one exception. The computational cost of the quad tree method increases at a high rate as the obstacles sizes increase. This occurs since, as the area taken up by the obstacles increases, the number of leaf nodes in the quad tree also increases. This increase eventually tapers off as the maximum number of leaf nodes is reached.

The final simulations investigate the effect of the obstacles speed with four obstacles each $100u$-by-$100u$ in size. Figure 6.13 shows time taken for each algorithm to compute its path with an increasing obstacle speed. A speed of one is the same speed as the robots speed while stationary obstacles have a speed of zero. The Bug and visibility graph algorithms are the least affected by the increasing obstacle speed. The A* and Blind A* algorithms are marginally affected

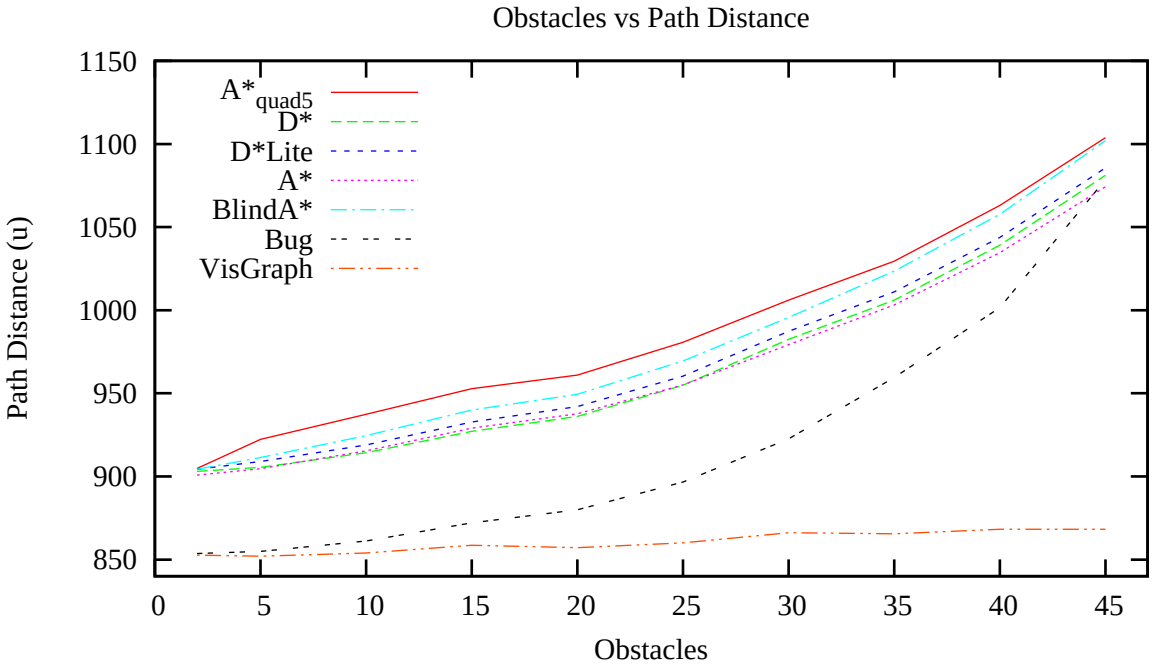

Figure 6.9: The path distance for each algorithm when increasing the number of obstacles

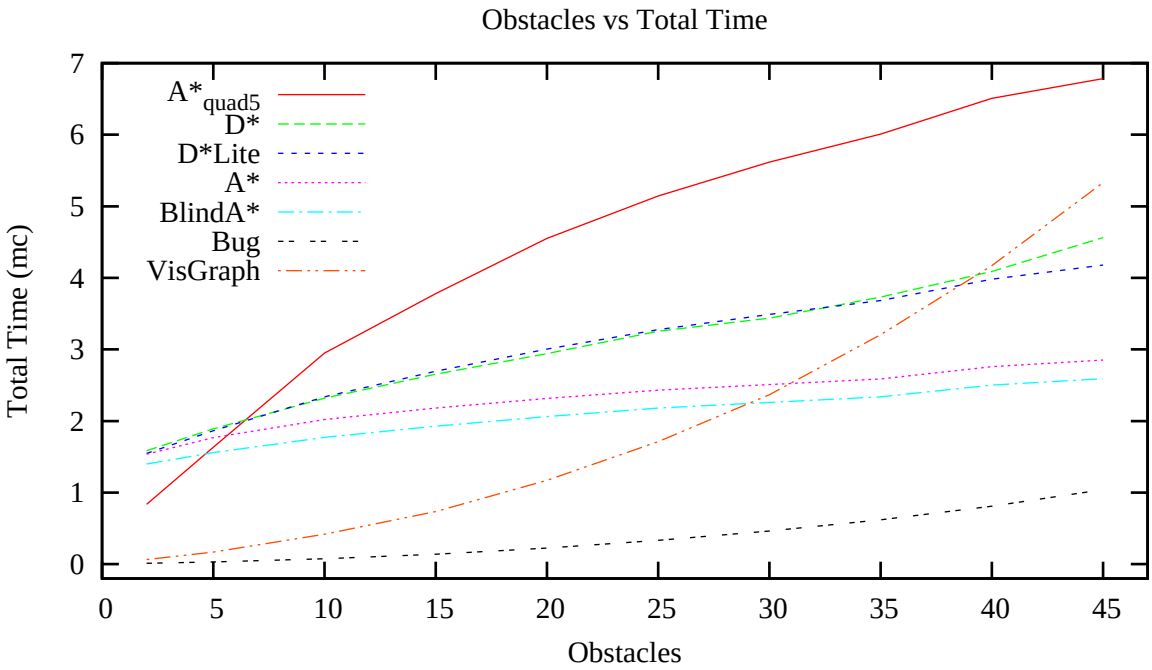

Figure 6.10: The total computation time for each algorithm when increasing the number of obstacles

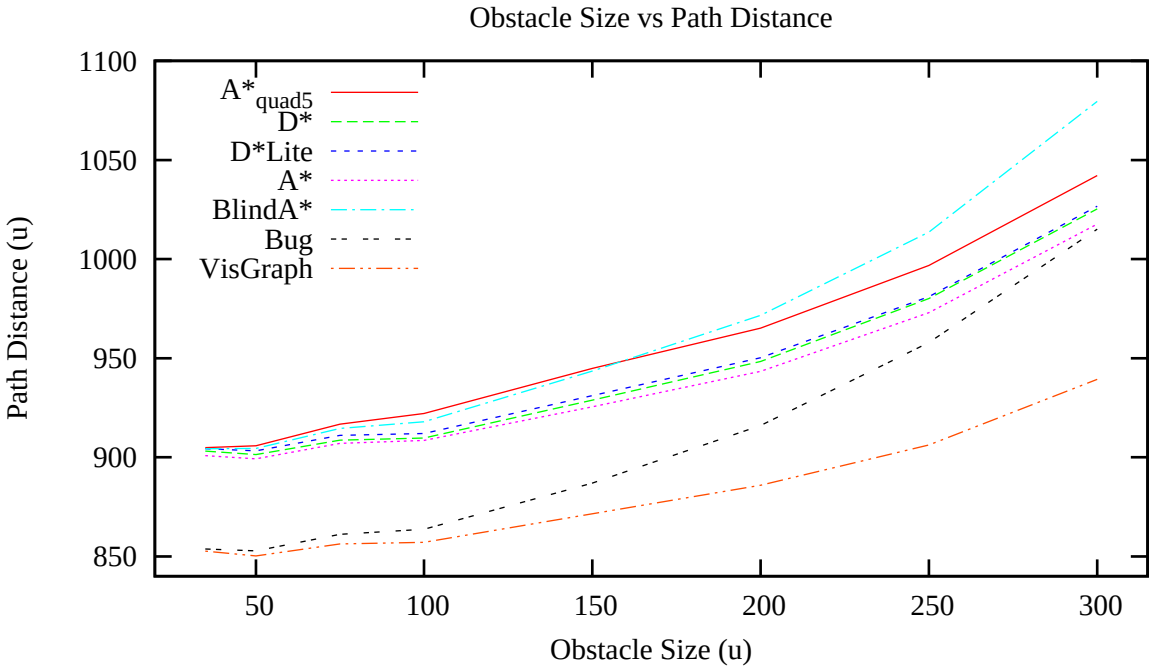

Figure 6.11: The path distance for each algorithm when increasing the obstacle area

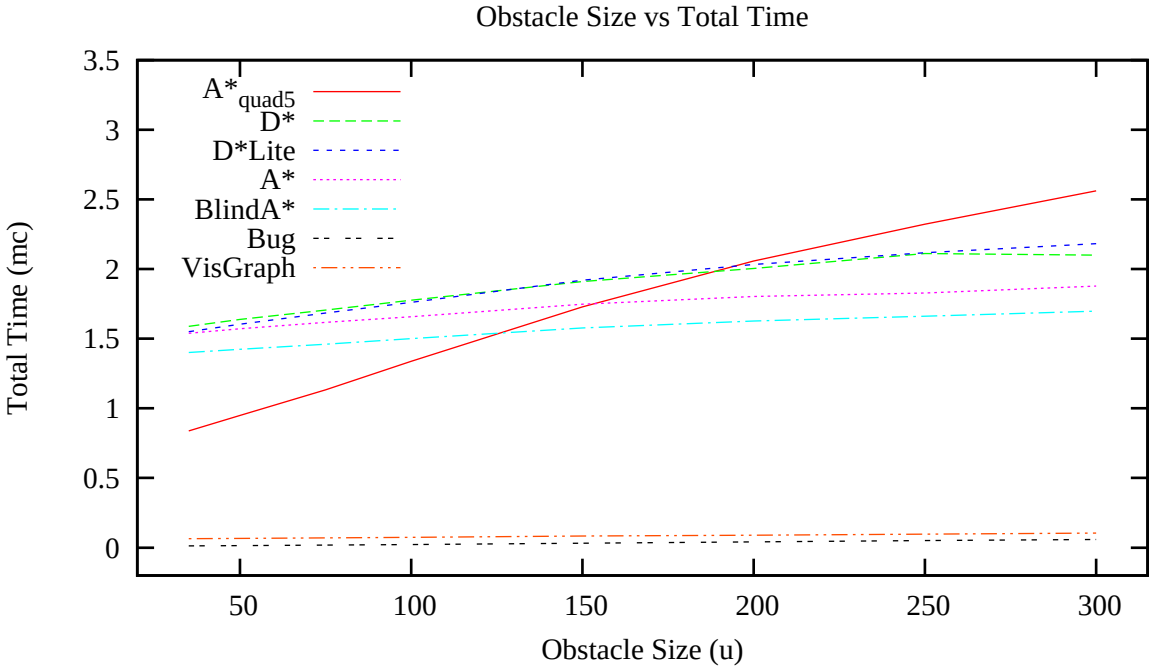

Figure 6.12: The total computation time for each algorithm when increasing the obstacle area

while the D* and D* Lite algorithms have a greater computational cost as the speed increases. As with the number of obstacles, the increase in computation time of the D* and D* Lite algorithms is due to the fact that there are more changes for these algorithms to adapt to. The quad tree representation also has a substantial increase in computation when the speed of the obstacles increase. This is due to the fact that there are more changes in the quad tree for an environment with fast moving obstacles.

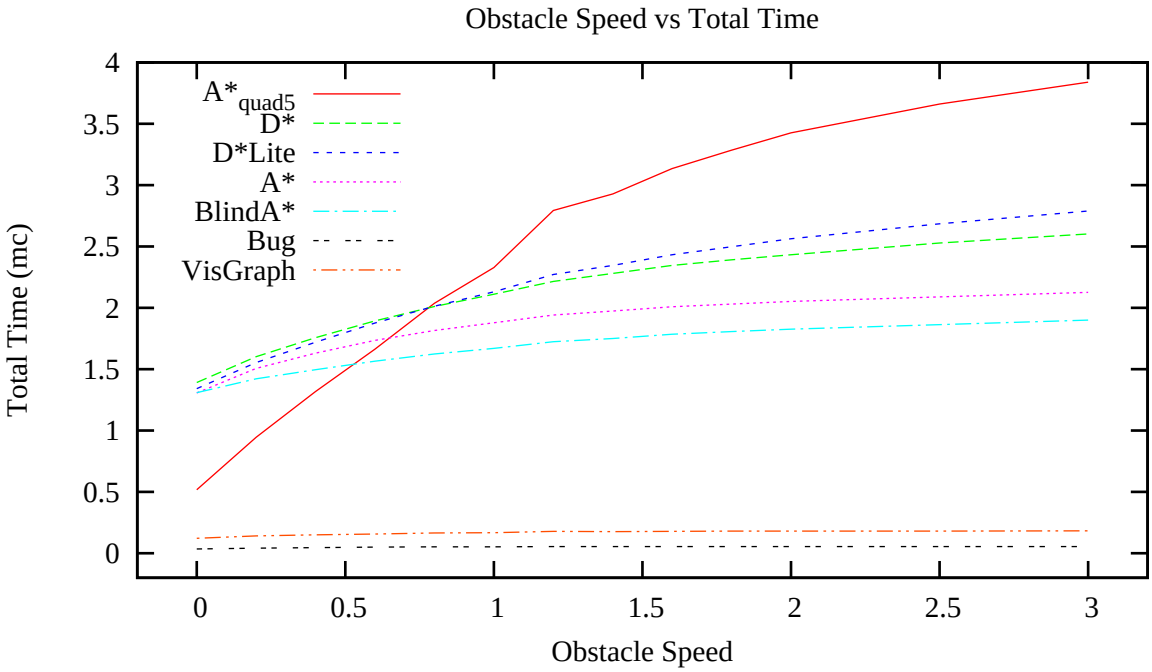

Figure 6.13: The total computation time for each algorithm when increasing the obstacle speed

6.7 Summary

The following list gives the main conclusions drawn from the experimental results and is followed by a more detailed summary.

- Using a visibility graph provides the best paths and has a low computational cost for few obstacles.
- The Dynamic Bug algorithm provides good paths for a low to medium obstacle area with

a very low computational cost.

- The A* algorithm is the best graph algorithm even in a dynamic environment.
- The D* algorithm expands far too many vertices and has a high computational cost.
- The D* Lite algorithm expands very few vertices, but still has a higher computational cost than the A* algorithm.
- The Blind A* algorithm produces a path which is on average between 1.4% and 4.3% longer than an A* path, but it has a fraction of the computational cost.
- At a certain grid resolution, the reduction in path length obtained by a further increase in grid resolution is not justified, given the increase in computational cost.
- Quad trees allow for a low computational cost, but their paths are longer than the paths of a grid.

The visibility graph using the A* algorithm always produces the best paths. In addition to this it has a very low computational cost for a small number of obstacles. When there are more than about 30 obstacles, the visibility graph's computational cost becomes more than that of the grid method.

Even though the Dynamic Bug algorithm is not optimal, it produces shorter paths than the optimal algorithms running on a grid up to a resolution of about 128-by-128. Its running time is a fraction of that of a graph algorithm. The Dynamic Bug algorithm does, however, fall short when the total area taken up by the obstacles increases. When this happens, the path distance increases at a much higher rate than any of the other algorithms. Its running time only increases when the number of obstacles increases, but it is still relatively low when compared to the other algorithms.

Since the A* algorithm does not attempt to reuse data from previous paths, it is surprising that it performs better, in terms of running time, than both the D* and D* Lite algorithms. Outperforming the D* algorithm is understandable since A* is a simpler algorithm and it expands fewer vertices than D*. The D* Lite algorithm on the other hand gives surprising results, since the D* Lite algorithm expands only a fraction of the number of vertices expanded by the

A* algorithm, but the computational cost of the A* algorithm is less. Another observation is that the D* and D* Lite algorithms perform worse than the A* algorithm in terms of adapting to many changes within the environment. The D* Lite algorithm performs much better than the D* algorithm in terms of computational cost since the D* Lite algorithm makes use of heuristics.

The Blind A* algorithm, developed by us, performs well in general. Even though it is a suboptimal variant of the A* algorithm, the path distance of paths produced by this algorithm ranges between only 1.4% and 4.3% longer than that given by the A* algorithm. The decrease in processing time is, however, substantial.

The grid resolution is an important factor. A higher resolution produces shorter paths, but it takes longer to manage the grid and to compute those paths. As the grid size increases, the gain in shorter paths becomes less, but the increase in computational cost grows. For instance, moving from a 16-by-16 grid to a 32-by-32 grid gives much shorter paths at a relatively low cost. Moving from a 64-by-64 grid to a 128-by-128 grid gives a very small improvement in terms of path distance, but the computational cost grows exponentially.

Using a quad tree gives a substantial gain in computation time, but the average path length is substantially longer than that of an equivalent resolution grid. In addition to this, since the quad tree's structure changes when the environment changes, the computational cost of the quad tree method depends on the number, size, and speed of the obstacles within the environment.

Chapter 7

Conclusion and future work

7.1 Conclusion

The goal of this thesis was to investigate a number of well-known path planning algorithms to determine under which circumstances a particular algorithm is best suited. We also set out to propose changes to existing algorithms to make them better suited for dynamic environments.

In Chapter 2 we gave our assumptions for a robot and its environment. We assumed that the robot is point-sized and that we can represent the configuration or position of the robot as a two dimensional position vector. In some cases, the robot's rotation was also considered. Since localisation and mapping is well studied, we assumed that the robot has full knowledge of its environment.

We defined the path planning problem as determining the next best move to make from the current robot configuration using the information available. Following this we explored two common graph-based path planning algorithms, namely, Breadth-First Search and Dijkstra's algorithm.

Chapter 3 investigated a number of well-known path planning algorithms. Two variations of the bug algorithm, namely, the Bug2 and Tangent Bug algorithm, were considered. The bug algorithms are in general not optimal and make their path planning decisions based on local information. The potential functions method and its associated problems, such as local

minima and path oscillations, were thoroughly explained. Next the A* algorithm by Hart *et al* was discussed. This is a graph algorithm which makes use of a heuristic function to minimise the number of vertices which have to be processed. The D* and Focused D* algorithms by Stentz were also considered. The D* algorithm is often described as Dynamic A* in literature, although it is more accurate to consider it as a dynamic version of Dijkstra's algorithm. The Focused D* algorithm, on the other hand, makes use of a heuristic function much like the A* algorithm does, and is, therefore, truly a dynamic A* algorithm. The D* algorithm is more complex than any of the previous graph algorithms and it is thoroughly explained in Appendix B. During this investigation an error in the algorithm was found and corrected. In addition to this, the algorithm was modified so that it could be applied to a directional graph. As with D*, the Focused D* algorithm is very complex. A simpler algorithm which achieves the same goal was developed by Koenig *et al*, namely, D* Lite. This algorithm is an extension of the Lifelong Planning A* algorithm so that it can be used on a mobile robot much like D*. The D* Lite algorithm itself is not based on the D* algorithm and the "Lite" portion of its name does not imply it is suboptimal. Chapter 3 concluded with a literature overview of work which compares path planning algorithms. None of the literature found gave a satisfactory comparison of path planning algorithms.

Various graph representations were discussed in Chapter 4. The most common representation is a grid which was discussed at length. Various options for the grid cell neighbourhood were given and the Moore neighbourhood was chosen as the neighbourhood to use for this thesis. A number of commonly used heuristics were also given. A number of potential problems associated with the grid representation were discussed and various solutions were given. A common oversight when using the grid representation is how a robot must actually move between grid cells. This was discussed and three solutions were given.

Following the discussion on grid representations we investigated the visibility graph representation. A visibility graph is a graph of the environment where the obstacles are represented as polygons and each corner on an obstacle is a vertex within the graph. The vertices are connected if there is a clear line of sight between them. Adding a vertex for the robot's position and another for the goal position gives a graph where by the shortest path between these two vertices gives the optimal path in the environment. The visibility graph was optimised

by integrating the A* algorithm into it so that the visibility of fewer vertices needed to be calculated.

Next, the quad tree representation, where the environment is represented in multiple resolutions, was given. This method seemed promising, but it was unclear if the overhead of maintaining the quad tree outweighs its potential benefit. Finally, the directional grid method was proposed. This method was developed to reduce the amount of rotation within the path generated by an algorithm by adding a cost to rotation.

Chapter 5 discussed various dynamic path planning algorithms and how these algorithms handle changes within the environment. The Dynamic Bug algorithm, which is original work, was presented. This algorithm is based on the Tangent Bug algorithm, but it does not attach to obstacles as the Tangent Bug algorithm does. This allows the Dynamic Bug algorithm to stop following obstacles which are moving out of its path to the goal.

Since the A* algorithm does not allow for changes in the environment, it needed to be adapted to be used in a dynamic environment. We developed rules under which A* must replan its path to preserve optimality, but without having to do so whenever there is a change. This lead to the development of another algorithm called the Blind A* algorithm, which is appropriately named since it only considers changes on its path within a certain distance. This algorithm is not optimal and was used to compare a suboptimal algorithm with an optimal one.

To gain some insight into the inner workings of the D* and D* Lite algorithms, a complete and detailed example were given for each. These examples showed how these dynamic algorithms adapted to a change within the environment.

In Chapter 6 the simulation system which was implemented to compare the path planning algorithms was discussed. Following this, a number of simulations were run and from their results conclusions were drawn.

An investigation was done into if the processing direction played a role in the paths of the A* algorithm. It was found that the A* algorithm produces shorter paths when processing the vertices from the start to the goal rather than the other way around. We also showed that the A* algorithm with a good heuristic produces a path with a minimal amount of rotation. This shows that the strategy of A* with a good heuristic, which minimises the distance to the goal

as fast as possible, is a good strategy to adopt.

Various heuristics were investigated. It was found that using an inconsistent heuristic, such as the Manhattan distance function, reduces the number of vertex expansions considerably. It does, however, increase the path distance by a relatively small factor. The Euclidean heuristic gives the shortest paths at a cost of expanding more vertices. The Diagonal Manhattan and the Moore heuristics give about the same path distance, but the Moore heuristic expands roughly 40% the number of vertices that are expanded when using the Diagonal Manhattan heuristic. For the A* algorithm, the Moore heuristic is, therefore, the best to use if speed is the biggest consideration, the Euclidean heuristic is the best to use if path distance is the biggest consideration, and the Manhattan heuristic is the best to use if speed outweighs the need for path optimality.

By comparing the various algorithms with different grid sizes as well as the number, speed and size of the obstacles, the following conclusions are drawn. The visibility graph environment model integrated with the A* algorithm should be used in all cases except where the number of obstacles is high. This combination always gives the shortest paths and does so without much computation except when there are a large number of obstacles. This means that for an application such as RoboCup soccer, the visibility graph will be a good solution since there are a fixed number of obstacles. For an application such as an outdoor explorer, certain environments such as a forest would make the visibility graph approach infeasible.

As a suboptimal algorithm, the Dynamic Bug algorithm gives surprisingly short paths. This algorithm's path length increases as the obstacles number or size increases, but its computational cost remains relatively low.

The grid is a good candidate if higher dimensional environments were to be considered. As shown with the directional grid, adding another dimension is a relatively easy task and graph algorithms can seamlessly use the higher dimensional grid. When using a grid representation of the environment, the D* algorithm performs poorly and should not be used in a practical application. The D* Lite algorithm requires a fraction of the number of vertex expansions that are required by both the D* and A* algorithms. However, due to its complicated structure and the simplicity of the A* algorithm, the D* Lite algorithm incurs a higher computational

cost. This shows that although the D* Lite algorithm does a great job at reducing the number of vertex expansions in a dynamic environment, the computational overhead incurred in doing so outweighs the potential benefit. If a grid is used and optimal paths are required, the A* algorithm is the best approach. This conclusion is supported further by the fact that the A* algorithm is more robust than the D* and D* Lite algorithms. This is due to the fact that the A* algorithm replans the entire path when it needs to as opposed to adjusting its previous path calculations. The A* algorithm is, therefore, less susceptible to rounding errors. In addition to this, the A* algorithm can use an inconsistent heuristic with minimal modifications to the algorithm.

When using the A* algorithm, it was found that the computational cost when using the quad tree representation is much lower than the grid representation when the area occupied by the obstacles is relatively low. The quad tree method performs even better by comparison at higher resolutions. However, the quad tree method performs poorly when it comes to its path distance.

The Blind A* algorithm has a relatively low computational cost which scales well with an increasing grid size. This algorithm does, however, give longer paths than the A* algorithm. On a grid size of 128-by-128, the Blind A* algorithm gives an average path distance which is about 1% longer than the A* paths, but the computational cost is only about one third that of the A* algorithm.

A general approximate rule seems to be that the simpler the path finding algorithm is, the more likely it is to perform well. This general trend is observed when comparing the A* algorithm to the D* Lite algorithm and to the Dynamic Bug algorithm.

We have therefore, in our opinion, succeeded in comparing a number of path planning algorithms in a sensible scientific fashion. We have also given conditions that are required for each of these algorithms to perform well in general and we have also proposed rules for the A* algorithm which allow it to only recompute its path when necessary. In addition to this, the Dynamic Bug algorithm was developed and its performance in a dynamic environment seems promising. Lastly, we showed that the Blind A* algorithm we developed, although suboptimal, can guide a mobile robot through a grid-represented dynamic environment with a low computational cost.

7.2 Future work

From our conclusions, the visibility graph representation integrated with the A* algorithm is a good choice for a path planning algorithm provided that the number of obstacles remains relatively low. This leads to the question whether the number of obstacles which are considered can be reduced? An investigation into this could make the visibility graph approach feasible for outdoor environments where there are many obstacles. This investigation should also include the visibility graph implementation on either a physical robot or within a simulator which simulates the physical aspects of a robot. This will determine how environment inaccuracies impact the performance of this method.

When looking at the number of vertices the D* Lite algorithm expands, the algorithm looks promising. However, since it has a complex structure, its actual computational performance is worse than that of the A* algorithm. It is our opinion that the D* Lite algorithm can be further optimised to perform better than the A* algorithm. One such optimisation would be to use back pointers within the algorithm to reduce the number of times all the neighbours of a vertex need to be examined.

The Dynamic Bug and Blind A* algorithms are simple extensions of the Tangent Bug and A* algorithms respectively. There is room for improvement and these algorithms could be extended further to produce better results.

Although it was not a goal of this thesis, the simulator which was developed could be extended to provide a software package which can be used to test and compare path planning algorithms. This would require the interfaces between the algorithms and the system to be standardised. The simulator could be extended further to fully support directional graphs as well as 3-dimensional environments. Other information, such as memory usage, can also be measured to determine the suitability of robotic path planning algorithms. The very nature of the simulator whereby thousands of random environments are considered, had the additional benefit of exposing the smallest bugs within an algorithm or its implementation.

Bibliography

[1] DARPA grand challenge [online]. October 2005. Available from: `http://en.wikipedia.org/wiki/DARPA_Grand_Challenge` [cited 2008/10/22].

[2] Robocup 2008 [online]. July 2008. Available from: `http://www.nimbro.de/` [cited 2008/10/22].

[3] Roland C. Arkin and Douglas C. Mackenzie. Planning to behave: A hybrid deliberative/reactive robot control architecture for mobile manipulation. In *Proceedings of the International Symposium on Robotics and Manufacturing*, pages 5–12, August 1994.

[4] Tucker Balch. Teambots [online]. April 2000. Available from: `http://www.teambots.org/` [cited 2008/11/10].

[5] Thomas Bräunl. Eyesim - eyebot simulator [online]. February 2008. Available from: `http://robotics.ee.uwa.edu.au/eyebot/doc/sim/sim.html` [cited 2008/11/10].

[6] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun. *Principles of Robot Motion*. The MIT Press, 2005.

[7] H. Choset and K. Nagatani. Topological simultaneous localization and mapping (SLAM): Toward exact localization without explicit localization. *IEEE Transactions on Robotics and Automation*, 17(2):125–137, April 2001.

[8] J. Crowley. Navigation for an intelligent mobile robot. *IEEE Journal of Robotics and Automation*, 1(1):31–41, March 1985.

[9] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: Part 1. *IEEE Robotics and Automation Magazine*, pages 99–108, June 2006.

[10] H. Durrant-Whyte, D. Rye, and E. Nebot. Localisation of automatic guided vehicles. In *Proceedings of the Robotics Research: The 7th International Symposium (ISRR'95)*, pages 613–625, October 1995.

[11] D. Ferguson and A. Stentz. Field D*: An interpolation-based path planner and replanner. In *Proceedings of the International Symposium on Robotics Research*, pages 239–253, October 2005.

[12] S. S. Ge and Y. J. Cui. New potential functions for mobile robot path planning. *IEEE Transactions on Robotics and Automation*, 16(5):615–620, October 2000.

[13] Felipe Haro and Miguel Torres. A comparison of path planning algorithms for omnidirectional robots in dynamic environments. In *Proceedings of the 3rd IEEE Latin American Robotics Symposium (LARS'06)*, pages 18–25, October 2006.

[14] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[15] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "A formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37:28–29, December 1972.

[16] Louis Hugues and Nicolas Bredeche. Sinbad 3d robot simulator [online]. December 2007. Available from: `http://simbad.sourceforge.net/` [cited 2009/01/30].

[17] Ishay Kamon, Elon Rimon, and Ehud Rivlin. Tangentbug: A range-sensor-based navigation algorithm. *The International Journal of Robotics Research*, 17(9):934–953, September 1998.

[18] S. Koenig and M. Likhachev. D* Lite. In *Proceedings of the AAAI Conference of Artificial Intelligence*, pages 476–483, August 2002.

[19] S. Koenig and M. Likhachev. Incremental A*. *Advances in Neural Information Processing Systems*, pages 1539–1546, December 2002.

[20] S. LaValle. *Planning Algorithms*. The Cambridge University Press, 2006.

[21] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, October 1979.

[22] Vladimir J. Lumelsky and Alexander A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(1):403–430, March 1987.

[23] M. J. Mataric. Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312, June 1992.

[24] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI Conference of Artificial Intelligence*, pages 593–598, August 2002.

[25] A. Nash, K. Daniel, S. Koenig, and A. Felner. Theta*: Any-angle path planning on grids. In *Proceedings of the AAAI Conference of Artificial Intelligence*, pages 1177–1183, April 2007.

[26] Jing Ren, Kenneth A. McIsaac, and Rajni V. Patel. Modified Newton's method applied to potential field-based navigation for mobile robots. *IEEE Transactions on Robotics*, 22(2):384–391, April 2006.

[27] E. Rimon and D. E. Koditschek. Exact robot navigation using artificial potential fields. *IEEE Transactions on Robotics and Automation*, 8(5):501–518, October 1992.

[28] L. Siklossy and M. A. Haecker. Skeleton planning spaces for non-numeric heuristic optimization. In *Proceedings of the 1974 ACM Annual Conference*, pages 187–192. ACM, 1974.

[29] A. Stentz. Optimal and efficient path planning for unknown and dynamic environments. Technical Report CMU-RI-TR-93-20, The Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1993.

[30] A. Stentz. The Focused D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, August 1995.

[31] R. Sukthankar. *Situation Awareness for Tactical Driving*. PhD thesis, The Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1997.

[32] S. Watanabe. D-Star panorama by Opportunity [online]. February 2008. Available from: `http://www.nasa.gov/mission_pages/mer/images/sol1162B-20080102a.html` [cited 2008/10/10].

Appendix A

Algorithms

A.1 Breadth first search

The breadth-first search algorithm given in Algorithm 1 is a basic graph search algorithm which calculates an optimal path in a graph in which all the edges have the same cost. This algorithm is described further in Section 2.4.1.

```
1  /* Finds an optimal path from v to w and returns it or false if none exist.  */
2  function BFS_get_path(Vertex v, Vertex w)
3  begin
4      list0 = an empty list
5      list1 = an empty list
6      list0.insert(v)
7      t(v) = VISITED
8      while not list0.isEmpty() do
9          u = list0.removeFirst()
10         foreach neighbour n of u do
11             if t(n) ≠ VISITED then
12                 list1.insert(n)
13                 t(n) = VISITED
14                 b(n) = u
15                 if n == w then  return the back pointer path from v to w
16             end
17         end
18         if list0.isEmpty() then  swap list0 and list1
19     end
20     return false
21 end
```

Algorithm 1: Breadth-First Search

A.2 Dijkstra's algorithm

Dijkstra's algorithm is given in Algorithm 2 and it obtains an optimal path in a graph with positive edge costs. This algorithm is discussed further in Section 2.4.2.

```
1  /* Finds an optimal path from v to w and returns it or determines if none exists and
      returns false */
2  function Dijkstra_get_path(Vertex v, Vertex w)
3  begin
4      /* Initialise the queue */
5      queue = an empty queue which orders its elements using their g values
6      g(v) = 0
7      t(v) = OPEN
8      queue.insert(v)
9      while not queue.isEmpty() do
10         u = queue.removeMinimum()
11         t(u) = CLOSED
12         if u == w then
13             return the back pointer path from v to w.
14         end
15         foreach neighbour n of u do
16             if t(n) ≠ CLOSED then
17                 /* Either the vertex has not been visited before, or it's g value can be
                      improved upon */
18                 if t(n) ≠ OPEN or g(n) > g(u) + c(u, n) then
19                     /* If we are performing a relaxation */
20                     if t(n) == OPEN then
21                         queue.remove(n)
22                     end
23                     g(n) = g(u) + c(u, n)
24                     b(n) = u
25                     t(n) = OPEN
26                     queue.insert(n)
27                 end
28             end
29         end
30     end
31     return false
32 end
```

Algorithm 2: Dijkstra's Algorithm

A.3 Dynamic Bug algorithm

The Dynamic Bug algorithm functions much like the Tangent Bug algorithm which is described in Section 3.1.1. A high level description of the Dynamic Bug algorithm is given in Section 5.1.

The pseudocode for the Dynamic Bug algorithm is given in Algorithm 3. The target position of the robot's range sensor in a given direction is obtained by calling `GetRangeSensorTarget`. The distance between the robot and one of its range sensor targets is always less than the maximum range of the sensor, and is also less than the distance between the robot and the goal. The `IsObstruction` function returns true if there is an obstruction in a given direction. The above two functions are used within the `GetNextTarget` function, which returns the target just past the edge of the blocking obstacle in the given direction. If the robot is surrounded by obstacles, this function will return false. The $last\_dir$ variable is used to keep the last direction in which the motion to goal mode moved in, so that an obstacle can be avoided. The initial value of this variable can be random. The shortest distance between any point on the current following obstacle and the goal is given by d_{min}. This value can be maintained in the `GetNextTarget` function. In the boundary follow mode, the `GetObstacleEdge` function is used to find the next point on the obstacle to move to so that the robot can move around the obstacle in a given direction.

The Dynamic Bug algorithm is used by calling `BugGetNext` with the goal position relative to the robot's current position. The function will return the next position to move to relative to the robot's current position.

Since the environment is assumed to be dynamic, a new position to move to will always be returned. The robot could stop moving if it is in the boundary follow mode and has moved completely around an obstacle without being able to move closer to the goal. The robot could then resume once the obstacles have moved.

One problem that could arise is if the robot is in boundary follow mode and it is moving in the same direction as the obstacle is moving. This would cause the robot to effectively follow the obstacle depending on the difference in their speed. This problem can be avoided by swapping the direction in which the robot is following the obstacle if it is found that the robot has been

moving in the same direction as the obstacle for a reasonable distance.

```
1  /* Returns the next position to move to so that the goal position can be reached.  */
2  function BugGetNext(Vector v_G)
3  begin
4      if motion to goal mode then
5          v = MotionToGoal(v_G)
6          if not v or ||v − v_G|| > ||v_G|| then
7              Switch to boundary follow mode
8          end
9      end
10     if boundary follow mode then
11         v = BoundaryFollow(v_G)
12     end
13     return v
14 end
15 /* Returns the next position to move to in motion to goal mode.  */
16 function MotionToGoal(Vector v_G)
17 begin
18     Vector t = GetRangeSensorTarget(v_G.angle)
19     if not IsObstruction(v_G.angle) then
20         return t
21     end
22     Vector t_pos = GetNextTarget(v_G.angle, POS)
23     if not t_pos then
24         Switch to boundary follow mode
25         return false
26     end
27     Vector t_neg = GetNextTarget(v_G.angle, NEG)
28     if ||t_pos|| + ||t_pos − v_G|| ≤ ||t_neg|| + ||t_neg − v_G|| then
29         last_dir = POS
30         return t_pos
31     else
32         last_dir = NEG
33         return t_neg
34     end
35 end
36 /* Returns the next position to move to in boundary follow mode.  */
37 function BoundaryFollow(Vector v_G)
38 begin
39     t = GetNextTarget(v_G.angle, last_dir)
40     if t and ||t − v_G|| < d_min then
41         Switch to motion to goal mode
42         return t
43     end
44     return GetObstacleEdge(last_dir)
45 end
```

Algorithm 3: Dynamic Bug algorithm

A.4 Rotational plane sweep algorithm

The rotational plane sweep algorithm is used to determine the edges of a visibility graph as explained in Section 4.2. The rotational plane sweep algorithm defines a `Point` to contain an x and y value. A `Vertex` contains a `Point` named *pos* for its position. The angle between the x-axis and a vertex v is given as $v.angle$.

Each obstacle is modelled as a polygon constructed out of a number vertices. For each vertex on an obstacle the left and right neighbours are defined such that if one is standing on the vertex and looking into the obstacle, then the left and right neighbours will be towards the left and right respectively. If a vertex is a corner of an obstacle, its `HasNeighbours` method will return true and it will have a reference to the obstacle as well as right and left corner neighbours (n_R and n_L respectively). It will also contain a_R and a_L which are the angles between the positive x-axis and the left and right corner neighbours respectively. The vertex at the robot's position and the goal vertex do not belong to an obstacle and, therefore, their `HasNeighbours` methods will return false.

To add the visibility graph edges, the `RotationalPlaneSweep` method is called on a vertex. This can be done on every single vertex in the graph. In the case where A* is integrated into the rotational plane sweep algorithm, `RotationalPlaneSweep` is only called on a vertex when it is processed by the A* algorithm for the first time.

The rotational plane sweep algorithm maintains a list of obstacle edges (S) which the sweep line intersects. This list is initialised using the `IsInitialEdge` method which determines if an obstacle edge is an initial edge, that is, if it intersects the x-axis. This is determined by the `IntersectsPosXAxis` method which also checks for edges on the x-axis. There are a few other criteria which `IsInitialEdge` uses to include or exclude special cases such as an edge which is on the x-axis. To determine whether a special case edge should be added or not is done by looking what `UpdateList` does to the list after it has processed all the vertices on the x-axis. After this has happened, all the edges left in S should still intersect the sweep line. Figure A.1 gives all the special cases to determine if an edge is an initial edge or not. In each case, two example obstacles are given on either side of the edge in question. In case (a), the edge in question lies on the y-axis. If we consider the sweep line rotating counter-clockwise, when it

is vertical, the edge will be either added or removed from the S list. For this reason, if the obstacle is on the right, the edge in question must be an initial edge so that when the sweep line is facing upwards, the edge will be removed from S. Using the same reasoning, the rest of the special cases can be justified.

A priority queue of vertices (E) is used to order all the vertices by the angle between the positive x-axis and the line from the sweep line origin to the given vertex. Ties are broken by the distance from the vertex to the sweep line origin.

The rotational plane sweep algorithm proceeds by processing each vertex in E in order. When a vertex v is processed, the visibility between the sweep line origin and v is determined using the `IsVisible` method. This uses the list of edges (S) which the sweep line currently intersects to determine whether a vertex is visible or not. The only edge in S which actually needs to be checked is the closest one, but to determine which is the closest one takes about the same amount of computation as checking if any of them are closer to the sweep line origin than the vertex which is being tested. If the vertex is determined to be visible, a graph edge is created for it. Some optimisation can be done here by creating the reverse edge as well and also remembering that the visibility between the two vertices has been tested and, therefore, the `IsVisible` method will not have to be called with the vertices reversed.

After the visibility has been calculated, the list S is updated by calling `UpdateList`. This method simply looks at the obstacle edges adjacent to the current vertex. If the edge is in S it is removed, otherwise, it is added.

If the list S is implemented as a double linked list and all the edges maintain a reference to their position in S, the time complexity of the `UpdateList` function is $O(1)$. The `IsVisible` function has a time complexity of $O(m)$ where m is the number of edges in the list S. The time complexity of the `RotationalPlaneSweep` function is, therefore, $O(mn \log n)$, where n is the total number of vertices in the visibility graph, provided that the vertex priority queue can remove the first element with a time complexity of $O(\log n)$. Since the number of edges in S is at most the number of obstacle edges which in turn is equal to $n - 2$, the time complexity can be written as $O(n^2 \log n)$.

If the obstacles have a fixed number of edges, then n is a constant multiple of the number of

obstacles. This gives a time complexity of $O(k^2 \log k)$ where k is the number of obstacles.

The time complexity shows that the algorithm may not scale well when the number of obstacles is increased.

```
1  /* Returns true if the line from p_0 to p_1 intersects the positive x-axis.  */
2  function IntersectsPosXAxis(Point p_0, Point p_1)
3  begin
4      /* If both y values are on the same side of the x-axis but not touching it */
5      if (p_0.y > 0 and p_1.y > 0) or (p_0.y < 0 and p_1.y < 0) then
6          return false
7      end
8      /* Both x values are greater or equal to zero */
9      if p_0.x ≥ 0 and p_1.x ≥ 0 then
10         return true
11     end
12     /* Both x values are less than zero */
13     if p_0.x < 0 and p_1.x < 0 then
14         return false
15     end
16     /* If both points are on the x-axis */
17     if p_0.y = 0 and p_1.y = 0 then
18         /* It intersects if either x values are non-negative */
19         return p_0.x ≥ 0 or p_1.x ≥ 0
20     end
21     /* Must calculate exact x-axis intersection point */
22     p = p_1 − p_0
23     return (p_0.x + p.x ∗ |p_0.y/p.y|) >= 0
24 end
```

Algorithm 4: Line intersects the positive x-axis

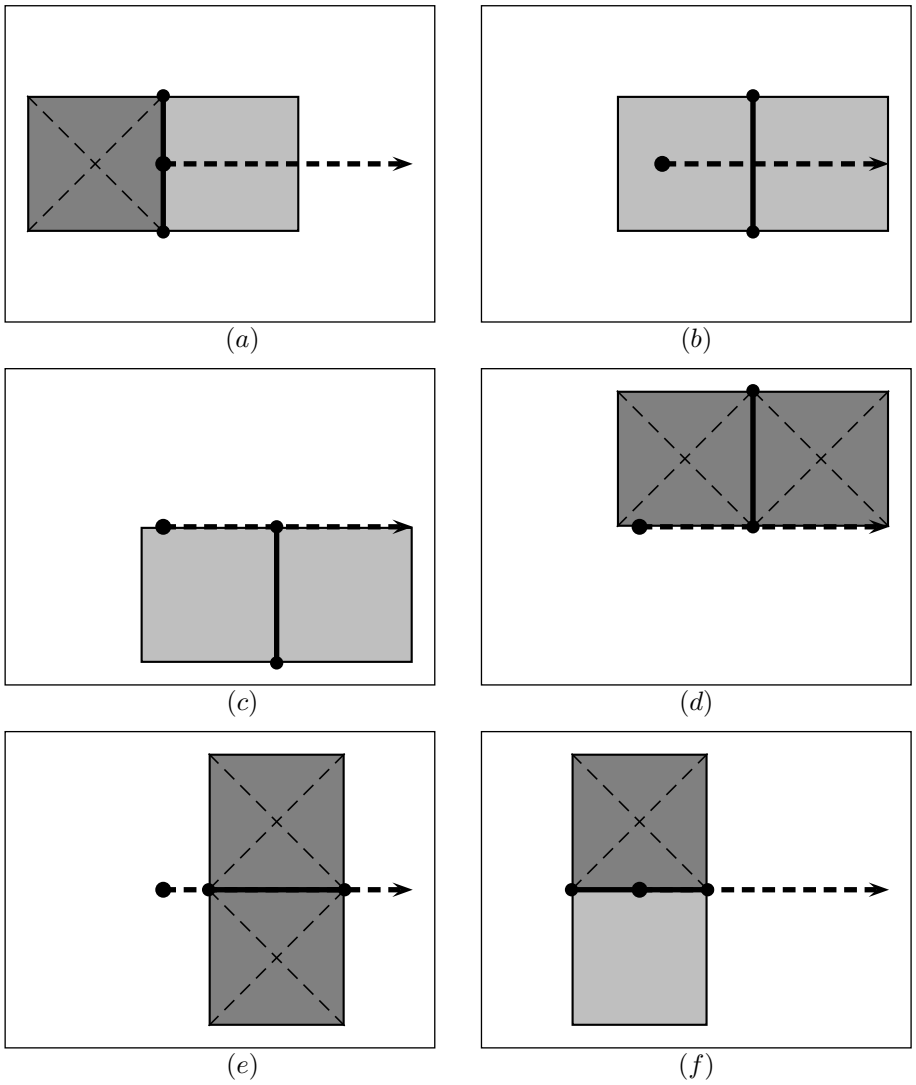

Figure A.1: Rotational plane sweep algorithm initial visibility special cases. The dashed line is the sweep line with the dot being its origin. The edge which the initial visibility is being determined is given as a bold line with two dots on the end points. A square which is filled in light grey is an obstacle who's edge will be initially visible while the square filled in dark grey and crossed is an obstacle who's edge will not be initially visible. Each special case corresponds to a check in `IsInitialEdge`. The lines for the cases are: (a) line 8, (b) line 13, (c) line 17, (d) line 21, (e) line 25 and (f) line 29.

```
1  /* Returns true if the edge from p_0 to p_1 is a valid initial edge.  The edge belongs
      to the obstacle centred at point p.  */
2  function IsInitialEdge(Point p, Point p_0, Point p_1)
3  begin
4      if not IntersectsPosXAxis(p_0, p_1) then
5          return false
6      end
7      /* If both points are on the y-axis */
8      if p_0.x = 0 and p_1.x = 0 then
9          /* Add only if the obstacle is on the positive x-axis */
10         return p.x >= 0
11     end
12     /* If neither point is on the x-axis */
13     if p_0.y ≠ 0 and p_1.y ≠ 0 then
14         return true
15     end
16     /* One point lies on the x-axis, so if the other is below it, we can add the edge
          */
17     if p_0.y < 0 or p_1.y < 0 then
18         return true
19     end
20     /* Now one point lies on the x-axis and neither are below it, therefore, if one is
          not equal, it must be above it and we must not add it.  */
21     if p_0.y ≠ 0 or p_1.y ≠ 0 then
22         return false
23     end
24     /* Both points are on the x-axis, only add the edge if one of the points is on the
          non-positive x-axis.  */
25     if p_0.x > 0 and p_1.x > 0 then
26         return false
27     end
28     /* One point is on the positive x-axis and the other is on the non-positive
          x-axis.  Only add the edge if the obstacle is below the x-axis.  */
29     if p.y > 0 then
30         return false
31     end
32     return true
33 end
```

Algorithm 5: Is a line an initial edge for the rotational plane sweep algorithm.

```
1  /* Returns if the line w → v with the given angle is inside the obstacle which v
      belongs to.  */
2  function IsLineInObstacle(Vertex v, Vertex w, Real angle)
3  begin
4      /* Tests if v is actually part of an obstacle and that its not adjacent to w.  */
5      if not v.HasNeighbours() or v.n_R = w or v.n_L = w then
6          return false
7      end
8      /* Must also ensure that v.a_R ≤ v.a_L and subtract 2π if it is not.  */
9      return v.a_R ≤ angle ≤ v.a_L
10 end
```

Algorithm 6: Is a line on the inside of an obstacle

```
1  /* Returns if the vertex w is visible from the vertex v which is the origin of the
      sweep plane.  */
2  function IsVisible(EdgeList S, Vertex v, Vertex w)
3  begin
4      /* If the vertex lies on the inside of an obstacle */
5      if IsLineInObstacle(v, w, w.angle) or IsLineInObstacle(w, v, w.angle − π) then
6          return false
7      end
8      foreach Edge e_0 → e_1 ∈ S do
9          if w ≠ e_0 and w ≠ e_1 then
10             /* Using a line intersection formula to determine visibility.  */
11             b = w.pos − v.pos
12             d = e_1.pos − e_0.pos
13             denom = d^⊥ · b
14             /* The edge is on the line v → w */
15             if denom = 0 then
16                 /* The line v → w is blocked if e_0 or e_1 is closer to v than w */
17                 if w.distance > e_0.distance or w.distance > e_1.distance then
18                     return false
19                 end
20             else
21                 c = (v.pos − e_0.pos)^⊥
22                 t = (c · d)/denom
23                 if t > 0 and t <= 1 then
24                     return false
25                 end
26             end
27         end
28     end
29     return true
30 end
```

Algorithm 7: Is a point visible from another one

```
1  /* Updates the list of edges which the sweep line intersects.  w is the current
      vertex the sweep line intersects.  */
2  function UpdateList(EdgeList S, Vertex w)
3  begin
4      if not w.HasNeighbours() then
5          return
6      end
7      if w → w.n_R ∈ S then
8          S.Remove(w → w.n_R)
9      else
10         S.Add(w → w.n_R)
11     end
12     if w.n_L → w ∈ S then
13         S.Remove(w.n_L → w)
14     else
15         S.Add(w.n_L → w)
16     end
17 end
```

Algorithm 8: Updates the rotational plane intersection list

```
1  /* Runs the rotational plane sweep algorithm on a vertex and adds the graph edges to
      vertices which are visible.  */
2  function RotationalPlaneSweep(Vertex v)
3  begin
4      /* The queue E is sorted by w.angle and then by w.distance for each w ∈ E.  */
5      VertexPriorityQueue E
6      EdgeList S
7      foreach Vertex w ∈ V\{v} do
8          w.angle = angle between v → w and the x-axis
9          w.distance = distance from v to w
10         E.Add(w)
11         /* Add the initial edges to S */
12         if w.HasNeighbours() then
13             if IsInitialEdge(w.obstacle.pos − v.pos, w.pos − v.pos, w.n_R.pos − v.pos) then
14                 S.Add(w → w.n_R)
15             end
16         end
17     end
18     while E.Size() > 0 do
19         Vertex w = E.RemoveFirst()
20         if IsVisible(S, v, w) then
21             v.AddEdge(w)
22         end
23         UpdateList(S, w)
24     end
25 end
```

Algorithm 9: Rotational plane sweep algorithm

A.5 A*

Algorithm 10 shows the A* algorithm which uses a heuristic function to produce an optimal path with the fewest number of vertex expansions. This algorithm is discussed in detail in Section 3.1.3.

```
1  /* Finds an optimal path from v_S to v_G and returns it or determines if none exists
      and returns false */
2  function AStar_get_path(Vertex v_S, Vertex v_G, HeuristicFunction h)
3  begin
4      /* Initialise the labels */
5      foreach u ∈ V do t(u) = NEW
6      /* Initialise the queue */
7      Q = an empty queue which orders its elements using their f(v) = g(v) + h(v, v_G) values
8      g(v) = 0
9      t(v) = OPEN
10     Q.insert(v)
11     while not Q.isEmpty() do
12         u = Q.removeMinimum()
13         t(u) = CLOSED
14         if u == v_G then
15             return the back pointer path from v_S to v_G.
16         end
17         foreach neighbour n of u do
18             if t(n) ≠ CLOSED then
19                 /* Either the vertex must not have been visited before, or it can be
                      improved */
20                 if t(n) ≠ OPEN or g(n) > g(u) + c(u, n) then
21                     /* If we are performing a relaxation */
22                     if t(n) == OPEN then
23                         queue.remove(n)
24                     end
25                     g(n) = g(u) + c(u, n)
26                     b(n) = u
27                     t(n) = OPEN
28                     queue.insert(n)
29                 end
30             end
31         end
32     end
33     return false
34 end
```

Algorithm 10: A* Algorithm

A.6 D*

The D* algorithm is a dynamic algorithm which can be considered as a dynamic version of Dijkstra's algorithm. Appendix B gives the original algorithm which is more complicated than it needs to be. By removing the previous cost values (p) and integrating their use into the priority queue key value (k), the D* algorithm can be simplified. The `insert` function becomes more versatile and is now responsible for setting both the k and g values. This requires both the `modifyCost` and `processVertex` functions to be modified. These changes are incorporated in the functions below. The `runAlgorithm` and `getNext` functions do not need to be modified at all and only line 8 of `initialise` needs to be changed from $p(v_G) = 0$ to $k(v_G) = 0$. The D* algorithm is explained further in Section 3.1.4 and a detailed example is given in Section 5.3.

```
1  function insert(Vertex v, Real g_new)
2  begin
3      /* If the vertex is already in the open list, remove it first */
4      if t(v) = NEW then
5          k(v) = g_new
6      else if t(v) = OPEN then
7          REMOVE(v)
8          k(v) = min(k(v), g_new)
9      else if t(v) = CLOSED then
10         k(v) = min(g(v), g_new)
11     end
12     g(v) = g_new
13     INSERT(v)
14     t(v) = OPEN
15 end
```

Function `insert`(Vertex v, Real g_{new})

```
1  function modifyCost(Vertex v, Vertex w, Real cost)
2  begin
3      c(v, w) = cost
4      if t(w) = CLOSED then
5          insert(w, g(w))
6      end
7  end
```

Function `modifyCost`(Vertex v, Vertex w, Real *cost*)

```
1  function processVertex()
2  begin
3      /* Get the next vertex to process */
4      v = REMOVEMIN()
5      if v = null then return false
6      t(v) = CLOSED
7      k_old = k(v)
8      /* Reduce g(v) by lowest cost neighbour if possible */
9      foreach w ∈ out(v) do
10         if t(w) = CLOSED and g(w) ≤ k_old and g(v) > g(w) + c(v, w) then
11             b(v) = w
12             g(v) = g(w) + c(v, w)
13         end
14     end
15     /* Process each incoming neighbour of v */
16     foreach w ∈ in(v) do
17         /* Propagate cost to NEW vertex */
18         if t(w) = NEW then
19             b(w) = v
20             insert(w, g(v) + c(w, v))
21         /* Propagate cost change along the back pointer */
22         else if b(w) = v and g(w) ≠ g(v) + c(w, v) then
23             insert(w, g(v) + c(w, v))
24         /* Reduce cost of neighbour if possible */
25         else if b(w) ≠ v and g(w) > g(v) + c(w, v) then
26             if k(v) = g(v) then
27                 b(w) = v
28                 insert(w, g(v) + c(w, v))
29             else if t(v) = CLOSED then
30                 insert(v, g(v))
31             end
32         end
33     end
34     if t(v) = CLOSED then
35         /* Set up cost reduction by neighbour if possible */
36         foreach w ∈ out(v) do
37             if b(w) ≠ v and g(v) > g(w) + c(v, w) and t(w) = CLOSED and g(w) > k_old then
38                 insert(w, g(w))
39             end
40         end
41     end
42     return true
43 end
```

Function `processVertex`

A.7 D* Lite

The D* Lite algorithm is given in Algorithm 14 and is a dynamic version of the A* algorithm. This algorithm is discussed in section 3.1.6 and a detailed example is given in Section 5.4.

```
1  function calculateKey(Vertex v)
2    return [min(g(v), rhs(v)) + h(v_S, v) + k_m; min(g(v), rhs(v))]
3  function initialise()
4  begin
5      U = ∅; k_m = 0
6      foreach v ∈ V do rhs(v) = g(v) = ∞
7      rhs(v_G) = 0; U.insert(v_G, calculateKey(v_G))
8  end
9  function updateVertex(Vertex v)
10 begin
11     if v ≠ v_G then rhs(v) = min_{w∈out(v)}(c(v, w) + g(w))
12     if v ∈ U then U.remove(v)
13     if g(v) ≠ rhs(v) then U.insert(v, calculateKey(v))
14 end
15 function computeShortestPath()
16 begin
17     while U.topKey() < calculateKey(v_S) or rhs(v_S) ≠ g(v_S) do
18         k_old = U.topKey()
19         v = U.pop()
20         if k_old < calculateKey(v) then
21             U.insert(v, calculateKey(v))
22         else if g(v) > rhs(v) then
23             g(v) = rhs(v)
24             foreach w ∈ in(v) do updateVertex(w)
25         else
26             g(v) = ∞
27             foreach w ∈ in(v) ∪ {v} do updateVertex(w)
28         end
29     end
30 end
31 function main()
32 begin
33     v_last = v_S
34     initialize()
35     computeShortestPath()
36     while v_S ≠ v_G do
37         /* if (g(v_S) = ∞) then there is no known path */
38         v_S = arg min_{w∈out(v_S)}(c(v_S, w) + g(w))
39         Move to v_S
40         if there are changes to the environment then
41             k_m = k_m + h(v_last, v_S)
42             v_last = v_S
43             foreach directed edges (v, w) with changed edge costs do updateVertex(v)
44             computeShortestPath()
45         end
46     end
47 end
```

Algorithm 14: D* Lite Algorithm

Appendix B

The D* algorithm

B.1 Introduction

The D* algorithm was presented in [29]. Section 3.1.4 gives a brief overview of the algorithm. This appendix will give a detailed description of the algorithm with illustrated examples that will explain its various aspects and also provide a proof of correctness. The original presentation of D* is complex, difficult to understand and lacks sufficient examples to explain the algorithm effectively.

As mentioned in Section 3.1.4, D* does not actually use heuristics as is the case with A*, and the heuristic value (h) used in [29] is in fact equivalent to the current cost (g) values of A*. To remain consistent and avoid confusion, this thesis substitutes the heuristic value (h) of the original article with the current cost value (g).

A big part of this appendix is similar to the original D* article, but using the symbols and syntax of this paper. There are, however, a couple of significant changes. One of these is that the algorithm and its proof is presented for a directed graph, which requires some alterations to the original algorithm.

B.2 Definitions

The D* algorithm operates on a directed graph $G(V, E)$ (see Section 2.4 for the graph definitions) with the goal vertex denoted by v_G. The edge cost of moving from a vertex v to a neighbouring vertex w is given by $c(v, w)$. This definition of cost, as well as many others, are different to the original D* article which defines the parameters in the reverse order. An open list Q, which is a priority queue, is used to store all the vertices which are to be processed. Each vertex has an associated tag, $t(v)$, which is *NEW* if the vertex has never been in the open list, *OPEN* if the vertex is currently in the open list, and *CLOSED* if the vertex has been removed from the open list.

D* makes use of various functions which operate on the vertices within the graph. The estimated cost function $g(v, w)$ gives the current estimated cost from the vertex v to vertex w. The optimal cost function $o(v, w)$ gives the optimal cost from vertex v to vertex w. The previous cost function $p(v, w)$ gives the previous value of $g(v, w)$ before v was placed in Q.

A vertex v in the open list Q is said to be a raised vertex if $g(v, v_G) > p(v, v_G)$ and a lowered vertex if $g(v, v_G) \leq p(v, v_G)$.

The key function $k(v, w)$ is used to order the open list Q and is the minimum of the previous and current costs, i.e., $k(v, w) = MIN(g(v, w), p(v, w))$. The minimum k value is given as $k_{min} = min(k(v, v_G))$ for all $v \in V$ with $t(v) = OPEN$.

The back pointer function $b(v) = w$ gives for a vertex v the vertex w that set $g(v)$ and indicates the optimal path when the algorithm is complete.

A shorthand notation will be used for all vertex functions involving the goal, whereby the goal vertex is omitted, e.g., the shorthand of a function $f(v, v_G)$ is given by $f(v)$. Furthermore the notation $f(\cdot)$ is used to refer to a function independent of its domain.

B.3 Algorithm

One important aspect to remember regarding this algorithm is that the vertices are processed from the goal vertex and in the opposite direction to the edges, or the direction of motion. The

open list is managed using a number of functions. *SIZE*() returns the number of elements in the list, *GETKMIN*() returns the k value of the vertex with the smallest k value, *INSERT*(v) inserts a vertex, *REMOVE*(v) removes a vertex, and *REMOVEMIN*() removes the vertex with the smallest k value and returns it.

The `runAlgorithm` function is used to run the algorithm from a starting vertex to a goal vertex. This is a basic example and what happens when a path is not found is left up to the application.

The algorithm is initialised through the `initialise` function. This sets the tag function value for all the vertices to *NEW* except for the given goal vertex which it inserts into the open list after initialising its g, p and b values. The g, p and b values for all the other vertices can remain undefined since they will be initialised when the vertex is first made open.

After the algorithm has been initialised, the next vertex to move to is obtained by calling the `getNext` function. This will repeatedly call the `processVertex` function until a path from the given vertex to the goal is found. If no path exists, `getNext` will return **null**. Once a path has been found and if the graph hasn't changed, further calls to `getNext` will not cause `processVertex` to be called at all.

If an edge cost changes, the `modifyCost` function must be called to update the cost function and insert the destination vertex of the edge into the open list if it is closed. If this vertex tag was *NEW*, the cost change has no effect on other vertices since it does not yet have any back pointer pointing to it. If a cost change results in a vertex which is closer to the goal than the current position of the robot, the next call to `getNext` will invoke one or more calls to `processVertex`.

The `processVertex` function contains the main part of the D* algorithm.

B.4 Examples

The examples given in Section 3.1.4 show the basic workings of D*. This section however will focus on a number of examples which show how each case within the algorithm arises.

Figure B.1 shows four examples labelled (a) to (d) which were chosen to illustrate all aspects

```
1  function runAlgorithm(Vertex v_S, Vertex v_G)
2  begin
3      initialise(v_G)
4      v = v_S
5      while v ≠ v_G do
6          w = getNext(v)
7          if w ≠ null then
8              Move to w.
9              v = w
10         else
11             /* There is no path.  Either wait for a change in the graph and try again or
                  return with a failure message.  */
12         end
13     end
14 end
```

Function `runAlgorithm`(Vertex v_S, Vertex v_G)

```
1  function initialise(Vertex v_G)
2  begin
3      /* Initialise the functions and goal vertex */
4      foreach v ∈ V do
5          t(v) = NEW
6      end
7      g(v_G) = 0
8      p(v_G) = 0
9      b(v_G) = null
10     insert(v_G)
11 end
```

Function `initialise`(Vertex v_G)

```
1  function getNext(Vertex v)
2  begin
3      while SIZE() > 0 and GETKMIN() < g(v)  do
4          processVertex()
5      end
6      if t(v) = CLOSED then
7          return b(v)
8      else
9          return null
10     end
11 end
```

Function `getNext`(Vertex v)

```
1 function insert(Vertex v)
2 begin
3     /* If the vertex is already in the open list, remove it first */
4     if t(v) = OPEN then
5         REMOVE(v)
6     end
7     INSERT(v)
8     t(v) = OPEN
9 end
```

Function `insert`(Vertex v)

```
1 function modifyCost(Vertex v, Vertex w, Real cost)
2 begin
3     c(v, w) = cost
4     if t(w) = CLOSED then
5         p(w) = g(w)
6         insert(w)
7     end
8 end
```

Function `modifyCost`(Vertex v, Vertex w, Real *cost*)

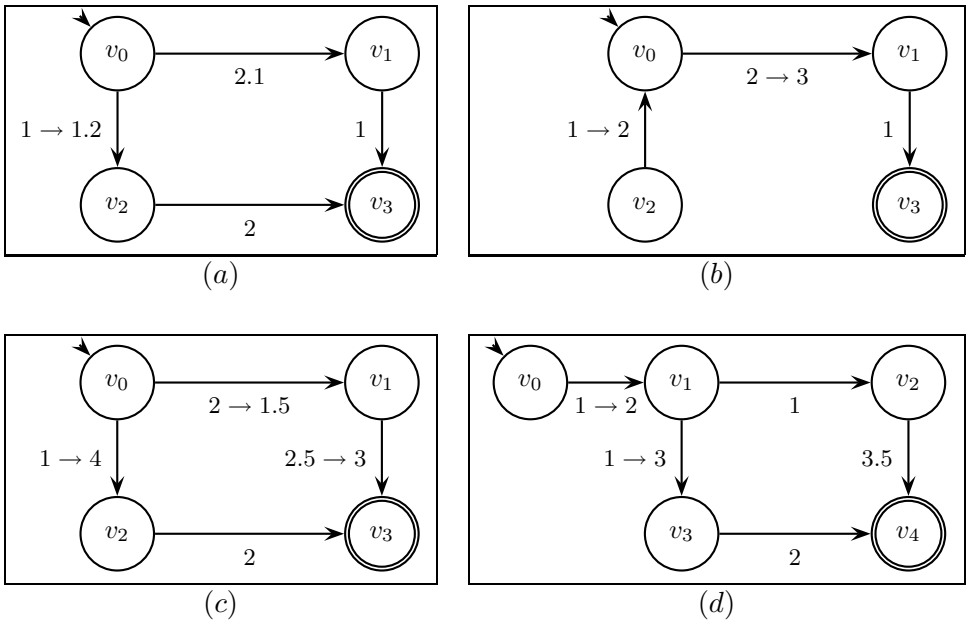

Figure B.1: Four examples chosen to illustrate all aspects of the D* algorithm

```
1  function processVertex()
2  begin
3      /* Get the next vertex to process */
4      v = REMOVEMIN()
5      if v = null then return false
6      t(v) = CLOSED
7      k_old = k(v)
8      /* Reduce g(v) by lowest cost neighbour if possible */
9      foreach w ∈ out(v) do
10         if t(w) = CLOSED and g(w) ≤ k_old and g(v) > g(w) + c(v, w) then
11             b(v) = w
12             g(v) = g(w) + c(v, w)
13         end
14     end
15     /* Process each incoming neighbour of v */
16     foreach w ∈ in(v) do
17         /* Propagate cost to NEW vertex */
18         if t(w) = NEW then
19             b(w) = v
20             g(w) = g(v) + c(w, v)
21             p(w) = g(w)
22             insert(w)
23         /* Propagate cost change along the back pointer */
24         else if b(w) = v and g(w) ≠ g(v) + c(w, v) then
25             if t(w) = OPEN then
26                 p(w) = k(w)
27             else
28                 p(w) = g(w)
29             end
30             g(w) = g(v) + c(w, v)
31             insert(w)
32         /* Reduce cost of neighbour if possible */
33         else if b(w) ≠ v and g(w) > g(v) + c(w, v) then
34             if p(v) ≥ g(v) then
35                 b(w) = v
36                 g(w) = g(v) + c(w, v)
37                 if t(w) = CLOSED then p(w) = g(w)
38                 insert(w)
39             else if t(v) = CLOSED then
40                 p(v) = g(v)
41                 insert(v)
42             end
43         end
44     end
45 ...
```

Function `processVertex` (1 of 2)

```
46 ...
47     if t(v) = CLOSED then
48         /* Set up cost reduction by neighbour if possible */
49         foreach w ∈ out(v) do
50             if b(w) ≠ v and g(v) > g(w) + c(v, w) and t(w) = CLOSED and g(w) > k_old then
51                 p(w) = g(w)
52                 insert(w)
53             end
54         end
55     end
56     return true
57 end
```

Function `processVertex` (2 of 2)

of the algorithm. Each example is a graph with vertices labelled $v_1, \ldots, v_n$ for n vertices and directed edges with their costs as labels. If an edge label contains an arrow ($\rightarrow$), the number on the left shows the initial cost of the edge while the number on the right shows what the cost of the edge becomes after some time. All edge cost changes are made at the same time. The starting point (i.e., the robot position) is indicated with a bodiless arrow pointing to the vertex. The goal vertex is indicated with a second circle around the vertex. In the investigation of these examples, the contents of the open list Q will be given as $Q = \{v_1(p_1, g_1), \ldots, v_n(p_n, g_n)\}$, where the vertex v_i is open with $p(v_i) = p_i$ and $g(v_i) = g_i$ for all $i \in [1, n]$ and the vertices are given in order such that $k(v_1) <= k(v_2) <= \cdots <= k(v_n)$.

Example (a) has four vertices. Initially, the shortest path to the goal from v_0 is through v_2, but after $c(v_0, v_2)$ changes, the shortest path is through $v1$. Initially, the open list is $Q = \{v_3(0, 0)\}$ and all other vertices are new. When `processVertex` is called, v_3 is removed from the open list and since all its neighbours are new, lines 19 through 22 are executed with w as v_1 and v_2 giving $Q = \{v_1(1, 1), v_2(2, 2)\}$. Next v_1 is processed and v_0 is opened in the same way as above and $Q = \{v_2(2, 2), v_0(3.1, 3.1)\}$. Now v_2 is processed and lines 35 through 38 are executed with $w = v_0$ leaving $Q = \{v_0(3.1, 3)\}$. Finally v_0 is processed, making no changes, and since it is closed and the open list is empty, the algorithm has been successful in finding a path. Now the cost change to $c(v_0, v_2)$ is made and since v_2 is closed, `modifyCost` opens it leaving $Q = \{v_2(2, 2)\}$. This gives $k_{min} = 2 < k(v_0) = 3$ and `processVertex` must be called to propagate the cost change. First v_2 is processed and the cost change is propagated in lines 28 through 31 and now $Q = \{v_0(3.2, 3.2)\}$. When v_0 is processed, $g(v_0) = 3.2 > g(v_1) + c(v_0, v_1) = 1 + 2.1 = 3.1$ so

lines 11 and 12 are executed redirecting the path through v_1. After this, no other changes are made and a new path is found since Q is empty and v_0 is closed.

In Example (b), there is only one path from v_0 to v_3, however, v_0 now has an incoming edge which will be changed. After `processVertex` has been run until $g(v_0) < k_{min}$, all the back pointers are the same as the edges, $p(v_0) = g(v_0) = 3$, $p(v_1) = g(v_1) = 1$ and the open list contains $Q = \{v_2(4,4)\}$. When the cost changes are made, the open list is $Q = \{v_1(1,1), v_0(3,3), v_2(4,4)\}$. When v_1 is processed, lines 26, 30 and 31 are executed with $w = v_0$ leaving the open list as $Q = \{v_0(3,4), v_2(4,4)\}$. After v_0 is processed again, $Q = \{v_2(4,6)\}$. The vertex v_2 has to be processed as well since $g(v_0) = 4$ is not smaller than k_{min}. The reason why line 26 is needed is explained in the proof of D*.

Example (c) is processed initially to find a path from v_0 through v_2 to v_3 and the open list is emptied during this process. All the vertices are closed with $p(v_i) = g(v_i)$ for all $i \in [0,3]$ and $g(v_3) = 0$, $g(v_2) = 2$, $g(v_1) = 2.5$, and $g(v_0) = 3$. After the cost changes are made, $Q = \{v_3(0,0), v_2(2,2), v_1(2.5,2.5)\}$. When v_3 is processed, the cost change for $c(v_1, v_3)$ is propagated along the back pointer to v_1 in lines 26, 30, and 31 giving $Q = \{v_2(2,2), v_1(2.5,3)\}$. Now v_2 is processed and the cost change for $c(v_0, v_2)$ is propagated along the back pointer in lines 28 through 31 so that $Q = \{v_1(2.5,3), v_0(6,6)\}$. When v_1 is processed, it is able to reduce v_0, but, it is currently a raised vertex. Therefore, in lines 40 and 41, v_1 is reopened with $Q = \{v_1(3,3), v_0(6,6)\}$. Now v_1 is processed again and its a lowered vertex so now it is able to reduce the cost of v_0 and $Q = \{v_0(6,4.5)\}$. Finally v_0 is processed without any changes.

The last portion of the algorithm which hasn't yet been seen in these examples is in lines 51 and 52 and Example (d) will show this. Once the initial path for this example has been found, the open list is empty, $p(v_i) = g(v_i)$ for all $i \in [0,4]$, $g(v_4) = 0$, $g(v_3) = 2$, $g(v_2) = 3.5$, $g(v_1) = 3$, and $g(v_0) = 4$. After the cost changes have been applied, $Q = \{v_3(2,2), v_1(3,3)\}$. Processing v_3 propagates the cost change along the back pointer and $Q = \{v_1(3,5)\}$. Now v_1 is processed and since v_2 is closed, $g(v_2) = 3.5 > k_{old} = 3$ and v_2 can reduce the estimated cost of v_1, lines 51 and 52 are executed and $Q = \{v_2(3.5,3.5)\}$. Now when v_2 is processed, it can reduce the cost of v_1 in lines 36 through 38. The cost reduction is propagated back to v_0 and the new path is found.

From these examples, one can see how every part of the algorithm is used. The exact reasons why certain things are done in a specific way are shown in the proof of the algorithm.

B.5 Proof

D* is proven to be sound, optimal and complete in the original article. This is done by defining a property for each of these attributes and proving that these properties hold throughout the algorithm. To do this, some definitions about sequences are needed:

Definition B.1. An ordering of vertices $\langle v_1, \ldots, v_n \rangle$ is defined to be a sequence if $b(v_i) = v_{i+1}$ for all $i \in [1, n)$ and $v_i \neq v_j$ for all $1 \leq i < j \leq n$.

Definition B.2. A sequence $\langle v_1, \ldots, v_n \rangle$ is defined to be monotonic if $t(v_i) = CLOSED$ and $g(v_i) < g(v_{i-1})$ or $t(v_i) = OPEN$ and $p(v_i) < g(v_{i-1})$ for all $i \in (1, n]$.

Definition B.3. Given a sequence $\langle v_1, \ldots, v_n \rangle$, if $1 \leq i < j \leq n$ then v_i is a descendant of v_j and v_j is an ancestor of v_i.

A monotonic sequence can be described as a sequence where all the members are ordered. Using these definitions, the fundamental D* properties are defined:

Property B.1. If $t(v) \neq NEW$, then a sequence $\langle v \rangle$ is constructed and is monotonic.

Property B.2. When $k_{min} \geq g(v)$ then $g(v) = o(v)$.

Property B.3. If a path from v to v_G exists and the search space consists of a finite number of vertices, then $\langle v \rangle$ will be constructed after a finite number of calls to `processVertex`. If a path does not exist then `processVertex` will return false with $t(v) = NEW$.

Property B.1 is the soundness property and states that once a vertex has been opened, a path to the goal can be found. Property B.2 is the optimality property and states that once the $g(\circ)$ value of a vertex is smaller or equal to the smallest $g(\circ)$ or $p(\circ)$ value of any open vertex, that $g(\circ)$ value is optimal. This also gives a stopping condition for the algorithm. Property B.3 is the completeness property and states that the algorithm will always get the optimal path if it exists or it will terminate if there is no path.

Properties B.1 and B.2 are in fact invariants of `processVertex` which will be proven.

These properties are proven in a series of theorems. The first theorem states that a sequence is unique and its proof is trivial.

Theorem B.1. *If each vertex has only one back pointer, the sequence $\langle v, \ldots, w \rangle$ is unique.*

Theorem B.2 gives a condition where by changing the back pointer of a vertex to one of its descendants, all the sequences containing that vertex will no longer exist. This theorem also shows that if the back pointer of a vertex is changed to any vertex which isn't one of its descendants, all sequences which existed before the change will still exist after the change. The original D* article only proved the first condition, but the theorem was used although this implied the second condition.

Theorem B.2. *Assume Property B.1 holds for a given set of vertices. Given the sequence $\langle v, \ldots, v_G \rangle$, let W be the set of vertices, such that $w \in W$ if the sequence $\langle w, \ldots, v \rangle$ exists, i.e., W is the set of all the descendants of v.*

i If $b(v)$ is set to some vertex $w_s \in W$, then no sequence $\langle w, \ldots, v_G \rangle$ exists for a vertex $w \in W$.

ii If $b(v)$ is set to some vertex $x_s \notin W$ and the sequence $\langle x_s, \ldots, v_G \rangle$ exists, then the sequence $\langle w, \ldots, v_G \rangle$ exists for every vertex $w \in W$.

Proof. The proof is trivial using Theorem B.1 and the definition of W. □

The following theorem gives restrictions on members of monotonic sequences. It shows that the sequence must contain a raised vertex if there is a vertex in the sequence with a higher $g(\circ)$ value than the first vertex in the sequence.

Theorem B.3. *Let $\langle v_1, \ldots, v_n \rangle$ be a monotonic sequence and let z be a vertex with $g(z) > g(v_1)$. Then z cannot be a member of $\langle v_1, \ldots, v_n \rangle$, unless $\langle v_1, \ldots, v_n \rangle$ contains at least one raised vertex.*

Proof. The proof follows directly from the definition of a monotonic sequence. □

Theorem B.4 is similar to Theorem B.3 and defines a restriction on the raised vertices within a monotonic sequence.

Theorem B.4. *Assume the sequence $\langle v_1, \ldots, v_n \rangle$ is monotonic. If any of the elements v_2 through v_n are raised vertices, then for at least one of these raised vertices, v_s, $k(v_s) < g(v_1)$.*

Proof. Assume that $\langle v_2, \ldots, v_n \rangle$ contains at least one raised vertex and let v_s be the raised vertex with the smallest index. Therefore, the subsequence $\langle v_1, \ldots, v_{s-1} \rangle$ contains no raised vertices. Therefore, since v_s is a raised vertex and by the definition of a monotonic sequence, $k(v_s) = p(v_s) < g(v_{s-1}) < \cdots < g(v_1)$, thus, $k(v_s) < g(v_1)$. □

Using most of what has been proven, Theorem B.5 shows under which conditions the back pointer, $b(\circ)$, of a vertex can be changed to preserve Property B.1.

Theorem B.5. *Assume that Property B.1 holds for a given set of vertices. Let v be a vertex such that $g(v) \leq k_{min}$. Let w be a neighbour vertex of v such that $c(w, v)$ is defined and $g(v) < g(w)$. If $b(w)$ is changed to v, then Property B.1 is preserved.*

Proof. Consider two cases: 1) w is not a member of the sequence $\langle v, \ldots, v_G \rangle$; and 2) w is a member.

Case 1: Since v is not a descendant of w, then all sequences will still exist after the back pointer is redirected (Theorem B.2 ii). Since $\langle v, \ldots, v_G \rangle$ is monotonic, $g(v) < g(w)$, and all sequences ending with w are monotonic (Property B.1), then all resultant sequences will be monotonic.

Case 2: Since $g(v) \leq k_{min}$, then from Theorem B.4, there can't be any raised vertices in $\langle v, \ldots, v_G \rangle$. But from Theorem B.3, if there are no raised vertices in $\langle v, \ldots, v_G \rangle$, and $g(v) < g(w)$, w cannot be a member of $\langle v, \ldots, v_G \rangle$, so case 2 cannot occur. □

Using mostly the definition of monotonic sequences, the following theorem gives restrictions under which $g(\circ)$ can be changed.

Theorem B.6. *Assume Property B.1 holds for a given set of vertices. Consider two vertices, v and w, such that $b(w) = v$. The following modifications can be made to $g(w)$ while still preserving Property B.1. If $t(w) = OPEN$, then $g(w)$ can be modified to assume any value*

provided that $g(w) > p(v)$ if $t(v) = OPEN$ and $g(w) > g(v)$ if $t(v) = CLOSED$. If $t(w) = CLOSED$, then $g(w)$ can be decreased provided that $g(w) > p(v)$ if $t(v) = OPEN$ and $g(w) > g(v)$ if $t(v) = CLOSED$.

Proof. The proof follows directly from the definition of a monotonic sequence.

□

The following corollary states how the $p(\circ)$ values for open vertices can be modified.

Corollary B.7. *Assume that Property B.1 holds for a set of vertices. Given a vertex v such that $t(v) = OPEN$, if $p(v)$ is reduced then Property B.1 is preserved.*

Proof. Since $p(v)$ has an upper bound but no lower bound, it can be reduced and Property B.1 still holds. □

The conditions under which the $t(\circ)$ function can be altered is given in the next theorem.

Theorem B.8. *Assume that Property B.1 holds for a given set of vertices. If $t(v)$ is changed from OPEN to CLOSED, Property B.1 is preserved if $g(v) \leq p(v)$. If $t(v)$ is changed from CLOSED to OPEN, Property B.1 is preserved if $p(v) \leq g(v)$.*

Proof. The proof follows directly from the definition of a monotonic sequence. □

Now that we have conditions under which the functions $b(\circ)$, $g(\circ)$, $p(\circ)$, and $t(\circ)$ can be changed, we can prove that the D* algorithm preserves Property B.1 by analyzing every point where these functions are changed in the algorithm and ensuring that the property still holds after the change.

Theorem B.9. *D* preserves Property B.1.*

Proof. The portions of `processVertex` and `modifyCost` that modify the functions $b(\circ)$, $g(\circ)$, $p(\circ)$, and $t(\circ)$ need to be examined.

The changes made within `processVertex` are summarised in the following table:

Line	Change	Reason Property B.1 is preserved
6	$t(v)$	This change could possibly invalidate Property B.1 if $p(v) \neq g(v)$, however, any problem is fixed in 30 by propagating cost changes along the back pointers to v.
11	$b(v)$	The property holds since $g(w) \leq k_{old} \leq k_{min}$, $g(w) < g(v)$ and from Theorem B.5.
12	$g(v)$	The property holds since $g(v) > g(w) + c(v, w)$ before the modification, $g(v)$ is being decreased to a value greater than $g(w)$ with $t(v) = t(w) = CLOSED$ and from Theorem B.6.
19	$b(w)$	Since $t(w) = NEW$, w is not part of any sequence so setting $b(w)$ creates a new sequence starting with w.
20	$g(w)$	The property holds since $g(w) > g(v)$ and w is only part of the sequence $\langle w, \ldots, v_G \rangle$.
21	$p(w)$	Since there are no back pointers pointing to w, $p(w)$ can assume any value.
22	$t(w)$	The property holds since there are no back pointers pointing to w.
30	$g(w)$	The property holds since $g(w)$ is set to a value greater than $g(v)$ for all w with $b(w) = v$. This also restores monotonicity possibly lost at line 6.
26	$p(w)$	The property holds since $p(w)$ is reduced (from Corollary B.7).
28, 40, 52	$p(a)$	Since $t(a) = CLOSED$, $p(a)$ can assume any value and setting it to $g(a)$ preserves monotonicity after $t(a)$ is changed to $OPEN$.
31, 38	$t(w)$	If $t(w)$ was $CLOSED$, $p(w) = g(w)$, and from Theorem B.8 the property holds.
35	$b(w)$	From Theorem B.5 the property holds since $g(v) = k_{old} \leq k_{min}$ and $g(v) < g(w)$.
36	$g(w)$	From Theorem B.6 the property holds since $g(w)$ is reduced, $g(w) > g(v)$, $t(v)$ is $CLOSED$ or $t(v)$ is $OPEN$ (lines 40 and 41 must have been executed for a previous neighbour) and $g(w) > p(v) = g(v)$.
41, 52	$t(a)$	From Theorem B.8 the property holds since $p(a) = g(a)$.

The `processVertex` function given in the original D* article contained a logical error in line 28 of the `processVertex` function. The original article had $p(w)$ assigned to $g(w)$ after $g(w)$ was updated when w was closed. This would invalidate Property B.1 if $g(w)$ was increased high enough. By setting $p(w)$ to $g(w)$ when w is closed makes much more sense when looking at the definition of a monotonic sequence.

The function `modifyCost` affects only $t(\circ)$ and $p(\circ)$. On line 5, $p(w)$ is initialised to $g(w)$ which has no affect on the monotonicity. On line 6, $t(w)$ is set to $OPEN$. Since $p(w) = g(w)$, from Theorem B.8, Property B.1 is preserved. □

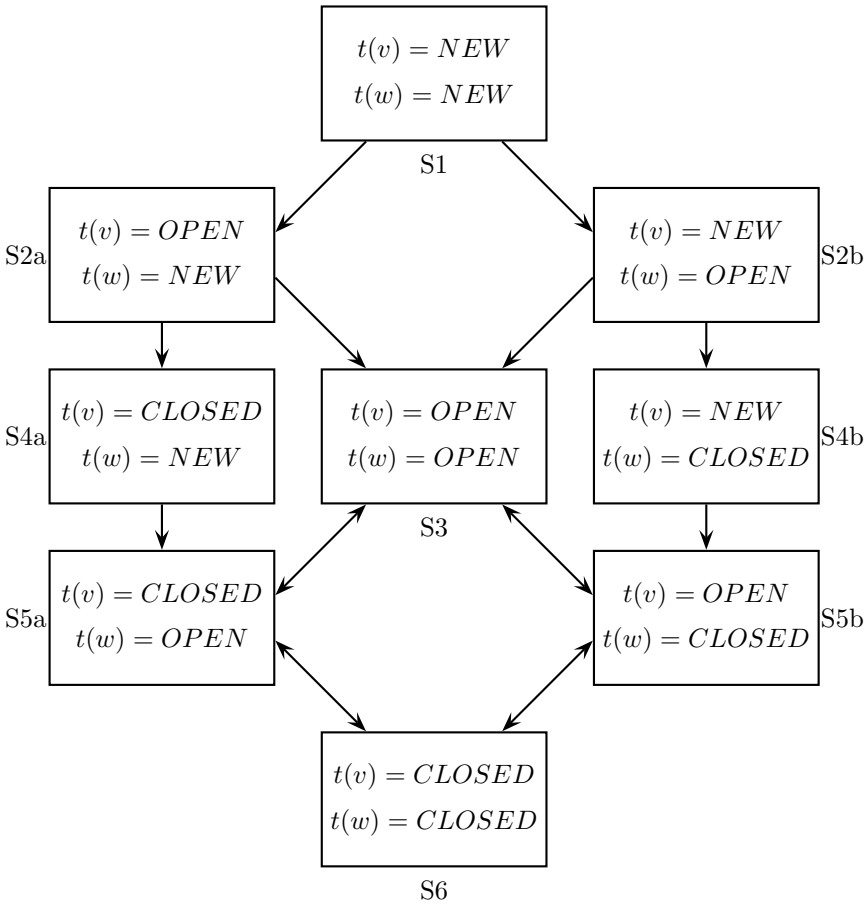

Figure B.2: Vertex state transition diagram

Figure B.2 gives the state transition diagram for the tag states of two adjacent vertices and all possible transitions between them. Using this figure, Theorem B.10 gives a relationship between the $k(\circ)$ value of an open vertex and the $g(\circ)$ value of a closed neighbouring vertex. In the proof, if reference is made to a state without implicitly specifying the a or b counterpart,

this should be read as the a state, however, the statement is similar for the b state.

Theorem B.10. *Given any two vertices, v and w, such that $c(w, v)$ is defined with $t(v) = CLOSED$ and $t(w) = OPEN$, then $k(w) \leq g(v) + c(w, v)$.*

Proof. Initially v and w are *NEW* vertices as shown in S1. Since they can be opened by different vertices, no assumptions are made about their $g(\circ)$ and $k(\circ)$ values in the transition through S2 to S3. The transition from S2 through S4 to S5 occurs in lines 19 through 22 where v is closed and w is opened. In this segment $k(w) = g(w)$ is set to $g(v) + c(w, v)$, thus the theorem holds.

In the transition from S3 to S5, v is closed while w remains open. Consider three cases: 1) $b(w) = v$; 2) $b(w) \neq v$ and v was a lowered vertex; and 3) $b(w) \neq v$ and v was a raised vertex.
Case 1: At lines 26 through 31, $g(w)$ is set to $g(v)+c(w, v)$ and $k(w) \leq g(w)$; thus, the theorem holds.
Case 2: If $g(w) \leq g(v) + c(w, v)$, then since $k(w) \leq g(w)$ no action is taken and the theorem holds. If $g(w) > g(v) + c(w, v)$, then at lines 35 through 37, $g(w)$ is set to $g(v) + c(w, v)$. Since $k(w) \leq g(w)$, the theorem holds.
Case 3: If $g(w) > g(v) + c(w, v)$, then v is placed on the open list at lines 40 and 41, and v and w are transitioned back to state S3.
In state S5, it is possible for a vertex other than v to modify $g(w)$. Since reducing $g(w)$ can only reduce $k(w)$, only modifications that increase $g(w)$ are of concern (lines 26 and 30). In this segment, $p(w)$ is reassigned to prevent $k(w)$ from increasing, and the theorem holds.

In the transition from S5a to S6, consider three cases: 1) an immediate transition is made within `processVertex` from S6 back to S5a; 2) an immediate transition is made from S6 to S5b; and 3) the state remains in S6.
Case 1: At lines 40 and 41, w is placed back on the open list with $k(w) = g(w)$ after a possible reduction of $g(w)$ at line 12. Since $k(w)$ can only be reduced, $k(w) \leq g(v) + c(w, v)$ and the theorem holds.
Case 2: w is closed and v is opened. At lines 28 through 31 and 35 through 38, $k(v)$ is set to $g(w) + c(v, w)$ and the theorem holds. At lines 51 and 52, $k(v)$ is set to $g(v)$. Since $g(w) > g(v) + c(w, v)$, then $k(v) = g(v) < g(w) - c(w, v) < g(w) + c(v, w)$ and the theorem

holds.

Case 3: The only segment of `processVertex` that leaves both vertices closed are lines 11 and 12. If $g(w)$ exceeds $g(v)$ by more than $c(w,v)$, then $g(w)$ is set to $g(v)+c(w,v)$; thus, $g(w) \leq g(v)+c(w,v)$. Note also that $g(v) \leq g(w)+c(v,w)$, otherwise, v or w will be placed on the open list at lines 18 through 54, and the state will transition out of S6. If the transition is made from S6 to S5a, $k(w) = g(w) \leq g(v)+c(w,v)$; and if it is made to $S5b$, $k(v) = g(v) \leq g(w)+c(v,w)$ and the theorem holds for both cases.

The last case analyzes the effects of the `modifyCost` function. This function is able to change a vertex from closed to open. Since state S4 is only a temporary transition within `processVertex`, it cannot be affected by `modifyCost`. The `modifyCost` function can affect transitions from S5 to S3, but the theorem states nothing about these transitions. The only applicable transitions are from S6 to S5a and S5b. Since $g(w) \leq g(v)+c(w,v)$ for states in S6 (see above), if w is placed on the open list, then $k(w) \leq g(v)+c(w,v)$ and the theorem holds. This occurs at lines 5 and 6 of `modifyCost`. The same reasoning applies to the transition from S6 to S5b. □

Corollary B.11 was actually proven within the proof of Theorem B.10 and gives a relationship between two neighbouring closed vertices.

Corollary B.11. *Given two vertices v and w such that $t(v) = t(w) = CLOSED$, if $c(v,w)$ is defined, then $g(v) \leq g(w)+c(v,w)$, and if $c(w,v)$ is defined, then $g(w) \leq g(v)+c(w,v)$.*

Proof. See the proof for Theorem B.10. □

The next theorem proves an important property of the D* algorithm, which is that the k_{min} values are monotonic between calls to `modifyCost`.

Theorem B.12. *Between calls to* `modifyCost`*, the parameter k_{min} increases or remains at the same value for a finite number of calls to* `processVertex`*.*

Proof. Let v be the next vertex to be removed from the open list; therefore, $k(v) = k_{min}$. It will be shown that v can only be re-inserted again with a higher k value, or it could reposition vertices on the open list to have $k(\circ)$ values greater than $k(v)$. Since there are a finite number of vertices on the open list with $k(\circ)$ equal to $k(v)$, then k_{min} must increase or remain the same

for a finite number of iterations. At lines 11 and 12 in `processVertex`, $g(v)$ can be reduced. Since w is closed, the value of $g(v)$ cannot be reduced below $k(v)$ (Theorem B.10). At lines 19 through 22, w is inserted onto the open list with $k(w) = g(w) = g(v) + c(w, v) > g(v) \geq k(v)$, so the theorem holds. At lines 26 through 31, w is inserted or repositioned on the open list. If it is repositioned, changing the value of $g(w)$ can only reduce $k(w)$ or leave it unmodified. If $k(w)$ is reduced, then $k(w) = g(w) = g(v) + c(w, v) > g(v) \geq k(v)$ and the theorem holds. If $k(w)$ is unmodified, then it was either greater or equal to $k(v)$, and the theorem holds. If w is inserted onto the open list, then for the same reasons as lines 19 through 22, the theorem holds.

At lines 35 through 38, w is inserted, repositioned, or left in the same place on the open list. Since $p(v) \geq g(v)$, then $g(v) = k(v)$. Since after the modification $g(w) > g(v)$, if w is inserted, then $k(w) = g(w) > g(v) = k(v)$, and the theorem holds. If w is already, on the open list, then either $p(w) < g(w)$ and w is not repositioned, or $p(w) \geq g(w)$, and $k(w) = g(w)$; thus, the theorem holds for the above reasons. At lines 40 and 41, v is re-inserted onto the open list. Since $p(v) < g(v)$ before re-insertion and $p(v) = g(v)$ after re-insertion, then $k(v)$ must increase and the theorem holds. At lines 51 and 52, w is inserted onto the open list. Since $k(w) = g(w) > k_{old} = k(w)$, the theorem holds. □

The theorem below states that if the vertices with $g(\circ)$ values less than equal to k_{min} at an invocation of `processVertex` are optimal, then the vertices with $g(\circ)$ values less than or equal to the new k_{min} at the next invocation of `processVertex` are optimal.

Theorem B.13. *If $g(v) = o(v)$ for all $v \in V$ such that $g(v) \leq k_{old}$, then $g(w) = o(w)$ for all $w \in V$ such that $k_{old} \leq g(w) \leq k_{min}$.*

Proof. A vertex w_i, such that $k_{old} \leq g(w_i) \leq k_{min}$ must be either 1) closed or 2) open with $g(w_i) = k_{min}$. Order the vertices w_i by their $g(\circ)$ values such that $k_{old} \leq g(w_1) \leq g(w_2) \leq \ldots \leq g(w_n) \leq k_{min}$.

Consider w_1 first.

Case 1: From the premise, $g(z) = o(z)$ for all neighbour vertices z such that $g(z) < g(w_1)$. From Corollary B.11, none of these optimal neighbour vertices can reduce $g(w_1)$ nor can any

neighbour vertices on the open list reduce $g(w_1)$, since the $g(\circ)$ values for vertices on the open list are greater than or equal to k_{min}. These open vertices cannot reduce each other's $g(\circ)$ values below k_{min}, nor can their closed neighbours (Theorem B.10). Thus since $g(w_1) \leq k_{min}$, we have that $g(w_1) = o(w_1)$.
Case 2: Since $g(w_1) = k(w_1)$, we have from Theorem B.10 that a neighbour vertex v with $g(v) = o(v) < g(w_1)$ cannot reduce $g(w_1)$. Thus, $g(w_1) = o(w_1)$.

The above argument can be repeated for the remaining vertices in order, and by induction, $g(w_i) = o(w_i)$ for all $i \in [2, n]$. □

Using the monotonicity of k_{min} and the inductive step proven above, the optimality of D* is proven below irrespective of the calling sequence of `modifyCost` and `processVertex`.

Theorem B.14. *D* preserves Property B.2.*

Proof. Initially, the goal vertex v_G is placed on the open list with $g(v_G) = o(v_G) = 0$. There are no closed vertices since $t(v) = NEW$ for all $v \in V/\{v_G\}$. When `processVertex` removes and expands a vertex from the open list, k_{old} is set to k_{min} and k_{min} is increased monotonically (Theorem B.12). From Theorem B.13, for the vertices v such that $k_{old} \leq g(v) \leq k_{min}$, $g(v) = o(v)$; therefore $g(v) = o(v)$ if $g(v) \leq k_{min}$.

Assume that $c(w, v)$ is modified using `modifyCost`. If v is open, the change to $c(w, v)$ cannot affect optimality of vertices with $g(\circ) \leq k_{min}$, since $g(v)$ can be reduced no lower than k_{min} (Theorem B.10). Since $g(v) + c(w, v)$ must be greater than k_{min}, then $g(\circ)$ values equal to or lower than k_{min} cannot be reduced; thus the premise to Theorem B.13 is still true and `processVertex` can be called to propagate the modification. If v is closed, then `modifyCost` ensures that $k_{min} \leq g(v)$ by placing v on the open list. This operation preserves the truth of the premise to Theorem B.13, since reducing k_{min} selects a subset of the optimal closed vertices. Thus Property B.2 is preserved regardless of the pattern of cost modification and propagation. □

The final theorem below shows that D* will always find a sequence of vertices between any vertex reachable from the goal if it exists; otherwise, it will terminate and indicate that there is no such sequence.

Theorem B.15. *D* preserves Property B.3.*

Proof. Assume that a vertex v is reachable from the goal, i.e., $\langle v_1, \ldots, v_n \rangle$ exists, where $v_1 = v$ and $v_n = v_G$. Initially, v_G is placed on the open list. When vertex v_i is expanded, it places or repositions v_{i-1} on the open list and redirects its back pointer if necessary. By induction, the sequence $\langle v_1, \ldots, v_n \rangle$ will be constructed unless some vertex v_s exists in the sequence $\langle v_1, \ldots, v_n \rangle$ such that $t(v_s) = OPEN$, and v_s is never selected for expansion. Once v_s is on the open list, $k(v_s)$ never increases. Thus, due to the monotonicity of k_{min} (Theorem B.12), v_s will eventually be removed from the open list.

If a vertex v_1 is unreachable from the goal, eventually all reachable vertices will be placed on the open list and will be removed given the above reasoning. Since the number of search vertices is finite, the list will empty after a finite number of calls to `processVertex`, and null will be returned with $t(v_1) = NEW$. □