ABSTRACT

Robot Path Planning with a Moving Goal

Daniel Drake

Chairperson: Scott Koziol, Ph.D.

Path planners in which a hunter is required to chase after a moving target is an important problem for modern robotic systems such as Unmanned Aerial Vehicles (UAVs) and Unmanned Underwater Vehicles (UUVs). This thesis describes an incremental moving target path planning algorithm which leverages previous planning data to update the path in the case where the target moves. The algorithm in this thesis addresses the need for a quick path-planner that can be used in an environment where the target is moving. The algorithm does this by sacrificing optimality in order to reduce the complexity of the problem. The algorithm will be shown to reduce the complexity of re-planning by approximately 12 times while only increasing path length taken by 1.5%. Within this thesis analytical estimates of the best and worst case complexity of the algorithm were developed, and these estimates were validated with experimental data.

Robot Path Planning with a Moving Goal

by

Daniel Drake, B.S.E.C.E

A Thesis

Approved by the Department of Electrical and Computer Engineering

_____

Kwang Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Electrical and Computer Engineering

Approved by the Thesis Committee

_____

Scott Koziol, Ph.D., Chairperson

_____

Michael Thompson, Ph.D.

_____

Lesley Wright, Ph.D.

_____

Eugene Chabot, Ph.D.

Accepted by the Graduate School
August 2017

_____

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

Introduction

Moving target path planners are used in situations where a hunter has to reach a goal which moves. Applications of this within robotics includes Unmanned Aerial Vehicles (UAVs) and Unmanned Underwater Vehicles (UUVs) where a robot is required to chase another robot or moving object such as a submarine [2], or robotic arms where an object required to be grabbed is moved along a conveyer. Path planners can also be used in networking where one has to send data to a certain location, with a moving target algorithm being used when that location is uncertain or changing [3]. If a path planner which was not specialized for cases with a moving target was used in these situations, it is possible that by the time a new path was calculated, it would be made irrelevant by the distance the target moved during calculation.

The way the robot's environment is mapped into a path planner's data structure can vary based on the application. Popular choices for representing an environment include:

- Potential fields: A field is placed across the robot's map with obstacles being represented by values of repulsion and the target being represented by an attraction value allowing for a steepest decent to be used to maneuver a robot through the environment. This also creates a smooth path for a robot to follow making it popular for manipulators [4].
- Cell Decomposition, Fig. 1.1 : A graph is made with connected nodes representing a set amount of physical space. Blocked nodes are included to represent the obstacles and open nodes to represent space that can be navigated. This allows scaling of complexity with the resolution of the nodes as smaller

nodes would lead to a more exact but more complex environment. Two popular ways to connect the nodes are an 8 connected and 4 connected grid.

- Visibility Graphs , Fig. 1.2: A system where a set of nodes and paths are created by connecting the vertices of the obstacles with one another by drawing a line straight between them and seeing if an obstacle blocks that path. The implementation of this is simple and can be proven to give an optimal path unless a safety factor is introduced for the robot [4].
- Voronoi diagrams, Fig. 1.3: This method maximizes the distance of the robot to obstacles by using all points that are equidistant between two obstacles as possible areas the robot can go. This maximizes the safety of the robot though can lead to a relatively long path.

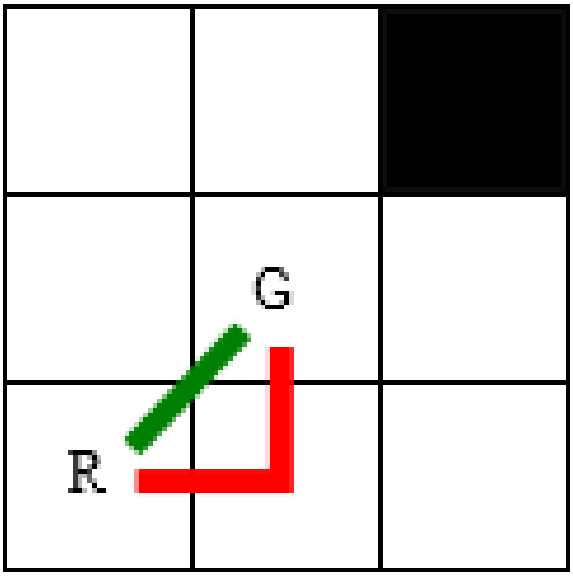

Figure 1.1: Example of a Cell Decomposition Environment. With the black square representing an obstacle, the green line indicating the shortest path in an 8 connected grid, and the red line indicating a shortest path within a 4 connected or Manhattan grid.

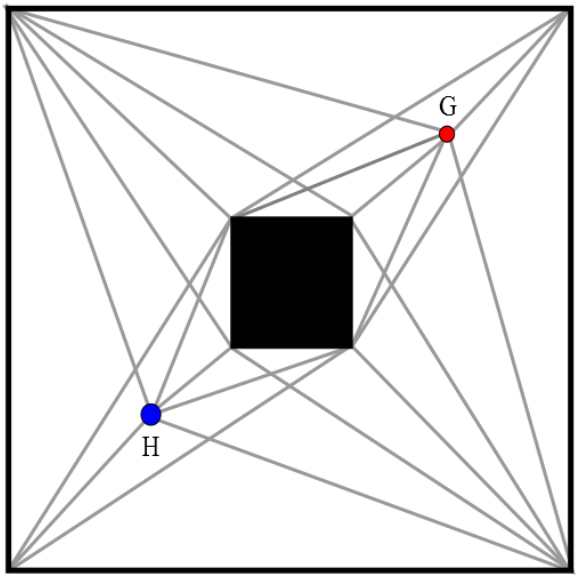

Figure 1.2: Example of a Visibility Graph. The black box represents an obstacle, the blue circle represents the hunter, the red circle represents the goal, and the paths able to be taken are shown in gray.

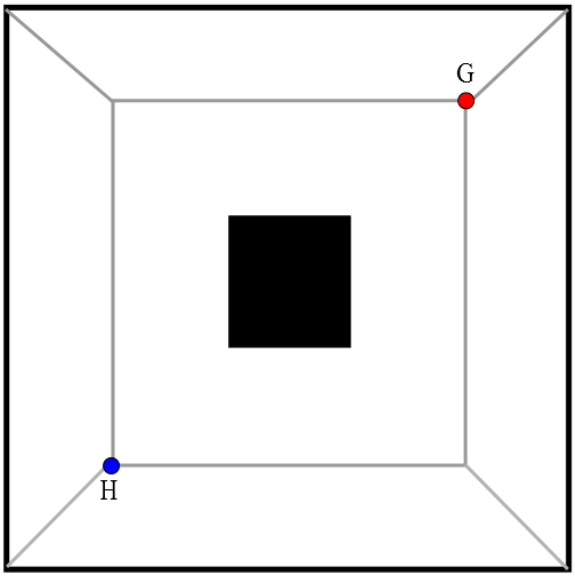

Figure 1.3: Example of a Voronoi Graph. The black box represents an obstacle, the blue circle represents the hunter, the red circle represents the goal, and the paths able to be taken are shown in gray. The paths are equidistant between the obstacle and the wall.

Moving target problems are often represented by cell decomposition by using a connected graph of nodes, ([1] [5] [6]) and in this paper the problem will be in a four-connected graph , or Manhattan graph, where the hunter or target can move from any unblocked node to any unblocked neighbor with a cost of one. Blocked nodes cannot be moved to or from and represent obstacles in the environment. Edges of the graph are treated as blocked nodes to keep a static sized graph and prevent the hunter or the target from leaving the grid. Given the nature of a chase, the target will be set to move at the same speed as the hunter, but skip a turn to move every so often. This is done to allow the hunter to catch up with the target in the case of the target moving directly away. It is also assumed that all nodes can be checked for a straight line distance, which ignores obstacles, to another node for purposes of a heuristic search.

The popular metrics used for evaluating path planners include: completeness, optimality, and complexity [7]. Completeness is the idea that if a path exists from the start to the goal, the algorithm will find it, and if a path does not exist the planner will tell you. This is important because if a path planner is not complete it is unable to be used in certain circumstances. Optimality in the context of this thesis relates to the planner's ability to find the shortest path. This metric can expanded for non-optimal algorithms to see how sub-optimal they are. To do this the length of the path generated can be compared to a path generated by an algorithm known to be optimal. Space Complexity is a measure of the amount of memory required for the algorithm to plan a path. Time Complexity is a measure of how long the algorithm will take to find a solution. This measurement can vary based upon the algorithm and environment. Within environments and algorithms based in cell decomposition a common metric is to see how many cells have been calculated. Within A* based algorithms the calculation of a node involves searching neighboring nodes for a path.

This is the most repeated process in algorithms such as A* [8] a popular path planner algorithm.

Many path planning algorithms within a cell decomposition environment are based upon A* [8]. A* creates a path by pointing connected nodes to one another creating a chain from the start to goal. To connect two nodes A* checks two things, if a path is possible between the two nodes, and if that path is shorter than any other path already calculated involving those nodes. A* uses a heuristic and list of *closed* and *open* nodes to determine which nodes need to be checked as to determine a path. This heuristic is important as it prevents simply all nodes from being looked at in a wave like fashion. The heuristic used is an admissible one which means that it underestimates or correctly estimates the distance between two nodes [9]. The admissibility of a heuristic is important as an admissible heuristic can maintain optimality of a path.

In order to approach the problem of a moving target, incremental algorithms were used that re-plan an already known path based on information gathered during the initial path planning. One example of this is D* [10] which is for use in a partially known environment, or where obstacles will move. It does this in a cell decomposition map by blocking the node in which the obstacle moved into or where a new obstacle was found. It then checks if the path went through that node and if so checks the search tree connected to that node for changes that need to be made. This significantly decreased the number of nodes required to be recalculated as compared to re-evaluating the map every time a new obstacle was found. In this case a search tree refers to a set of nodes that point to one another to create paths that the robot may travel.

Other search trees within this thesis were looked at to see what was affected when a target moved. It was found that when the goal moved the search trees generated by the nodes immediately surrounding goal and where these trees edges touched

one another could change drastically based upon how and where the goal moved. From this an algorithm was created to track and update these nodes by creating a list of *leaf* nodes to be recalculated upon goal move.

The algorithm in this thesis addresses the need for a quick path-planner that can be used in an environment where the target is moving. The algorithm does this by sacrificing optimality in order to reduce the complexity of the problem. The algorithm will be shown to reduce the complexity of re-planning by approximately 12 times while only increasing path length taken by 1.5%.

Chapter 2 will go over a more in-depth background on path planning. This will be followed by a description of the algorithm and how it acts in best and worst case scenarios. 994 lines of Matlab code were written to implement the algorithm and test it with experiments, Appendix A . The experiments done to analyze the algorithm will then be detailed and the results of these experiments will be separated into two sections, optimality and complexity. Finally, the meaning and impact of the results will be discussed.

CHAPTER TWO

Related Work

Djikstra [11] established a path planning algorithm for a grid of connected nodes in which there exists a path between any two nodes, and a method to find the shortest path between any of these nodes. Djikstra's algorithm was then expanded upon to create a series of best fit algorithms [12] with a notable one being A* [8]. Path planning algorithms can address a variety of problems such as routing, networking, and speech recognition [9].

The algorithm described in this thesis is based on A*. The environment used in A* is based on cell decomposition and denotes each node as blocked if it is an obstacle, or unblocked if the robot is able to move through it, Fig. 1.1. A* connects nodes to create a path by placing a pointer in each node which points to another node it is connected to. A* aims to create the shortest chain of pointers between the start to the goal possible. In order to do this, two sets of nodes are created and a few parameters are assigned to them. The two sets nodes are assigned to are the *open* and *closed* sets. The *open* set contains all the nodes that are being considered for expansion, while the *closed* set contains all the nodes that have been expanded and are not currently considered for expansion. The parameters assigned to the nodes are: the cost of the robot moving to the current node, g, the cost of moving from one node to another, c (which can vary from node to node), and a heuristic provided by the user which estimates the cost of moving from the current node to the goal, h. The parameter h is typically the straight line distance from a node to the goal. For a simple environment a cost of $c = 1$ is used to move through any two connected open nodes. To expand a node A* takes a singular node and looks at all the nodes connected, so in a four connected grid it looks at the four surrounding nodes assuming they are all open. It

then checks the cost of moving from the expanded node c to its connecting nodes and adds this to the cost of the expanded node g. If this combined cost $g + c$ is less than the connected node's cost to the goal g' or if if the connected node does not have a cost assigned to it, the connected node now points to the expanded node and is put into the *open* set. Once all the connecting nodes have been checked, the expanded node is set to the *closed* set and a new node is expanded until the goal is reached. In order to determine the order of nodes expanded, A* first expands the start node creating an initial set of *open* nodes. The *open* node with the smallest distance to goal h is then expanded and this process is repeated until the goal is reached or the *open* set is depleted. Expanding until the *open* set is depleted or the goal is found insures that if there is a path from the start to goal it will be found, making this a complete algorithm.

This heuristic used in A* can be shown to be admissible in most cases and thus find an optimal solution [9]. The heuristic can been said to be admissible is it always perfectly estimates or under estimates the distance from the node to the goal. This is why straight line distance is typically used as it assumes the shortest distance possible from any given node. The reduction of complexity achieved by using a heuristic is important within path planners is important as it allows for more complex environments to be analyzed and opens up real-time path planners [13].

Challenges that A* based path planners have encountered include dynamic environments, any-angle movement, trading complexity for sub-optimality, and moving targets [14]. The algorithm in this thesis modifies the dynamic environment path planner, dynamic A*, known as D* [10], in order to address the complexity of the moving-target problem by trading optimality for a reduction in time complexity. D* is a complete and optimal A* variant with provisions in place to allow for quick re-planning in case of a change in the environment, but not the target. It accomplishes this by first expanding from the goal instead of from the start as was done in A* and

then leveraging non-optimal paths created in the initial expansion and searching from these to find a new path. D* itself has been modified several times to attack different problems such as Moving target D* lite [1], or reduce complexity with The Focused D* [15] which achieves this by incrementally updating the map, and D* lite [16] which combines both incremental and heuristic elements.

Moving target path-planners have been used in robotics [17], video games [18] [19] and networking [20]. These applications all rely on being able to quickly re-plan a path when the objective changes. Two major moving target path planners are Hierarchical Path-Finding Formula, or HPA* [6], which simplifies the environment to reduce the complexity of re-planning the initial path while sacrificing optimality, and Moving target D* lite [1], which leverages data generated from using D* to reduce the nodes recalculated during a re-plan while maintaining optimality.

The algorithm described in this thesis modifies D* [10] in a novel fashion by introducing the tracking of a set of search trees. This allows the reduction of complexity by estimating which nodes will need to be recalculated if the target moves. D* makes several changes to A* in order to allow for incremental planning in the case of an unknown environment while maintaining optimality. D* acts similar to A* in that it uses a *open* and *closed* set along with parameters assigned to the nodes, including: the cost of the robot moving to the current node g, the cost of moving from one node to another c, which can vary from node to node, and a heuristic denoted by h. In order to find the initial path D* does a very similar thing to A*. First it expands nodes around the goal, this is different than A*, as A* starts its expansion around the robot or start. The heuristic used is also changed, instead of straight line distance to the goal, h is changed to be the straight line distance to the start and the nodes within the *open* set with the lowest h are expanded first. Doing this creates a different set of nodes that are expanded but still leads to an optimal path being taken, and still provides an algorithm that is complete. D* then uses this new data

to re-plan the path in the event of a new obstacle being found or obstacles moving. It does this by first taking the node which was previously unblocked and setting it to a blocked state, representing the finding of an obstacle. That node is then analyzed further by looking at any nodes it points to and setting those nodes to a *raised* state. This node then checks if it can lower or maintain it's g by having it be pointed to by one of its neighbors. If this is not the case, the nodes g is set to a large number and all the neighboring nodes are set to the *raised* state. This process is repeated with the newly *raised* nodes until a node is able to lower or maintain its g. Once a node is able to maintain or lower its g, that node is set to a *lowered* state. The *lowered* node is then expanded allowing a new path to propagate through all the *raised* nodes while simultaneously setting them to the *lowered* state. This is done until all the *raised* nodes are set to the *lowered* state.

This method of *raised* and *lowered* states introduces the idea of search trees to this algorithm. The search tree starts at the newly blocked node and expands out to all the *raised* nodes. The search tree is then shifted so that its start begins at the first *lowered* node and expands from there. The algorithm described in this thesis looks at the search trees which start immediately around the goal in D* and instead of recalculating them entirely when the goal moves, creates a list of nodes that will be changed if the start of these trees is moved and instead expands those.

CHAPTER THREE

Design and Implementation

In this chapter the moving target path planner's algorithm will be detailed, estimates will be made on its performance, and the experiments run to analyze it will be explained.

Algorithm

The environment used for this algorithm is similar to the environment used in Dijkstra based path finding algorithms. A grid of nodes are created with a set of parameters: actual cost of movement from the goal, straight path distance to the hunter's location, cost to move through the node, and a tag which identifies which search tree it is a part of. A set of nodes cost's to move through them are set set to an extremely large value in order to represent an obstacle. A node is then set to represent the goal and a different node is set to represent the starting point.

The algorithm consists of four steps in which different nodes are expanded. Expansion of a node consists of taking a node and checking the nodes surrounding it. Expansion is important as it allows the algorithm to find obstacles, determine if one path is shorter than another and establish search trees for determining whether a node will be required to be recalculated later. The four steps taken are summarized as follows:

(1) Initial expansion around the goal: This establishes the initial search trees and begins the search for the first path.

(2) Expansion for finding the initial path: This expands upon the initial search trees and is run until a path is found, while marking nodes that will need to be expanded if the goal moves.

(3) Expansion around new goal when the goal moves: This step is run when the goal moves and creates a new set of search trees and connects them to the old search trees.

(4) Path Correction: This steps re-expands the nodes marked for expansion by steps 1 and 2.

Now a more detailed explanation of the listed steps will be given.

Step 1 (Initial Expansion around the goal): This algorithm begins its initial search by first taking the goal node and expanding around it. The cost of each of the nodes initially expanded to is set to the cost of moving from that node back to the goal. They are given a unique tag to show they are the beginning of a search tree along with assigning each search tree a unique number. All nodes expanded to are set to the open set.

Expanding (Algorithm 1): To expand a node, a new node around the expanded node is first checked to see if it is an obstacle, if not, the current cost of the new node is compared to the cost of the expanded node, plus the cost of moving through the new node. If the new cost is less, then the following changes are applied to the new node: the new node is set to point to the expanded node, the cost is updated to the new lower cost, the tag which identifies the search tree of the expanded node is copied to the new node, and the new node is set to the open set. If the cost is not changed and the two nodes being compared are from different search trees, a tag is placed upon both of them so that where the search trees touch is known. Once all the surrounding nodes have been checked, the expanded node is set to the closed set. This is shown in Algorithm 1, and the and the equivalent Matlab code written is found in Appendix B.

Step 2 (Expansion for Finding the initial path): The nodes in the open set are then expanded. The order of which the open set is expanded can be determined by a heuristic. For this algorithm, whichever node had the shortest straight line path to

Algorithm 1 Expansion

```
1: procedure ExpandNode
2:     for All nodes Surrounding Current Node do
3:         if surrounding node is not obstacle & cost of surrounding node > cost of
   node + movement then
4:             Surrounding node cost = cost of node + movement
5:             Surrounding node tree = expanded node's tree
6:             Surrounding node pointer = expanded node
7:             Surrounding node => open set
8:             if Surrounding Node is part of leaf set then
9:                 Surrounding node removed from leaf set
10:            end if
11:            Surrounding node priority = straight line distance to Hunter
12:        end if
13:        if Surrounding node is not obstacle & cost of surrounding node < cost of
   node + movement & surrounding node tree != expanded node tree then
14:            Surrounding node => Leaf set
15:            Expanded node => Leaf set
16:        end if
17:    end for
18:    Expanded Node => Closed Set
19:    Expanded Node removed from Open Set
20: end procedure
```

the hunter node was the first to be expanded. The open set is expanded until the start node is reached at which point the initial expansion is stopped, all the nodes in the open set assigned to the closed set, and the hunter and goal are allowed to move. Steps 1 and 2 are shown in Algorithm 2, and the equivalent Matlab code written is found in Appendix B.

Algorithm 2 Initial Path Finding

```
procedure INITIAL PATH FINDING
    Search_Tree = 1
    for All nodes Surrounding Goal do
        if surrounding node is not obstacle then
            Node tree = Search_Tree
            Node Cost = 1
            Node => Parent of Search_Tree
            Node => Open Set
            Node priority = Straight line distance to Hunter
            Search_Tree = Search_Tree +1
        end if
    end for
    while start/hunter is not in open set do
        for Nodes in Open Set do
            function EXPANDNODE(Node with lowest priority)
            end function
        end for
    end while
    Open Set => Closed Set
    Open Set is Cleared
end procedure
```

Step 3 (Expansion around new goal when the goal moves): When the goal moves another function is called to determine if a new path is needed before the hunter moves again. This algorithm first searches around the old goal location to find the new goal location. Once the new goal location is found, the goal is expanded around to create new search trees. The new search trees are expanded as if it were an empty map, until all of the old search trees ,"parent blocks", have been expanded to. No heuristic is used , in order to allow each search tree to expand the same amount. The costs of

all the nodes in the old search trees are then modified by adding the cost of moving from the old search tree to the new search tree. This cost can be negative if the goal moved onto the parent block of an old search tree.

Step 4 (Path Correction): All of the nodes which are marked as places where the search trees touch are then set to the open set. The open set is then expanded, starting from the lowest cost to goal node and then moving to higher cost nodes. This expansion continues to add new nodes into the open set and marks the new position where the search trees meet until the current node of the hunter is marked. Once this is done all the nodes still in the open set are set to closed and the actors are allowed to move again. This process is repeated until the hunter reaches the goal. This process, combined with step three is what is run when the goal moves as seen in Algorithm 3, and the equivalent Matlab code written is found in Appendix C.

Complexity Estimates

In order to estimate the complexity of the algorithm, worst case and best case scenarios were formed. The most basic search is on a map with no obstacles and no heuristic being used in the initial search. Fig. 3.1 shows what the search trees would look like in this scenario if the goal was in the center and the hunter was in one of the corners. All the search trees are equivalent, and touch one another along their edges. Once the goal moves, the edges that touch and the immediate blocks around the goal must be recalculated as shown in pink in Fig. 3.2. As the nodes one step away from the goal are always recalculated when the goal moves, the complexity of this scenario will scale with the length of the diagonals. The number of nodes within the diagonals is estimated using the length of a diagonal of a square (3.1). With two diagonals being present, we multiply the length by two, leading to a scaling factor shown in (3.2) for a worst case scenario.

Algorithm 3 Path correction

```
1: procedure PATH CORRECTION
2:     New goal position is found
3:     Parent of Search_Tree => Old Parent of Search_Tree
4:     Parents of Search_Trees cost = not calculated/high
5:     Search_Tree = 1
6:     for All nodes Surrounding Goal do
7:         if Surrounding node is not obstacle then
8:             Node tree = Search_Tree
9:             Node Cost = 1
10:            Node => Parent of Search_Tree
11:            Node => Open Set
12:            Node priority = Straight line distance to Hunter
13:            Search_Tree = Search_Tree +1
14:        end if
15:    end for
16:    while Old Parents of Search_Trees are not in Open Set do
17:        for Nodes in Open Set do
18:            function ExpandNode(Open Set)
19:            end function
20:        end for
21:    end while
22:    for Number of old Search trees do
23:        for All nodes in Search_tree do
24:            Node Cost = Node Cost + Cost of Old Parent
25:        end for
26:    end for
27:    Open Set => Closed Set
28:    Open Set is Cleared
29:    Leaf Set => Open Set
30:    Leaf Set Is Cleared
31:    while start/hunter is not in open set do
32:        for Nodes in Open Set do
33:            function ExpandNode(Open Set)
34:            end function
35:        end for
36:    end while
37: end procedure
```

$$Length\ of\ Diagonal = \sqrt{(2 * pathlength^2)} \tag{3.1}$$

$$Worst\ Case\ Complexity = \sqrt{(2 * pathlength^2)} * 2 \tag{3.2}$$

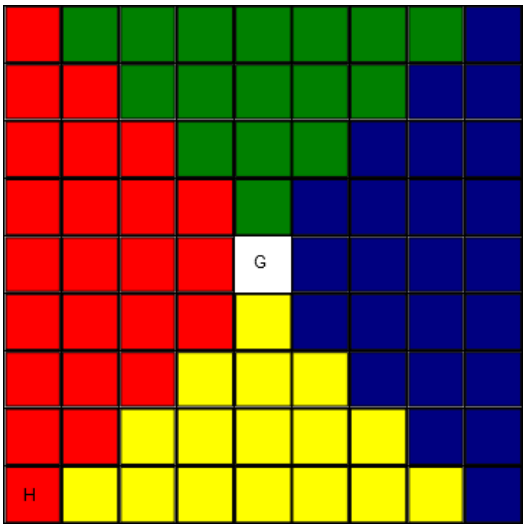

Figure 3.1: Example of search trees without using a heuristic: The Goal resides in the middle and the hunter is in the bottom left corner. Seperate colors show the four different search trees generated in a Manhattan connected environment. This is a worst case scenario as all the search trees fully touch each other across the entire map.

If the initial search is run with a heuristic, the map will look as in Fig. 3.3. Only two search trees actually expand in this case, as they are equal distance away from the hunter. As the goal moves, the leaves and nodes surrounding the goal must be recalculated as marked in pink in Fig. 3.4. Similar to the previous case, the nodes surrounding the goal must always be recalculated along with the leaf nodes which touch from separate search trees. Here the scaling factor from the non heuristic scenario is divided by 4, leading to a scaling factor shown in (3.3).

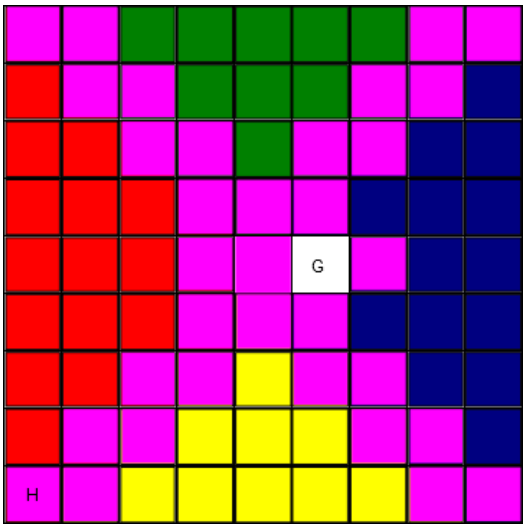

Figure 3.2: Full grid search after the goal moves to the right: The pink squares show the nodes that would be recalculated in this scenario and are estimated to scale with (3.2)

$$Worst\_Case\_Complexity\_(Heuristic) = \sqrt{(2 * pathlength^2)}/2 \qquad (3.3)$$

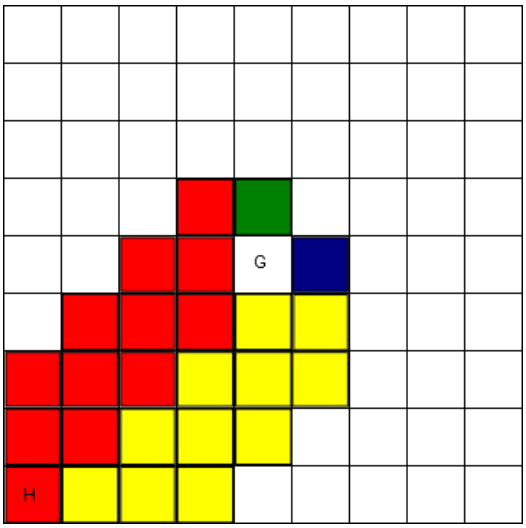

Figure 3.3: A full grid search with a heuristic with the goal in the center and hunter in the bottom left: the heuristic only allows expansion of the nodes from the goal towards the hunter instead of in a wave around the goal. The search trees generated fully touch each other over a quarter of the map.

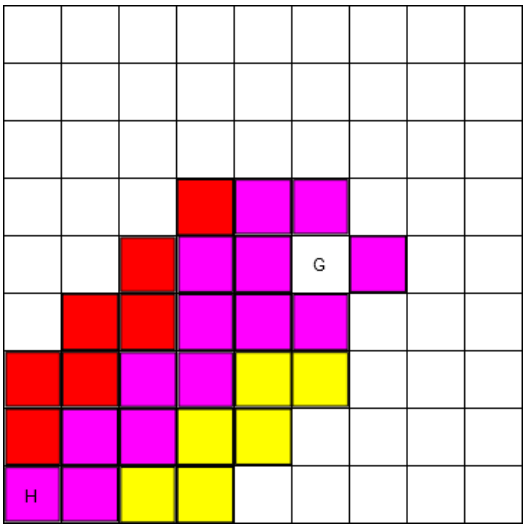

Figure 3.4: Heuristic grid search after the goal moves to the right: Only a quarter of the diagonals need to be recalculated in this case, leading to the estimated scaling factor shown in (3.3)

A best case scenario was formed where the hunter was in line with the goal (Fig. 3.5). When combined with a heuristic only one tree is expanded leaving no area where the leaves of search trees touch. When the goal moves, as shown in Fig. 3.6, only the nodes directly surrounding the goal require to be recalculated. This leads to only 7 nodes having to be recalculated. The best case value is independent of path length or grid size.

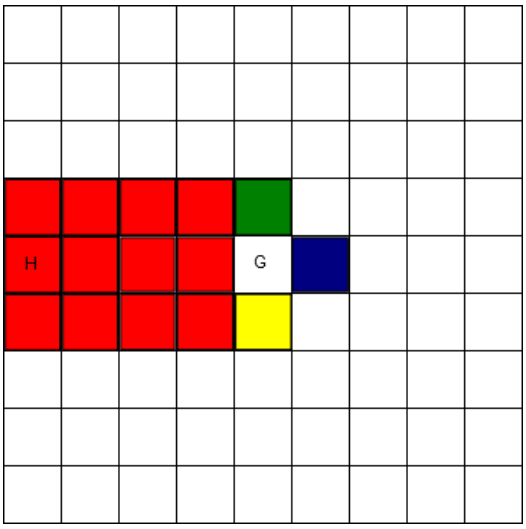

Figure 3.5: Best case scenario with the goal in the center and the hunter directly to the left: only one search tree is expanded towards the hunter with the search trees touching each other only in two places.

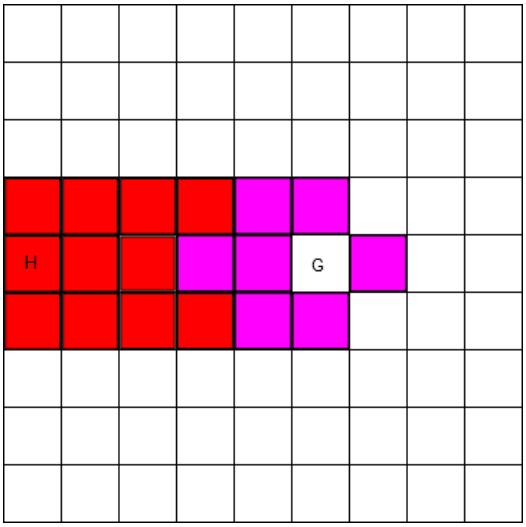

Figure 3.6: Best case scenario after the goal moves right: only the immediate surrounding nodes require recalculation, leading to 7 nodes needing expansion. This is true no matter how far the hunter is from the goal and gives a best case scenario which does not increase or decrease with path length or grid size.

Experiments

An experimental environment as similar as possible to the environment used in the testing of moving target D* lite [1] was created. The Matlab code which accomplishes this is in Appendix A. The environment consists of a NxN grid of nodes. During the creation of the environment, each node had a 25% chance of being marked as an obstacle. The grid was assumed to be four connected to only allow Manhattan movement. The start and goal position were then placed on random nodes within the grid. For the movement of the goal, a random node was chosen that the goal would move to. The goal moves 9 times for every 10 times the hunter moves so that even if the goal is heading in the direct opposite direction of the hunter, the hunter can catch the goal given enough time. The goal heads in a straight path towards the node chosen, randomly choosing between vertical or horizontal movement. If the goal hits an obstacle, a new node is selected for it to move towards. If no path from the hunter to the goal can be found, the environment is reset.

The environment was then tested by running the moving target path planner within it using a 1000x1000 grid over 1000 maps and comparing the results to those within the paper describing moving target D* lite [1]. The results are given in Table 3. As the exact maps used in Sun's paper were not able to be used, slight variations in the data were to be expected. In the maps run with the moving target path planner, a 5% longer path was observed with 13% fewer searches. These differences indicated that the maps run for the moving target path planner may have had the goal move into a corner, reducing the movements it could make, more often than moving target D* lite or the moving target path planner may be a non-optimal algorithm. A reduction in the number of nodes expanded per search, the main indicator of complexity within Sun's paper, was also seen. The moving target path planner had 257 times less complexity per search versus A*, and 12.3 times less complexity than MT-D* Lite. This reduction in complexity directly correlates with a reduction in the time taken

to run the algorithm, meaning that for a small increase in path length the moving target path planner was able to more quickly find a path.

Table 3.1: Comparing data from MT-D* Lite [1] to moving target path planner and environment. In the newly created environment with the moving target path planner there was: 5% longer path, 13% less searches, 257 times less complexity per search versus A*, 12.3 times less complexity than MT-D* Lite. These results show us that the environments perform similarly with the moving target path planner exchanging a slight increase in path length for a large reduction in complexity.

Algorithm Name	Searches Per Map	Moves per map	Expanded nodes per search (Complexity)
A*	379	689	14156
MT-D* lite	383	688	679
Moving Target Path Planner	335	722	55

Six sets of trials were then run so that the sensitivity of the Moving Target Path Planner's optimality and complexity as they relate to path length and grid size could be analyzed. These sets of trials were as follows:

(1) Optimality in a static grid size: The first set of trials to test the optimality of the path the moving target algorithm could find. Two identical 100x100 grids were used, one running with the moving target path planner algorithm and the other with D*. D*'s Matlab code can be found in Appendix D. Both grids had the same initial hunter and goal nodes along with the goal node following the same path. 1000 randomly created grids were tested using this method.

(2) Optimality in varying grid sizes: The second set of trials evaluates the sensitivity of complexity to grid size. These sets of trials only ran the moving target path planner. Grid sizes of N = 10,11,12....100 ,for a NxN grid with 1000 grids for each size N were used.

(3) Optimality in varying grid sizes expanded: The third set of trials evaluates the sensitivity of complexity to grid size and expands upon the second set of trials. A grid size of N = 150, 200, 250 ..1000 ,for a NxN grid with 100 grids for each size N were used. Fewer trials were run compared to previous sets so that the data could be acquired quickly.

(4) Complexity in a static grid size: The fourth set of trials tests the complexity of the new algorithm and verifies the results in Table 3. These trials only ran the moving target path planner on 1000 randomly generated grids of size 1000x1000.

(5) Complexity in varying grid sizes: In order to check the moving target path planner's sensitivity to grid size the fifth trials ran both the moving target path planner and D*. Again, N was scaled from 10-100, using 100 grids for each size. In this scenario, D* was run from scratch every time the goal moved, in order to find the optimal path at any point.

(6) Complexity with varying path length: The sixth set of trials was used to test scaling of path length with grid size, and scaling of complexity with with path length, running only the moving target path planner. Scaling N with values 50,250,500,750,1000 were used with 1000 grids for each size.

CHAPTER FOUR

Experimental Results

The results detail the sensitivity of the algorithm to grid size and initial distance between the hunter and goal. In this chapter the following are analyzed: The optimality and its sensitivity to grid size, the sensitivity of complexity to grid size, the sensitivity of complexity to initial distance between hunter and goal is analyzed. A relation between grid size and initial distance is also made.

Optimality

Optimality was tested in the first three sets of trials. End results showed that the moving target path planner was optimal in 38% of cases and had an average 1.5% increase in path length over the optimal path.

Figure 4.1 is a explains where the data used in the following figures was gathered from. Each grid size used had up to M trials with both the moving target path planner and Djikstra's returning the number of steps taken to reach the goal in y and x respectively. Equation 4.1 shows how the individual sub-optimality of each trial was found while the formula used to calculate the averaged sub-optimality is shown in (4.2).

$$Sub - optimality = (y_1 - x_1)/x_1 = E_1 \tag{4.1}$$

$$Averaged\_Sub - optimality = 1/M * \sum_{n=1}^{M} E_n \tag{4.2}$$

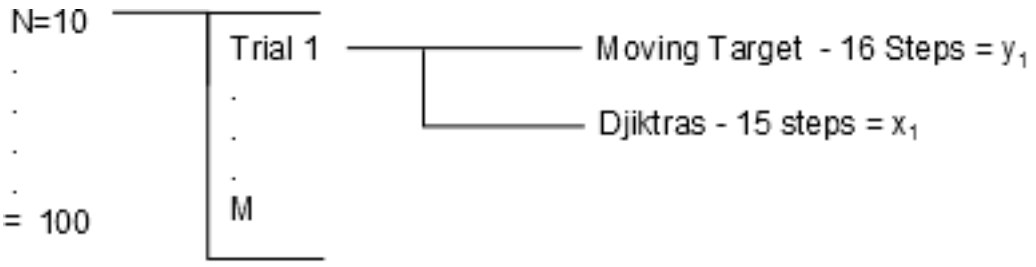

Figure 4.1: Data tree for sub-optimality, where N is the size of the grid, M is the number of trials ran, y is how long the moving algorithm's path is, and x is the length of the optimal path

In Fig. 4.2 the y-axis shows the percent sub-optimality of the moving target path planner's path versus an optimal path. If optimal path was 1 long and the path generated by the moving target path planner was 2 long, there would be a dot at x-axis 1 with an error of 100%. Figure 4.2 shows the correlation between path length and percent error in that more optimal path lengths lead to a lower variance in error. This can most likely be explained by the fact that the same number of steps off of the optimal path for a short path vs a longer path would lead to a much higher error. The grouping of most of the data around the x-axis shows that the algorithm is optimal or close to optimal, and is quantified in Fig.4.3. The curved groups of data that diminish as the optimal path length increases shows that in many cases the algorithm is only a couple nodes off optimal. This trade off of a few extra steps is good for a non-optimal algorithm, given the decrease in complexity the algorithm gives.

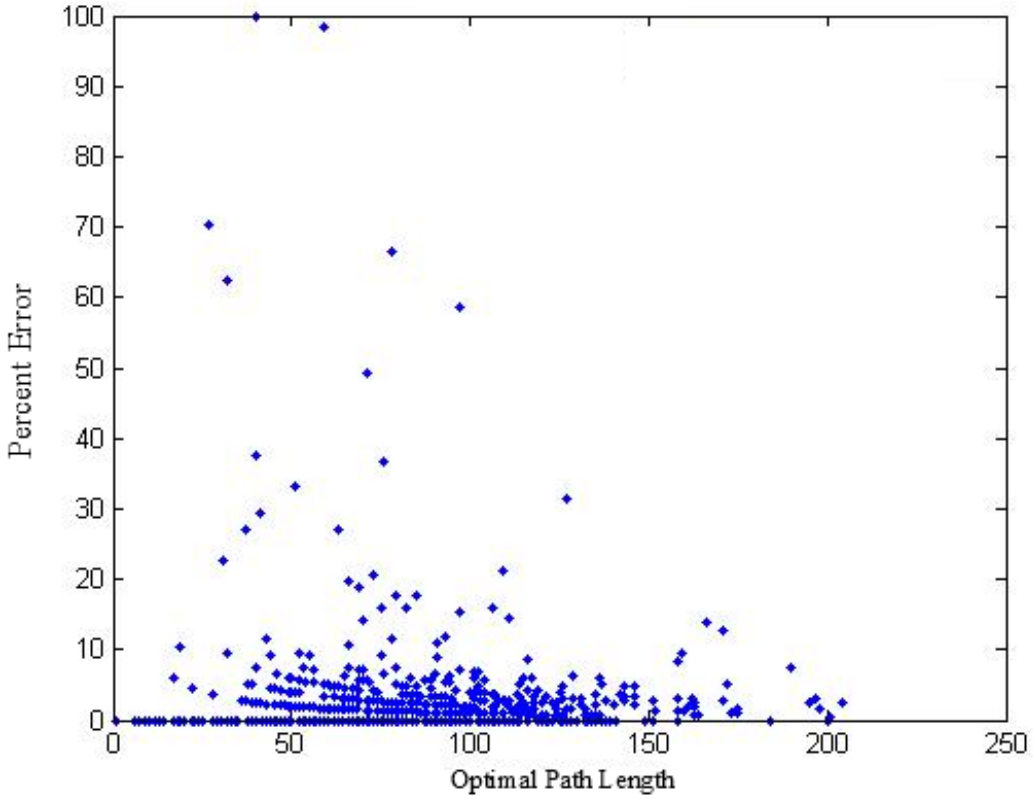

Figure 4.2: Scatter plot of 1000 trials sub-optimality at grid size N = 100 vs optimal path length. Data points were calculated using Eq. 4.1. The clustering of data near the x axis shows how most of the trials ended with low sub-optimality

Figure 4.3 is a cumulative distribution function (CDF) of the individual trials' error off optimal. The CDF describes the reliability of the new algorithm. Within the trials done, 90% of the trials fall within 6% error, and 95% of trials falling within 10% error. However most trials are sub-optimal, with only 35% of trials of trials having zero or near zero error. This steep curve confirms the fact that most of the trials are within a few steps of optimal. No cases were found where the moving target algorithm takes an incorrect route that would vastly increase the time needed to reach the goal. This level of reliability is important in systems where there is a set time limit or where a predictable path is required.

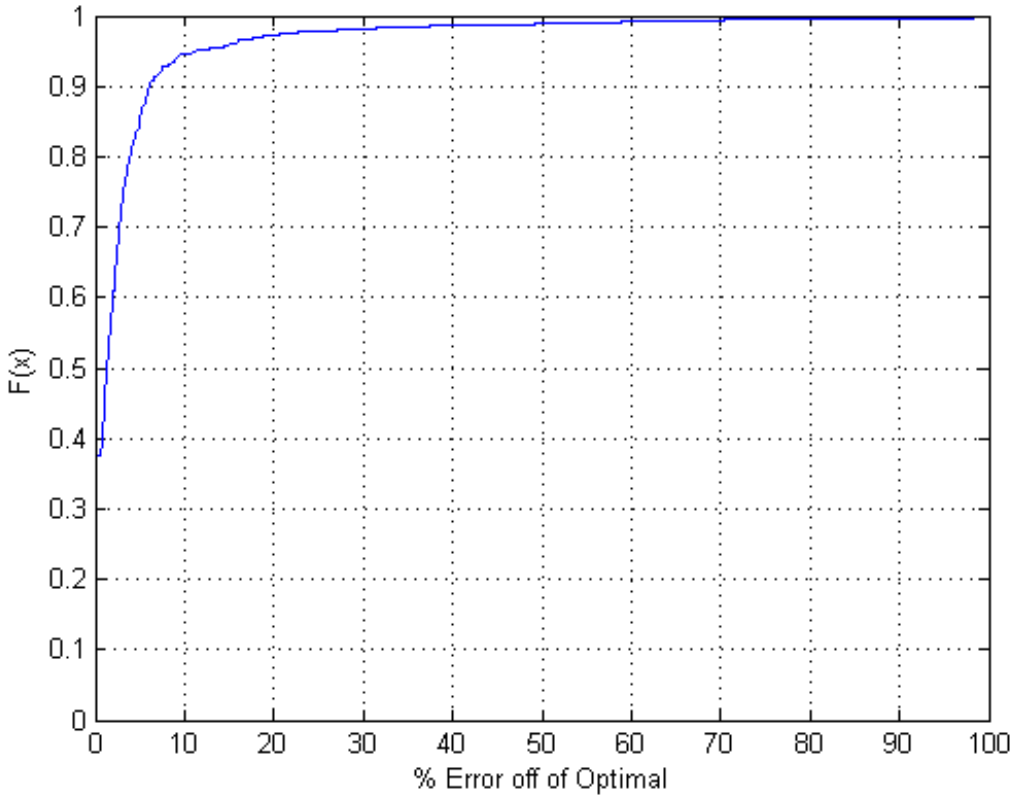

Figure 4.3: Cumulative distribution function of data in Fig. 4.2 with 95% of the trials falling under 10% longer than optimal path.

Figure 4.4 shows the relation between sub-optimality and grid size using the average error shown in (4.2). This data help shows how the optimality of the moving target algorithm scales as grid size increases. From grid size 10 to 30, the averaged error grows reaching a stable point at around 1.5%. The initial growth is most likely due to the fact that in smaller grid sizes the initial search dominates the results due to the shorter average path. The moving target algorithm increases in average sub-optimality until it reaches 1.5%. At this point the curve levels off and exhibits decreasing variance. Having a stable error at larger grid sizes means that this algorithm should be able to scale well where the reduction in complexity is more important and

opens up the possibility for more complex environments or cases where the target moves more in relation to the hunter.

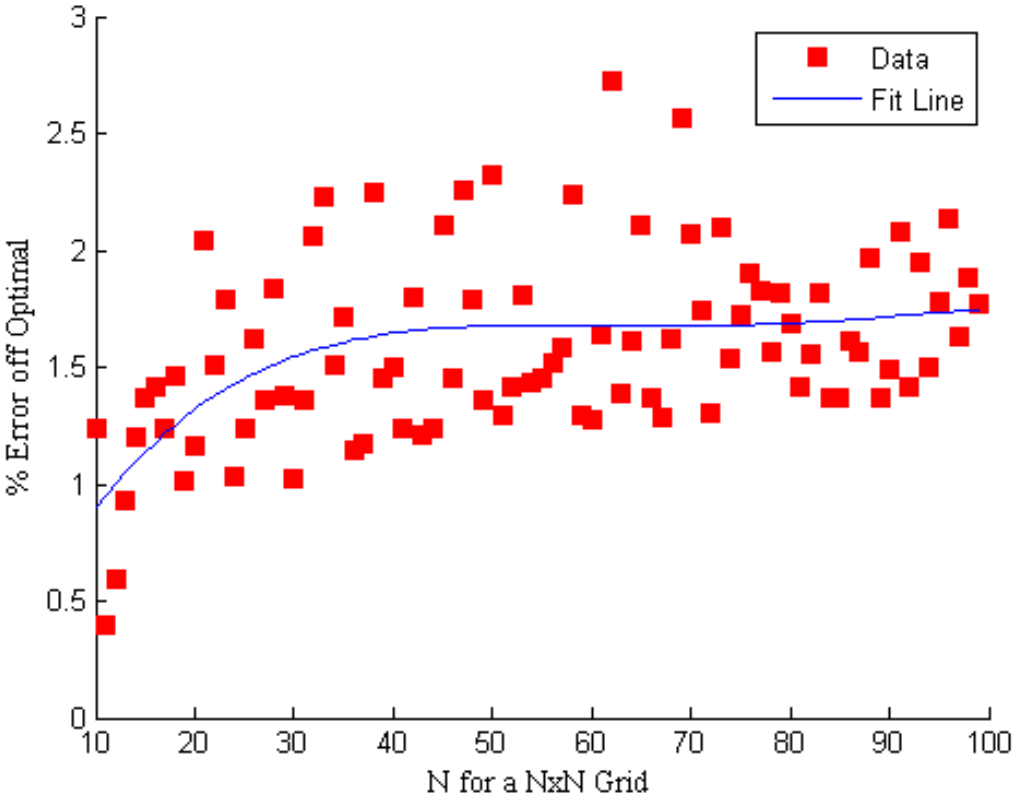

Figure 4.4: Averaged sub-optimality relative to grid size: A quadratic fit was made to the data to reveal any trends. The sub-optimality slowly increases from small grids, and at a grid size of 40 stabilizes out at 1.5% averaged sub-optimality. This shows the algorithms robustness to grid size.

Complexity

Complexity of the algorithm was analyzed in two ways. Once with relation to the grid size, and again with relation to the distance between the start and the goal. This was done so that the algorithm could be analyzed in relation to the number of nodes available to be expanded, and the expected path length separately.

Grid Size Results

The data gathered from both the second set of trials and the third set of trials were used for the grid size results. Figure (4.5) shows how the data was acquired. Each trial has a set of searches, an initial search when the map is created, and a re-plan every time the goal moved. When each search was done the number of nodes expanded, or recalculated, was counted. This is done because the expansion of nodes is the most repeated process within A* based path planners. The number of nodes recalculated from finding the initial path to reaching the goal were then summed (4.3), to create a total complexity of that search. The number of searches was also summed as denoted by X_1. Using this data and Equations 4.3 - 4.6, a metric of complexity was calculated. The main metric used for complexity is the average number of nodes recalculated per search. This correlates directly to how long re-planning a path will take when the goal moves.

$$Total\_nodes = \sum_{n=1}^{X_i} Y_n = T_i \tag{4.3}$$

$$Average\_Nodes\_Per\_Search = T_i/X_i = K_i \tag{4.4}$$

$$Averaged\_number\_of\_searches = 1/M * \sum_{n=1}^{M} X_i \tag{4.5}$$

$$Averaged\_complexity = 1/M * \sum_{n=1}^{M} K_i \tag{4.6}$$

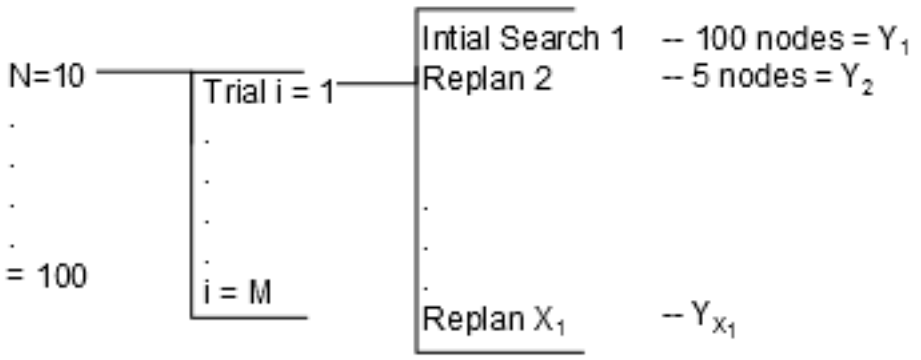

Figure 4.5: Data Tree for complexity with N is the grid size, M is the number of trials per grid size, X is the number of re-plans needed plus one, and Y is the number of nodes expanded per search.

Figure 4.6 was generated using the fourth trial's data. The x-axis shows the size of the grid while the y-axis indicates complexity using (4.6) to generate the points. This graph shows how the complexity of the algorithm changes as the grid size increases. For the most part, the complexity slowly increases as the size increases. The high values of complexity for $10 < \mathrm{N} < 20$ is due to the initial search calculating many nodes, but few re-plans to bring the average down. This confirms what was seen in Fig. 4.4 where the error increased at low grid sizes and then stabilized around a grid size of 30. The increase in complexity appears to be increasing linearly, which is promising since the grid complexity increases with a factor of N^2. Having a reduction in complexity versus the map is very important if the algorithm was to be used on a higher dimension map or in very large environments.

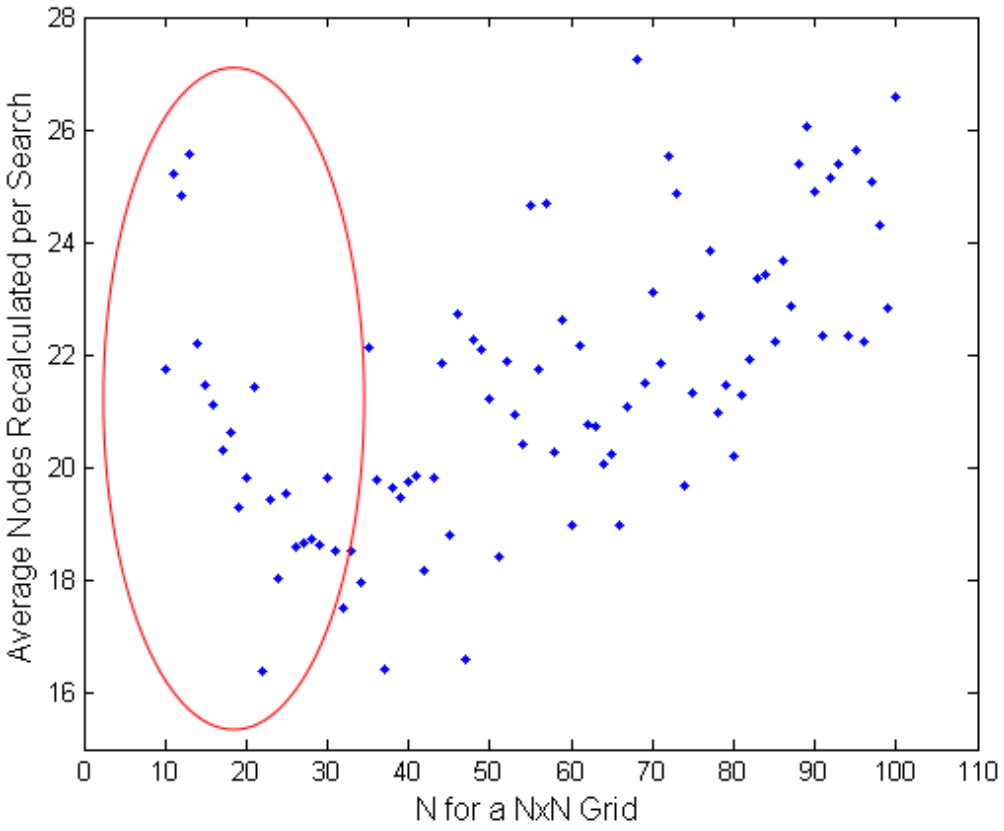

Figure 4.6: Complexity vs Size: The data was calculated using 4.6 which is indicative of the time complexity. The data within the red oval is decreasing as the initial searches complexity is mitigated by the re-planning algorithm. Once the re-planner dominates the search, complexity begins to increase linearly with grid size.

Figure 4.6 was expanded using data from the fifth set of trials to further evaluate the complexity's sensitivity to increasing grid size, Fig. 4.7 shows this data. In order to create the fit lines only the initial data was used, as the data from the fifth set of trials only had 100 trials per grid size and therefore increased variance. As seen the data points fall between the linear and log fit, meaning the complexity scales somewhere between those two. The fact that most of the data seems to fall under the linear fit is promising, as it shows the complexity is not increasing at the rate of the map complexity of N^2, and instead follows the linear complexity estimates.

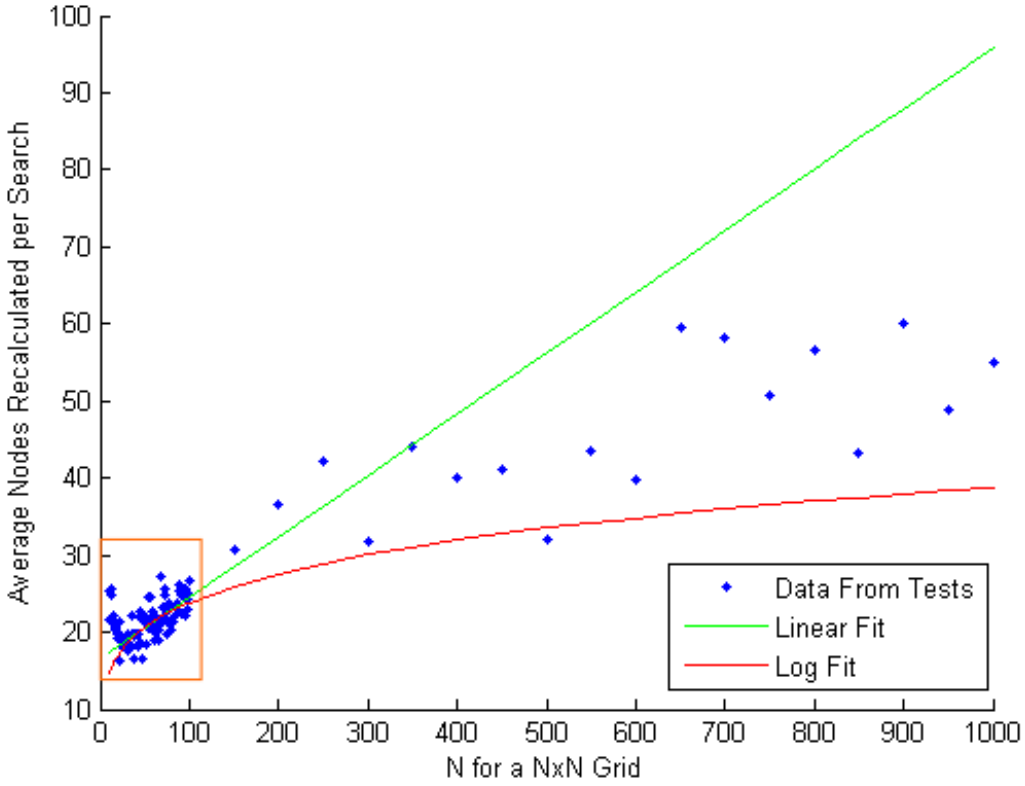

Figure 4.7: Expanded complexity vs size: The data within the orange box is from Fig. 4.6. The data from Fig. 4.6 was fit to a linear and logarithmic line. 78% of the data gathered from the sixth set of trials falls between these two lines. Therefore the complexity is shown to scale between linear and logarithmic with grid size.

The complexity analysis for a selected set of grid sizes was expanded by looking at a histogram of the complexities, Fig. 4.8. For the larger grid sizes (Fig. 4.8 b - e), points above 200 were removed to allow a better representation of the bulk of the data. All grid sizes have a large amount of cases within the data bin at 0. These cases are from when the goal only has one tree attached to it. This is due to the goal getting stuck in a corner. Cases of high complexity can either be explained as cases where the initial search dominates the searches, or where the search trees touch in several areas. A majority of the trials complexity resides below the 100 nodes per search with smaller grid sizes. As the grid sizes increase, the data shifts towards higher numbers

to follow the average increase in complexity. Having a majority of complexities in the lower range is good because it shows there are very few cases where the moving target algorithm will take a long time for re-plans. This is useful for real time applications and will allow for more accuracy as the algorithm can update more often as the target moves.

Figure 4.9 shows the averaged number of searches as in (4.5). The averaged number of searches was fit to a line. This verifies that as the grid size increases the path taken by the hunter will on average be longer. The average error off the linear approximation is 1.75% with 91% of data falling under 5% error off the fit line. This low amount of error helps show the reliability of the the algorithm's optimality in relation to size. This reliability is sought after, because it means that the algorithms behavior can be predicted.

Figure 4.10 shows the average nodes recalculated per trial as in (4.4). This represents the total complexity of getting the hunter from start to goal. This begins to increase at a slightly greater than linear rate as the grid size increases. This data also shows a fairly low variance, which shows the reliability of the algorithm. The slightly more than linear increase seen in the total complexity is to be expected. This is due to the increased number of searches compounded with the increased complexity of each of these searches at larger grid sizes.

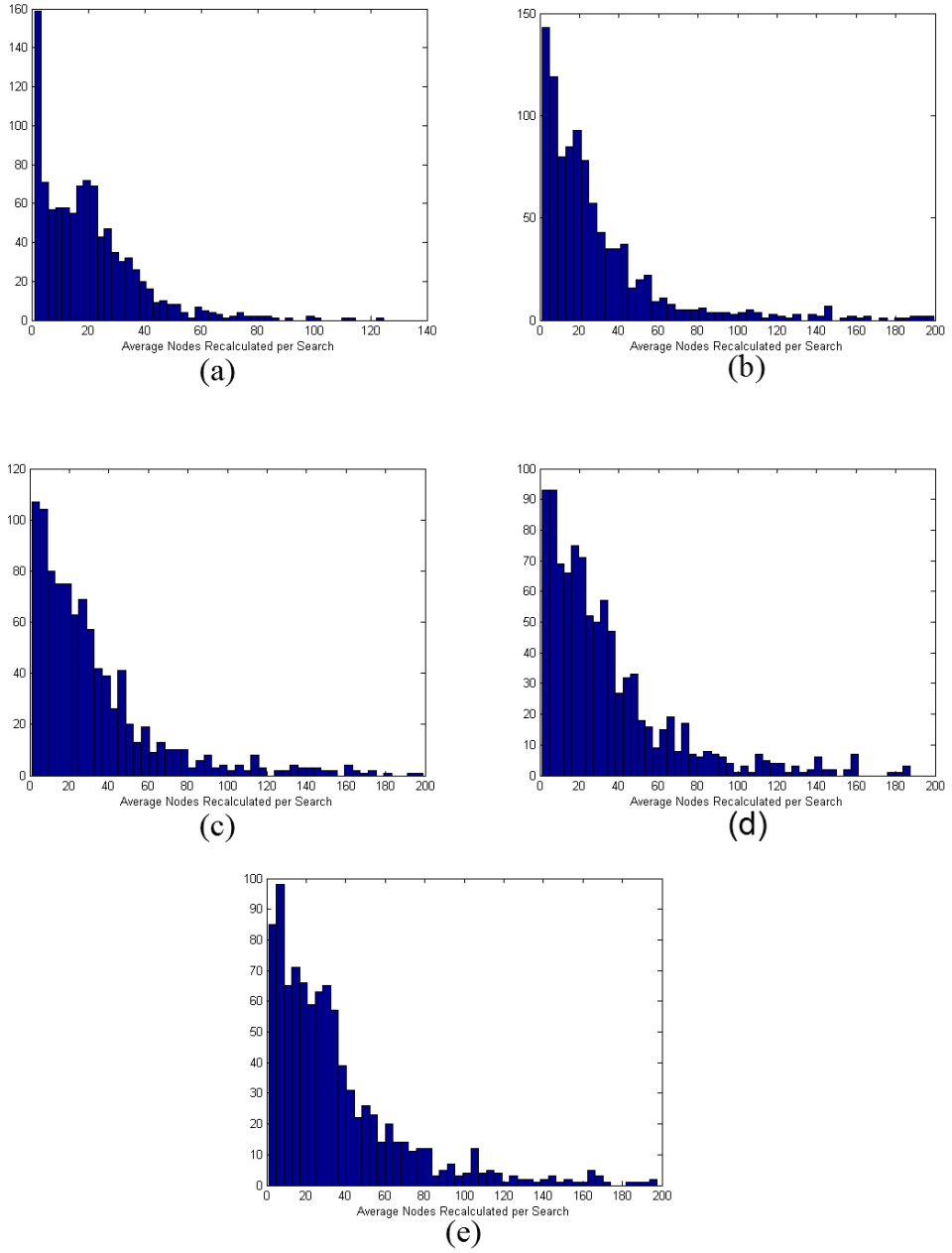

Figure 4.8: Complexity histogram over a set of grid sizes: a) is for N=50, b) N=250, c) N=500, d) N=750, e) N=1000. The spike at 0 complexity is due to cases where the goal only has one tree attached to it. Complexity slowly spreads away from 0 showing a slow increase in complexity as grid size increases. This shows that the Moving Target Path Planner stays reliable through the grid sizes.

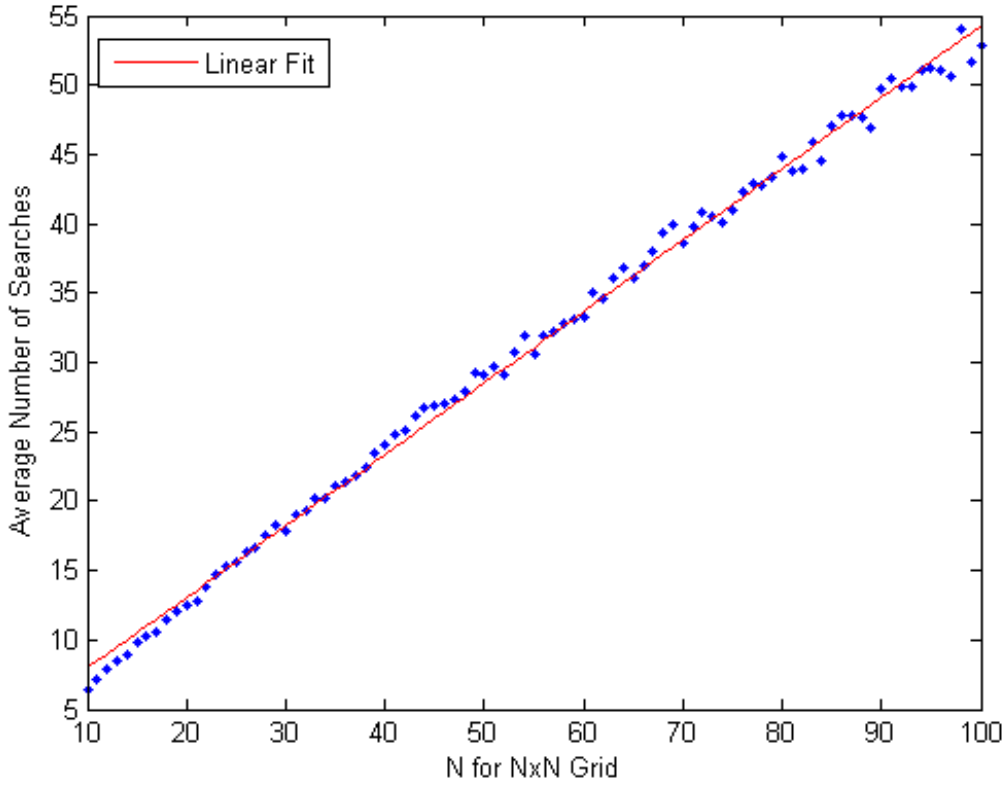

Figure 4.9: Searches vs grid size: The red fit line is a linear approximatation using the points plotted. The average error off the estimate is 1.75% with 91% of data falling under 5% error off the fit line. The low error is indicated low variance in the data. Low variance in the number of searches made supports the reliability of the algorithm.

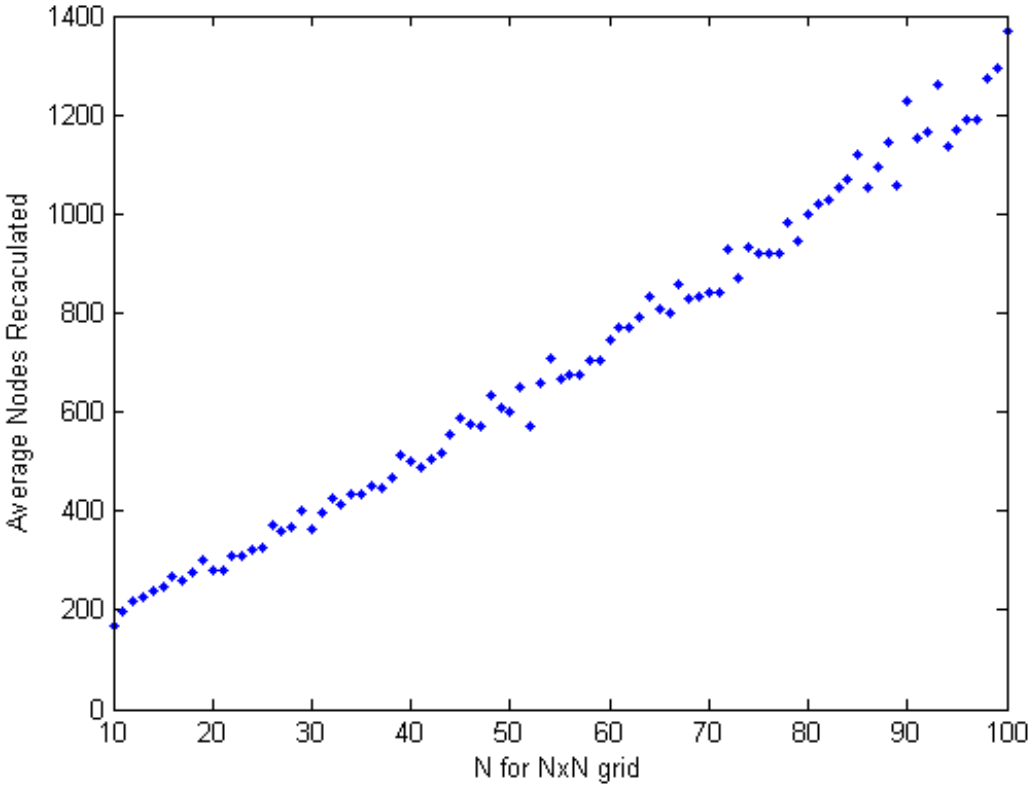

Figure 4.10: Total nodes recalculated vs grid size: The slightly greater than linear increase in total complexity coincides with the linear increase in number of searches and the increase in complexity per each of these searches. This shows that even with the reduction in complexity, as a map gets larger the total time for the algorithm to run and the hunter to reach the goal increases quickly.

Path Length Results

The data gathered from the sixth set of trials was used to analyze the complexity of the algorithm at different estimated path lengths. Path length was estimated as the initial straight line distance between the hunter and goal. This done over a set of grid sizes isolates the number of nodes available from the complexity, and allows for estimates of the complexity to be analyzed.

In Figure 4.11 the comparison of complexity to initial distance is shown. The number of recalculated nodes are averaged as shown in (4.4), for each trial in all the

grid sizes. This graph displays how the complexity of the algorithm changes with path length. More than 99% of the data falls under the lower worst case line generated from (3.2). This is important because it demonstrates empirical measurements matching theoretical predictions. The cases that are higher complexity are due to scenarios where the obstacles create more complex initial searches, leading to more complex search trees. The high complexity at low path length is due to the initial search dominating the search field. While 88% percent of the data falls under the worst case line from (3.3).

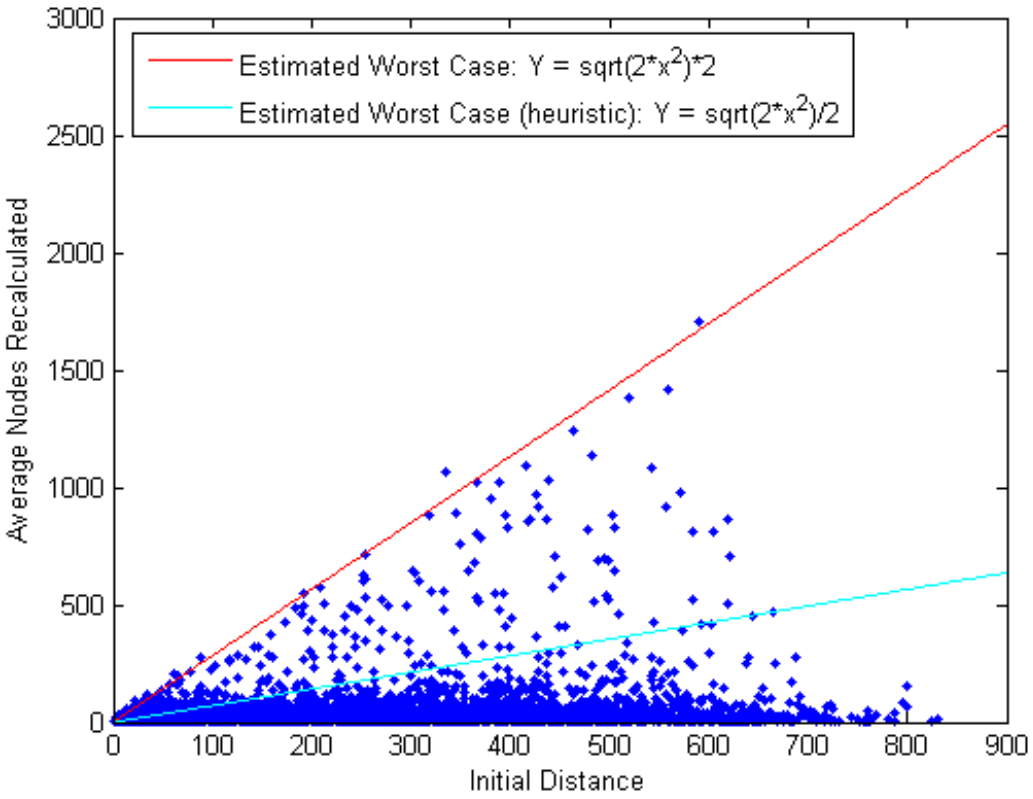

Figure 4.11: Initial distance vs complexity bounded by worst case estimates: Less than 1% of the data falls above the worst case line, with 90% of that data being at path lengths less than 40, where the initial search dominates the complexity. This validates the analytical estimates made in (3.2) & (3.3)

For Fig. 4.12, the data from Fig. 4.11 was averaged by taking each 10 data points closest in initial distance and averaging them. This was done to remove any outliers and gain a clearer picture of the data. In this case, the heuristic's worst case line from (3.3) and the best case line were used to bound the data. The short path length cases where the data exceeds the worst case scenario can be attributed to the initial search dominating the re-plans. The grouping of the data is mostly in the lower portion, and does not increase too much as the path length increases. This shows that most cases have fairly low complexity. No point passes the best case line and 88% of data falls under the worst case line. This shows the averaged complexity will scale between a linear and static with respect to initial distance between start and goal. Having this low complexity will allow for quicker re-planning. This allows for more accurate paths and longer runs without the worry of the target outrunning the algorithm.

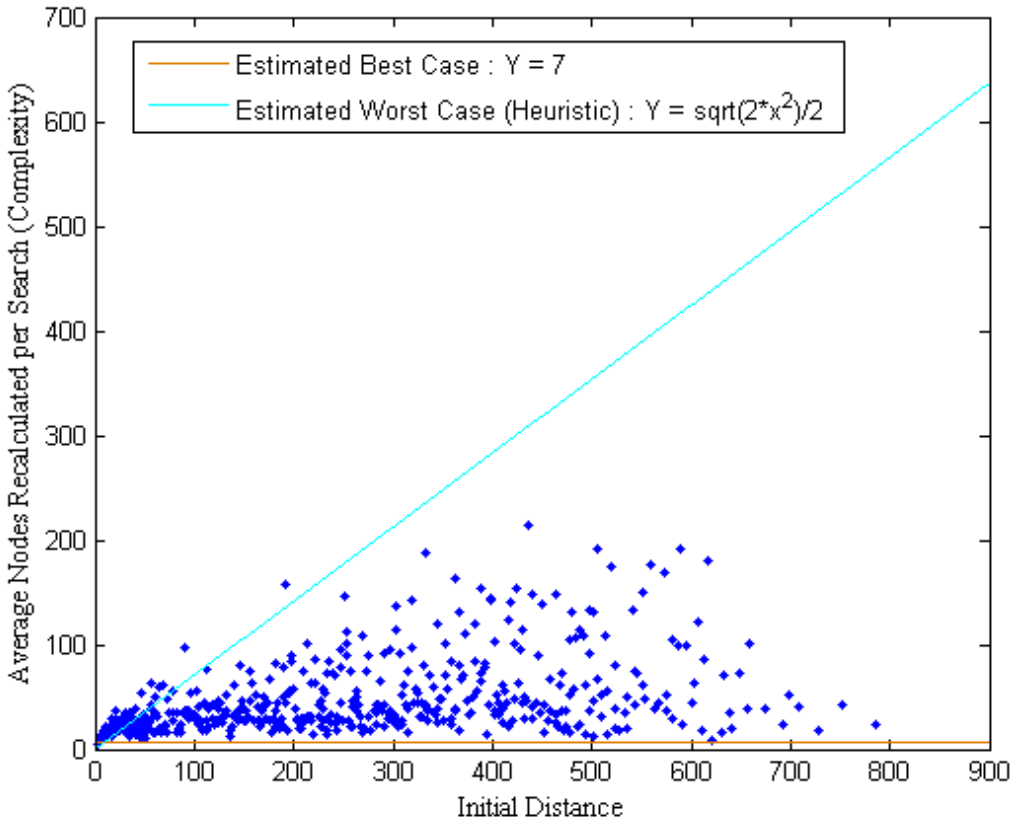

Figure 4.12: Path length vs complexity after averaging data from Fig. 4.11: All averages are above the best case scenario and 88% lie under the estimate. Clustering near the best case line shows worst case scenarios are rare within this environment. This also shows the algorithm on average scales well below the worst case estimate. This further validates the estimates made by (3.3) and Fig. 3.5.

In order to understand the relationship between grid size and optimal path length, the optimal path lengths from each grid size were separated and plotted into histograms (Fig. 4.13).

The straight line path length centers around a little under 1/2 of grid size, with the distribution favoring shorter paths. This is most likely due to the fact that longer paths are more likely to have impassible obstacles between the start and goal. Path length scales linearly with grid size. These patterns are to be expected, but show that in order to analyze extremely long path lengths a map may need to be created instead

of randomly generated. This also shows that the grid sizes used in previous results are a good correlate to optimal path length. Combined with previous results, this shows shows that the algorithm is able to handle extremely large grid sizes quickly as long as the initial path is short.

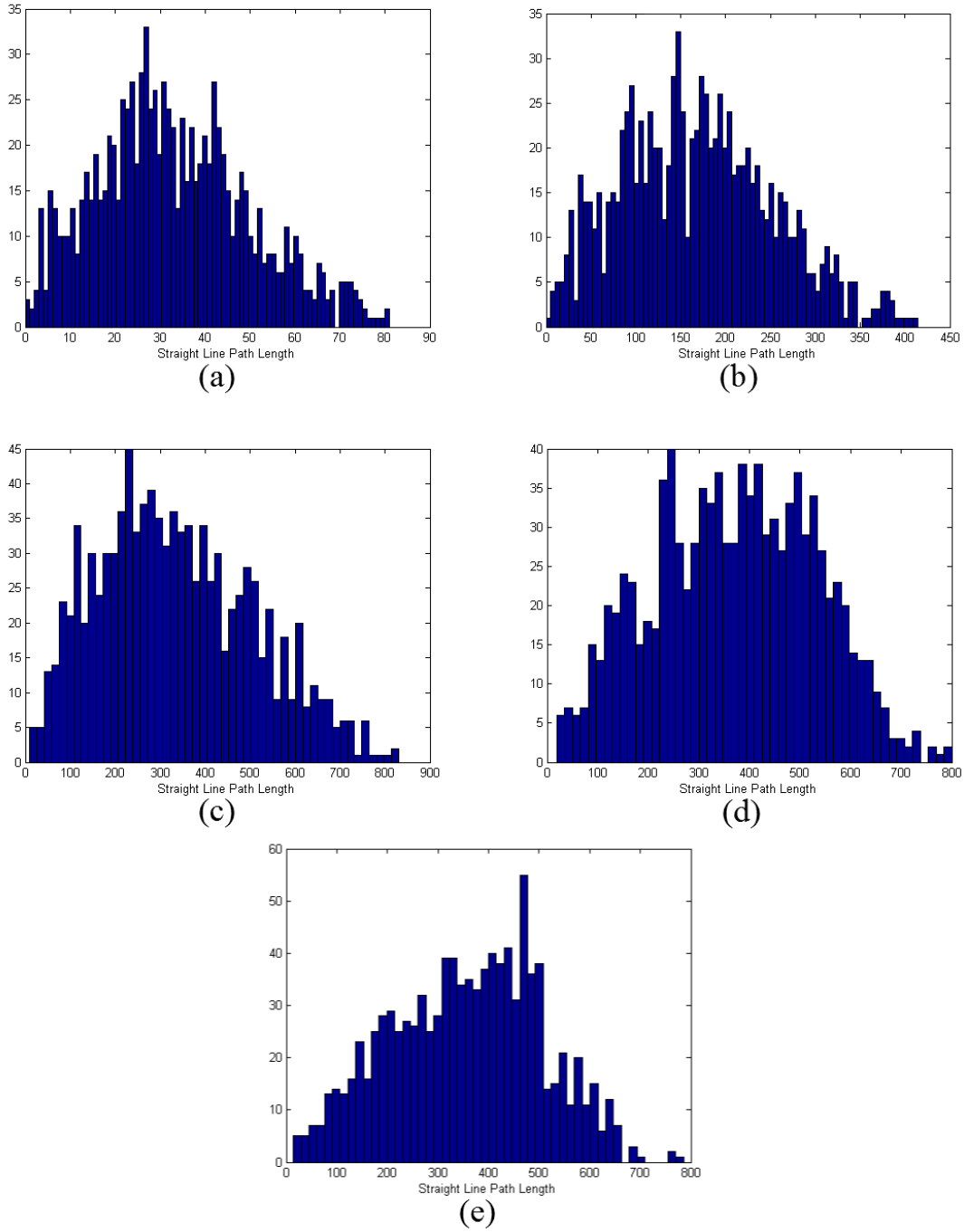

Figure 4.13: Histograms of initial distance based on grid size: a) is for N=50, b) N=250, c) N=500, d) N=750, e) N=1000. Data follows a normal function with a center close to 1/2 of grid size. It then tapers off on longer paths due to more possibility of no path being possible. This shows that grid size is a good correlate with initial path length.

CHAPTER FIVE

Discussion

This thesis describes a new path planner designed to pursue a moving target. This planner uses D* as a base for its initial path finding, and then takes data from its search trees to quickly re-plan the path when the target of the planner moves. In order to test this planner an environment was created based on work done in the paper describing Moving target D* lite [1]. This environment was tested by running the planner over 1000 trials, and then validated by comparing it to the data within MT-D* Lite paper. The new planner was found to have, on average: a 5% longer path, 13% fewer searches, 257 times less complexity per search versus A*, and 12.3 times less complexity than Moving target D* lite. This data shows that the planner traded complexity for optimality, allowing a longer path to be planned much faster than an optimal one.

The algorithm's complexity was estimated analytically by evaluating the planner's performance in both a worst case scenario and a best case scenario. It was estimated to, in the worst case, scale linearly with the distance from the hunter to the goal if the goal moved. In the best case scenario, the complexity was found to be a static number of 7 to only update the position of the goal and connect it to the previous path. These estimates were later tested and verified with empirical data.

A set of experiments were run so that the performance of the algorithm could be characterized by showing the sensitivity of optimality to grid size, and complexity to straight line path length and grid size. The experiments run showed that the algorithm increased, sub-optimality from .9% on average at low grid size, and then leveled out to 1.5% at larger grid sizes. This error in 90% of cases was found to be less than 5%, with 38% of cases being of optimal path length. This helped show the algorithm

planned path is not ever an extreme amount longer than the optimal path, and in a majority of cases is only a few nodes different in length.

The experiments run that related grid size to optimality showed that the algorithm scaled somewhere between linear and logarithmic as grid size increased. This is a good result as the complexity of the map increases with a factor of N^2. It was also found that the algorithm performed predictably amongst the trials. This reliability is a desired trait as the algorithm can be expected to perform in a particular way under all tested environments. When doing experiments which tested the sensitivity of the algorithm to optimal path length, first the analytical estimates made were verified. It was found amongst 5000 trials with initial path lengths varying from 1 to 850 nodes that: 99% of the data fell under the worst case estimate without a heuristic, 90% of the data fell under the worst case estimate with a heuristic, all cases fell above the best case estimate. These results verified the analytical estimates. The initial distance and grid sizes were then shown to correlate well with one another. This indicates that the data taken with a varying grid size provides more valuable information about the algorithm.

In this paper the algorithm developed was shown to:

- Be a complete planner that will find a path if one exists.
- Have a complexity that is predictable, lower than the current algorithms being used, and scales in a worst case scenario linearly with an increased initial path length.
- Be sub-optimal, but remain within 1.5% of optimal on average with no scaling with path length or grid size.
- Have reliable and predictable performance in a large variety of environments.

This algorithm is useful for moving target scenarios where generating and updating a path quickly takes priority over spending more time to generate an optimal path. This helps to ensure that the algorithm is able to keep up with the target.

APPENDICES

APPENDIX A

Top level code for Simulation

```
% movenorm = zeros(1);
% move = zeros(1);

%stats = struct('indmovenorm',movenorm,'indmove',move);
%sizetrials=10;

while sizetrials < 100;
trials = 100;

if montcount == trials
montcount =1;
%
% % define stuff

searchtotal = 0;
counttotal = 0;
movetotal= 0;
indsearch = zeros(1,trials);
indmove = zeros(1,trials);
indcount = zeros(1,trials);

searchtotalnorm = 0;
counttotalnorm = 0;
movetotalnorm = 0;
indsearchnorm = zeros(1,trials);
```

```
indmovenorm = zeros(1,trials);
indcountnorm = zeros(1,trials);
end

while montcount < trials
clearvars -except sizetrials searchtotal counttotal movetotal
   montcount indsearch indmove...
indcount trials map searchtotalnorm movetotalnorm
   counttotalnorm indsearchnorm indmovenorm indcountnorm stats

pblockversion =0; %set to 1 to make pblock version
mapcreate = 1; %set to 0 to use previous map data
planesize = sizetrials;
heur = 1;
if mapcreate  == 1


start = [randi([1 planesize-1]),randi([1 planesize-1])];
goal = [randi([1 planesize-1]),randi([1 planesize-1])];
% start = [2,2];
% goal = [7,7];

heur = 1;
%for testing with no movie
%     Movie = struct('map',{},'Dstar.tags',{},'Dstar.pointer
   ',{},'f',{},'j',{},'writerObj',{});
```

```
%Dstar.cost map
cost = zeros(planesize+1);
%Dstar.tags such as: where robot is, Dstar.goal, and whether a
   block has been
%calculated/ needs to be Dstarmovinggoal.recalculated.
tags = zeros(planesize+1);
%where each block "points to" to create a path
pointer = zeros(planesize+1,planesize+1,2);
%Dstar.pointer x,y,1 = x coordinate
%Dstar.pointer x,y,2 = y coord
closedset = zeros(1,2);
%set for the current "wavefront" so eronius calculations do not
    need to be
%made
openset = zeros(1,2);

%where the robot start
robottrack = zeros(1);

Dstar = struct('cost',cost,'tags',tags,'pointer',pointer,'
   closedset',closedset,...
'openset',openset,'robottrack',robottrack,'start',start,'goal'
   ,...
goal);

%set all blocks as uncalculated
for n=1:planesize
```

```
for p=1:planesize
g = rand;
if g < .25
Dstar.tags(n,p) = 'O';
else
Dstar.tags(n,p) = 'n';
end
end
end

%set all blocks as uncalculated
for n=1:10
for p=1:10
Dstar.tags(n,p) = 'n';
end
end
% %set obstacles in Dstar.tags
% Dstar.tags(6,1:4) = 'O' ;
% Dstar.tags(6,6:9) = 'O' ;
% Dstar.tags(5:9,8) = 'O' ;
%set robot in Dstar.tags
Dstar.tags(Dstar.start(1),Dstar.start(2)) = 'R';
%set Dstar.goal in Dstar.tags
Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 'G';

%point Dstar.goal to self
Dstar.pointer(Dstar.goal(1),Dstar.goal(2),1) = Dstar.goal(1);
Dstar.pointer(Dstar.goal(1),Dstar.goal(2),2) = Dstar.goal(2);
```

```
Dstar.robottrackx = Dstar.start(1);
Dstar.robottracky = Dstar.start(2);
Dstar.robottrackxold = Dstar.start(1);
map(montcount) = Dstar;
else
Dstar = map(montcount);
end

%where each blocks path ends up around the Dstar.goal
%parent block and blocks needed to be Dstarmovinggoal.
   recalculated if Dstar.goal moves
parentblock = zeros(planesize+1);
pblock = zeros(1,2);
pblockcount =0;
pblockcost = zeros(1);
recalc = zeros(1,2);
recalccount = 1;
Dstarmovinggoal = struct('parentblock',parentblock,'pblock',
   pblock,'pblockcount',pblockcount,'pblockcost',pblockcost,'
   recalccount',recalccount,'recalc',recalc);
Dstarnorm = Dstar;

%find Dstar.goal
%original path
```

```
count =0;
countnorm = 0;
j=1;

[ Dstar, Dstarmovinggoal, count,planesize,pathfound] =
   Dstarpathfindwpblock( Dstar, Dstarmovinggoal, count,
   planesize);

[ Dstarnorm, countnorm,planesize,heur,pathfoundnorm ] =
   Dstarpathfind( Dstarnorm, countnorm,planesize,heur);




move =0;
movenorm = 0;
search =0;
searchnorm =0;
goalmove=1;
newgoal = [randi([1 planesize-1]),randi([1 planesize-1])];
gmove =1;
checker =0;
blah =0;
checkernew = 0;


tic;
while Dstar.tags(Dstar.goal(1),Dstar.goal(2)) ~= 'R'
```

```
if pathfound ==0|| pathfoundnorm ==0
break
end


% movement for goal, chooses random point, moves towards it
   9/10 of robots
% moves
regoal=0;
if gmove < 10
regoal=0;
smove=0;
regoal=0;

if Dstar.goal(1) ~= newgoal(1)
if Dstar.goal(1) < newgoal(1)
if Dstar.tags(Dstar.goal(1)+1,Dstar.goal(2)) ~= 'O'
Dstar.tags(Dstar.goal(1)+1,Dstar.goal(2)) = 'G';
Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 0;
                        if Dstarnorm.tags(Dstar.goal(1),Dstar.
                           goal(2)) ~= 'R'
                            Dstarnorm.tags(Dstar.goal(1)+1,
                               Dstar.goal(2)) = 'G';
                            Dstarnorm.tags(Dstar.goal(1),Dstar.
                               goal(2)) = 0;
                        end
smove=1;
else
```

```
regoal=1;
end
elseif Dstar.goal(1) > newgoal(1)
if Dstar.tags(Dstar.goal(1)-1,Dstar.goal(2)) ~= 'O'
Dstar.tags(Dstar.goal(1)-1,Dstar.goal(2)) = 'G';
Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 0;
                        if Dstarnorm.tags(Dstar.goal(1),Dstar.
                           goal(2)) ~= 'R'
                            Dstarnorm.tags(Dstar.goal(1)-1,
                               Dstar.goal(2)) = 'G';
                            Dstarnorm.tags(Dstar.goal(1),Dstar.
                               goal(2)) = 0;
                        end
smove=1;
else
regoal=1;
end

end
else
regoal=1;
end


if Dstar.goal(2) ~= newgoal(2) && regoal==1
if Dstar.goal(2) < newgoal(2)
if  Dstar.tags(Dstar.goal(1),Dstar.goal(2)+1) ~= 'O'
Dstar.tags(Dstar.goal(1),Dstar.goal(2)+1) = 'G';
```

```
Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 0;

                        if Dstarnorm.tags(Dstar.goal(1),Dstar.
                           goal(2)) ~= 'R'
                            Dstarnorm.tags(Dstar.goal(1),Dstar.
                               goal(2)+1) = 'G';
                            Dstarnorm.tags(Dstar.goal(1),Dstar.
                               goal(2)) = 0;
                        end


regoal=0;
smove=1;
end
elseif Dstar.goal(2) > newgoal(2)
if Dstar.tags(Dstar.goal(1),Dstar.goal(2)-1) ~= 'O'
Dstar.tags(Dstar.goal(1),Dstar.goal(2)-1) = 'G';
Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 0;
                         if Dstarnorm.tags(Dstar.goal(1),Dstar.
                            goal(2)) ~= 'R'
                            Dstarnorm.tags(Dstar.goal(1),Dstar.
                               goal(2)-1) = 'G';
                            Dstarnorm.tags(Dstar.goal(1),Dstar.
                               goal(2)) = 0;
                        end
regoal=0;
smove=1;
end
end
```

```
else
regoal =1;
end


if regoal == 1
newgoal = [randi([1 planesize-1]),randi([1 planesize-1])];
blocked =0;
while blocked ==0
if Dstar.tags(newgoal(1),newgoal(2)) == 'O';
newgoal = [randi([1 planesize-1]),randi([1 planesize-1])];
else
blocked =1;
end
end
end

gmove = gmove+1;

else
gmove=1;
end


if goalmove ==10
goalmove=1;

else
```

```
goalmove = goalmove+1;

end

%end goal moving

%if Dstar.goal moves
%pblock version

if Dstar.tags(Dstar.goal(1),Dstar.goal(2)) ~= 'G' && Dstar.tags
   (Dstar.robottracky,Dstar.robottrackx) ~= 'G'


[ Dstar,Dstarmovinggoal, count] = ...
Dstarreplangoalmove( Dstar, Dstarmovinggoal, count,planesize );


Dstar.tags(Dstar.robottracky,Dstar.robottrackx) = 'R';
Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 'G';

search = search +1;
end

%non pblock version
if Dstarnorm.tags(Dstarnorm.goal(1),Dstarnorm.goal(2)) ~= 'G'
   && Dstarnorm.tags(Dstarnorm.robottracky,Dstarnorm.
   robottrackx) ~= 'G'
for n=1:planesize
```

```
for p=1:planesize
if Dstarnorm.tags(n,p) ~= 'R' && Dstarnorm.tags(n,p) ~= 'O'&&
   Dstarnorm.tags(n,p) ~= 'G'
Dstarnorm.tags(n,p) = 'n';
elseif Dstarnorm.tags(n,p) == 'G'
Dstarnorm.goal = [n p];
end

end
end
Dstarnorm.openset = zeros(1,2);
Dstarnorm.closedset = zeros(1,2);
Dstarnorm.cost = zeros(planesize+1);
Dstarnorm.pointer = zeros(planesize+1,planesize+1,2);
[ Dstarnorm, countnorm,planesize,heur,pathfoundnorm ] =
   Dstarpathfind( Dstarnorm, countnorm,planesize,heur);

searchnorm = searchnorm+1;
end
```

```
%end Dstar.goal moving stuff


%do normal stuff here

if Dstar.tags(Dstar.goal(1),Dstar.goal(2)) == 'R' || (Dstar.
   tags(Dstar.robottracky,Dstar.robottrackx) == 'G' ...
 && Dstarnorm.tags(Dstarnorm.robottracky,Dstarnorm.robottrackx)
     == 'G')
pathfound = 0;
break
end
if pathfound == 0
break
end

if Dstar.tags(Dstar.robottracky,Dstar.robottrackx) ~= 'G' ||
   Dstar.tags(Dstar.goal(1),Dstar.goal(2)) ~= 'R'
%go through path
oldcost = Dstar.cost(Dstar.robottracky,Dstar.robottrackx);
Dstar.tags(Dstar.robottracky,Dstar.robottrackx) =0;
Dstar.cost(Dstar.robottracky,Dstar.robottrackx) =oldcost;
Dstar.tags(Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,1)
   ,Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,2)) = 'R'
   ;
move = move+1;
```

```
Dstar.robottrackx = Dstar.pointer(Dstar.robottracky,Dstar.
   robottrackx,2);
Dstar.robottracky = Dstar.pointer(Dstar.robottracky,Dstar.
   robottrackxold,1);
Dstar.robottrackxold = Dstar.robottrackx;
if Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,1)== Dstar
   .robottracky...
&& Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,2)== Dstar
   .robottrackx
checker = checker+1;
else
checker = 0;
end

end




if Dstarnorm.pointer(Dstarnorm.robottracky,Dstarnorm.
   robottrackx,1) ~= 0
if Dstarnorm.tags(Dstarnorm.robottracky,Dstarnorm.robottrackx)
   ~= 'G' || Dstarnorm.tags(Dstarnorm.goal(1),Dstarnorm.goal(2)
   ) ~= 'R'
%go through path
oldcost = Dstarnorm.cost(Dstarnorm.robottracky,Dstarnorm.
   robottrackx);
Dstarnorm.tags(Dstarnorm.robottracky,Dstarnorm.robottrackx) =0;
```

```
Dstarnorm.cost(Dstarnorm.robottracky,Dstarnorm.robottrackx) =
   oldcost;
Dstarnorm.tags(Dstarnorm.pointer(Dstarnorm.robottracky,
   Dstarnorm.robottrackx,1),Dstarnorm.pointer(Dstarnorm.
   robottracky,Dstarnorm.robottrackx,2)) = 'R';
movenorm = movenorm+1;
Dstarnorm.robottrackx = Dstarnorm.pointer(Dstarnorm.robottracky
   ,Dstarnorm.robottrackx,2);
Dstarnorm.robottracky = Dstarnorm.pointer(Dstarnorm.robottracky
   ,Dstarnorm.robottrackxold,1);
Dstarnorm.robottrackxold = Dstarnorm.robottrackx;
if Dstarnorm.pointer(Dstarnorm.robottracky,Dstarnorm.
   robottrackx,1)== Dstarnorm.robottracky...
&& Dstarnorm.pointer(Dstarnorm.robottracky,Dstarnorm.
   robottrackx,2)== Dstarnorm.robottrackx
checkernew = checkernew+1;
else
checkernew=0;
end

% if there is no path break
end
end

if checkernew >= 5 || checker >= 5
checkernew;
checker;
break
```

```
end

if Dstar.tags(Dstar.goal(1),Dstar.goal(2)) == 'R'...
&& Dstarnorm.tags(Dstarnorm.goal(1),Dstarnorm.goal(2)) == 'R'
break
end
time = toc;
if time > 600
toc
break
end

end
%goal has been reached


%keep count of stats
avgtime =0;
if Dstar.tags(Dstar.goal(1),Dstar.goal(2)) == 'R' && Dstarnorm.
   tags(Dstarnorm.goal(1),Dstarnorm.goal(2)) == 'R'
%put normal count in here
searchtotalnorm = searchtotalnorm + searchnorm;
counttotalnorm = counttotalnorm + countnorm;
movetotalnorm = movetotalnorm + movenorm;
indsearchnorm(montcount) = searchnorm;
indmovenorm(montcount) = movenorm;
indcountnorm(montcount) = countnorm;
```

```
searchtotal = searchtotal + search;
counttotal = counttotal + count;
movetotal= movetotal + move;
indsearch(montcount) = search;

indmove(montcount) = move;
indcount(montcount) = count;
montcount = montcount+1;

end



end
sizetrials
indmovenorm;
%display stats
nsearchavg = searchtotal/(montcount-1);
ncounttotal = counttotal/(montcount-1);
nmovetotal= movetotal/(montcount-1);
navgcounttotal = ncounttotal/nsearchavg;

%stuff
%     for x=1:99
%          avgerror(x) = mean(percenterrrorvspathlength(1:x));
%          avglength(x) = mean(newtest(1:x));
%     end
stats.movenorm(sizetrials-9,1:montcount) = indmovenorm;
stats.move(sizetrials-9,1:montcount) = indmove;
```

```
sizetrials = sizetrials+1;
save('DifferentSizes');

end
```

DstarDrakewithmovinggoalbothscaling.m

APPENDIX B

Initial Path Finding Code for moving target algorithm

```
function [ Dstar, Dstarmovinggoal, count,planesize,pathfound ]
   = Dstarpathfindwpblock( Dstar, Dstarmovinggoal, count,
   planesize)
pathfound =0;
heur = 100000;
setcount =1;
% calc Dstar.cost around Dstar.goal
%set parrent Dstar.pointers
x=1;
for n = Dstar.goal(1)-1:Dstar.goal(1)+1
for p = Dstar.goal(2)-1:Dstar.goal(2)+1
if n > 0 && p >0 && n < planesize+1 && p < planesize+1
if abs(n-Dstar.goal(1))+abs(p-Dstar.goal(2)) <=1
if Dstar.tags(n,p) ~= 'O' && (n ~= Dstar.goal(1) || p ~= Dstar.
   goal(2));
if Dstar.tags(n,p) == 'n' && Dstar.tags(n,p) ~= 'R';
%set Dstar.cost, tag as calculated
Dstar.cost(n,p) = 1;
Dstar.tags(n,p) = 'p';
%Set parent block numbers and remember locations for if Dstar.
   goal moves
Dstarmovinggoal.parentblock(n,p) = x;
Dstarmovinggoal.pblock(x,1) =n;
Dstarmovinggoal.pblock(x,2) =p;
Dstarmovinggoal.pblockcount = Dstarmovinggoal.pblockcount+1;
```

```
x=x+1;
%point to Dstar.goal
Dstar.pointer(n,p,1) = Dstar.goal(1);
Dstar.pointer(n,p,2) = Dstar.goal(2);
%add to current wavefront and increase setcount
Dstar.openset(setcount,1) = n;
Dstar.openset(setcount,2) = p;
setcount = setcount+1;
%add block to calculation Dstar.costs
end
end
end
end
end
end

%create a open set to be determined
newcurrentset = zeros(1,2);
newsetcount =1;
%add current open set to list of blocks
Dstar.closedset = Dstar.openset;
setcount = setcount-1;
%find neighbors
%calculate Dstar.cost of neighbors
%repeat untill path is found
%need to add a heuristic here
tic
while pathfound ==0
```

```
for m = 1:min(setcount,heur)
for n = Dstar.openset(m,1)-1:Dstar.openset(m,1)+1
for p = Dstar.openset(m,2)-1:Dstar.openset(m,2)+1
if abs(n-Dstar.openset(m,1))+abs(p-Dstar.openset(m,2)) <=1
if n > 0 && p >0 && n <= planesize && p <= planesize
if Dstar.tags(n,p) ~= 'O' && (n ~= Dstar.openset(m,1) || p ~=
   Dstar.openset(m,2));
if (Dstar.cost(n,p) > Dstar.cost(Dstar.openset(m,1),Dstar.
   openset(m,2))+1 || Dstar.tags(n,p) == 'n') && Dstar.tags(n,p
   ) ~= 'G';
%set Dstar.pointer to lowest Dstar.cost
%future: add random here if multiple possible Dstar.pointers
Dstar.pointer(n,p,1) = Dstar.openset(m,1);
Dstar.pointer(n,p,2) = Dstar.openset(m,2);
%EXPERIMENTAL FOR MOVING TARGET set parent
%block to parent block of where we are pointing
Dstarmovinggoal.parentblock(n,p) = Dstarmovinggoal.parentblock(
   Dstar.openset(m,1),Dstar.openset(m,2));

%add block to new set
newcurrentset(newsetcount,1) =n;
newcurrentset(newsetcount,2) =p;
newsetcount = newsetcount+1;
% tag as calculated and add up Dstarmovinggoal.recalculations
if Dstar.tags(n,p) == 'n'
Dstar.tags(n,p) = 0;
end
```

```
% give new Dstar.cost
Dstar.cost(n,p) = Dstar.cost(Dstar.openset(m,1),Dstar.openset(m
   ,2))+1;

end



if Dstarmovinggoal.parentblock(n,p) ~= Dstarmovinggoal.
   parentblock(Dstar.openset(m,1),Dstar.openset(m,2)) && Dstar.
   cost(n,p) > 1 && Dstar.cost(Dstar.openset(m,1),Dstar.openset
   (m,2)) > 1
if Dstarmovinggoal.parentblock(n,p) ~= 0 && Dstarmovinggoal.
   parentblock(Dstar.openset(m,1),Dstar.openset(m,2)) ~= 0
Dstarmovinggoal.recalc(Dstarmovinggoal.recalccount,1) = n;
Dstarmovinggoal.recalc(Dstarmovinggoal.recalccount,2) = p;
Dstarmovinggoal.recalc(Dstarmovinggoal.recalccount+1,1) = Dstar
   .openset(m,1);
Dstarmovinggoal.recalc(Dstarmovinggoal.recalccount+1,2) = Dstar
   .openset(m,2);
Dstarmovinggoal.recalccount = Dstarmovinggoal.recalccount+2;

end
end


end
```

```
end
end
end
end
end
%make current set new set and clear new set
%     Dstar.closedset = [Dstar.closedset;Dstar.openset];
%     Dstar.closedset = unique(Dstar.closedset,'rows');
%     Dstar.openset = unique(newcurrentset,'rows');
%      setcount = length(Dstar.openset);
time = toc;
if time > 25
break
end


% heuristic
Dstar.closedset = [Dstar.closedset;Dstar.openset(1:min(setcount
   ,heur),1:2)];
Dstar.closedset = unique(Dstar.closedset,'rows');
Dstar.openset = [Dstar.openset(min(setcount+1,heur+1):end,1:2);
   unique(newcurrentset,'rows')];
Dstar.openset = unique(Dstar.openset,'rows');
distance = zeros(1);
tempopen = zeros(1,2);

if length(Dstar.openset) == 2
break
```

```
end

for z = 1:length(Dstar.openset)
distance(z) = abs(Dstar.start(1)-Dstar.openset(z,1))+abs(Dstar.
   start(2)-Dstar.openset(z,2));
end

[tempsort,Index] = sort(distance,2);
for z = 1:length(Index);
tempopen(z,1) =  Dstar.openset(Index(z),1);
tempopen(z,2) =  Dstar.openset(Index(z),2);
end
Dstar.openset = tempopen;
setcount = length(Dstar.openset);

newsetcount = 1;

% check if path has been found to robot
%if so break from the loop
for n = Dstar.start(1)-1: Dstar.start(1)+1
for p =  Dstar.start(2)-1: Dstar.start(2)+1
if abs(n-Dstar.start(1))+abs(p-Dstar.start(2)) <=1
if n ~=Dstar.start(1) || p ~= Dstar.start(2)
if n > 0 && p > 0 && n <= planesize && p <=planesize && Dstar.
   tags(n,p) ~= 'n' && Dstar.tags(n,p) ~= 'R' && Dstar.tags(n,p
   ) ~= 'O'
Dstar.pointer(Dstar.start(1),Dstar.start(2),1) = n;
Dstar.pointer(Dstar.start(1),Dstar.start(2),2) = p;
```

```
pathfound=1;
Dstar.cost(Dstar.start(1),Dstar.start(2)) = Dstar.cost(n,p)+1;
Dstarmovinggoal.parentblock(Dstar.start(1),Dstar.start(2)) =
   Dstarmovinggoal.parentblock(n,p);
oldcost = Dstar.cost(n,p)+1;
break
end
end
end
end

end




end
Dstar.closedset = [Dstar.closedset;Dstar.openset];
Dstar.closedset = unique(Dstar.closedset,'rows');

Dstarmovinggoal.recalc = unique(Dstarmovinggoal.recalc,'rows');
Dstarmovinggoal.recalccount = length(Dstarmovinggoal.recalc);
Dstar.robottracky = Dstar.start(1);
Dstar.robottrackx = Dstar.start(2);
Dstar.robottrackxold = Dstar.robottrackx;
clear newcurrentset;

end
```

Dstarpathfindwpblock.m

APPENDIX C

Replanning Code for moving target algorithm

```
function [ Dstar,Dstarmovinggoal, count ] = Dstarreplangoalmove
   ( Dstar, Dstarmovinggoal, count,planesize )

oldgoal(1) =Dstar.goal(1);
oldgoal(2) =Dstar.goal(2);

%find whereDstar.goal is
for n = oldgoal(1)-1:oldgoal(1)+1
for p = oldgoal(2)-1:oldgoal(2)+1
if n > 0 && p >0 && n < planesize+1 && p < planesize+1
if Dstar.tags(n,p) == 'G'
Dstar.goal(1) = n;
Dstar.goal(2) = p;
Dstar.pointer(n,p,1) = n;
Dstar.pointer(n,p,2) = p;

end
end
end
end

check =0;
pblocksfound =0;
z=0;
```

```
%&& n <= planesize && p<= planesize && Dstar.tags(n,p) ~= 'r'

while pblocksfound ==0
for n =Dstar.goal(1)-z:Dstar.goal(1)+z
for p =Dstar.goal(2)-z:Dstar.goal(2)+z
if n > 0 && p >0 && n < planesize+1 && p < planesize+1
for t = 1:Dstarmovinggoal.pblockcount
if Dstarmovinggoal.pblock(t,1) == n && Dstarmovinggoal.pblock(t
   ,2) == p && Dstar.tags(n,p) ~= 'Y' %&& Dstar.tags(n,p) ~= 'G
   '
Dstarmovinggoal.pblockcost(Dstarmovinggoal.parentblock(n,p))=
   abs(n-Dstar.goal(1))+abs(p-Dstar.goal(2))-1;
Dstar.cost(n,p) = abs(n-Dstar.goal(1))+abs(p-Dstar.goal(2));
check = check+1;
Dstar.tags(n,p) = 'Y';
end
if check == Dstarmovinggoal.pblockcount
pblocksfound = 1;
end

end
end
end
end
z = z+1;
end
```

```
Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 'G';
x=1;
oldpblockcount = Dstarmovinggoal.pblockcount;
Dstarmovinggoal.pblockcount=0;
gparentblock = zeros(planesize);
%calculate new parent blocks,
count = count+1;
for n =Dstar.goal(1)-1:Dstar.goal(1)+1
for p =Dstar.goal(2)-1:Dstar.goal(2)+1
if n > 0 && p >0 && n < planesize+1 && p < planesize+1
if abs(n-Dstar.goal(1))+abs(p-Dstar.goal(2)) <=1
if Dstar.tags(n,p) ~= 'G' && Dstar.tags(n,p) ~= 'O'
gparentblock(n,p) = x;
Dstarmovinggoal.pblock(x,1) =n;
Dstarmovinggoal.pblock(x,2) =p;
Dstarmovinggoal.pblockcount = Dstarmovinggoal.pblockcount+1;
Dstar.pointer(n,p,1)=Dstar.goal(1);
Dstar.pointer(n,p,2)=Dstar.goal(2);

x=x+1;
end
end
end
end
end
```

```
g=1;
%if a new parent block is more efficent switch to it
while g <= oldpblockcount;

for n = Dstarmovinggoal.pblock(g,1)-1:Dstarmovinggoal.pblock(g
   ,1)+1
for p = Dstarmovinggoal.pblock(g,2)-1:Dstarmovinggoal.pblock(g
   ,2)+1
if n > 0 && p >0 && n < planesize+1 && p < planesize+1 &&
   Dstarmovinggoal.parentblock(n,p) ~=0;
if abs(n-Dstarmovinggoal.pblock(g,1))+abs(p-Dstarmovinggoal.
   pblock(g,2)) <=1
if Dstar.tags(n,p) == 'Y'

nparentblock(Dstarmovinggoal.parentblock(n,p)) = gparentblock(
   Dstarmovinggoal.pblock(g,1),Dstarmovinggoal.pblock(g,2));
Dstarmovinggoal.pblockcost(Dstarmovinggoal.parentblock(n,p)) =
   Dstar.cost(n,p)-Dstar.cost(Dstarmovinggoal.pblock(g,1),
   Dstarmovinggoal.pblock(g,2));
Dstar.pointer(n,p,1)= Dstarmovinggoal.pblock(g,1);
Dstar.pointer(n,p,2)= Dstarmovinggoal.pblock(g,2);

Dstar.tags(n,p) = 0;

end
end
end
end
```

```
end
g=g+1;
end


Dstar.tags(Dstar.goal(1),Dstar.goal(2)) = 'G';




for g = 1:Dstarmovinggoal.pblockcount;
Dstar.tags(Dstarmovinggoal.pblock(g,1),Dstarmovinggoal.pblock(g
   ,2)) = 'p';
Dstar.cost(Dstarmovinggoal.pblock(g,1),Dstarmovinggoal.pblock(g
   ,2)) = 1;
Dstarmovinggoal.parentblock(Dstarmovinggoal.pblock(g,1),
   Dstarmovinggoal.pblock(g,2)) = gparentblock(Dstarmovinggoal.
   pblock(g,1),Dstarmovinggoal.pblock(g,2));
end


Dstar.closedset = unique(Dstar.closedset,'rows');
%change Dstar.costs based on old parent blocks
%make a Dstar.closedset of all calculated blocks and move
   through it?
if exist('nparentblock')
for z= 1:length(Dstar.closedset)
```

```
if Dstar.tags(Dstar.closedset(z,1),Dstar.closedset(z,2))~='p'
for s = 1:length(nparentblock)
if Dstarmovinggoal.parentblock(Dstar.closedset(z,1),Dstar.
   closedset(z,2)) == s
Dstar.cost(Dstar.closedset(z,1),Dstar.closedset(z,2))=Dstar.
   cost(Dstar.closedset(z,1),Dstar.closedset(z,2))+
   Dstarmovinggoal.pblockcost(s);
Dstarmovinggoal.parentblock(Dstar.closedset(z,1),Dstar.
   closedset(z,2)) = nparentblock(s);
end
end
end

end
end


%update parent blocks
%if on Dstarmovinggoal.recalc Dstar.closedset, Dstarmovinggoal.
   recalc
%also need to Dstarmovinggoal.recalc neighbors if their Dstar.
   cost is higher

for z = 1:Dstarmovinggoal.recalccount-1
if Dstarmovinggoal.recalc(z,1) > 0 && Dstarmovinggoal.recalc(z
   ,2) >0 && Dstarmovinggoal.recalc(z,1) < planesize+1 &&
   Dstarmovinggoal.recalc(z,2) < planesize+1
```

```
Dstar.tags(Dstarmovinggoal.recalc(z,1),Dstarmovinggoal.recalc(z
   ,2)) = 'r';
end
end



checker =1;
oldz =1;
newrecalccount =1;
newrecalc = zeros(1,2);
test =3;

%fix this somehow
strun =0;
if Dstarmovinggoal.recalc(1,1) ~=0;
while checker == 1


Dstarmovinggoal.recalccount = size(Dstarmovinggoal.recalc,1);
z = Dstarmovinggoal.recalccount;
toldz = z;
checker =0;

for z = oldz:size(Dstarmovinggoal.recalc,1)
%check if any neighbors need to be Dstarmovinggoal.recalculated
    and mark them
```

```
if Dstar.tags(Dstarmovinggoal.recalc(z,1),Dstarmovinggoal.
   recalc(z,2)) == 'r' && Dstar.cost(Dstarmovinggoal.recalc(z
   ,1),Dstarmovinggoal.recalc(z,2)) < (Dstar.cost(Dstar.
   robottracky,Dstar.robottrackx)+test)
count=count+1 ;
for n = Dstarmovinggoal.recalc(z,1)-1:Dstarmovinggoal.recalc(z
   ,1)+1
for p = Dstarmovinggoal.recalc(z,2)-1:Dstarmovinggoal.recalc(z
   ,2)+1
if abs(n-Dstarmovinggoal.recalc(z,1))+abs(p-Dstarmovinggoal.
   recalc(z,2)) <=1
if n > 0 && p >0 && n ~= planesize+1 && p ~=planesize+1 && (n
   ~=Dstarmovinggoal.recalc(z,1) && p ~= Dstarmovinggoal.recalc
   (z,2))
if (Dstar.cost(n,p) > Dstar.cost(Dstarmovinggoal.recalc(z,1),
   Dstarmovinggoal.recalc(z,2))+1) && Dstar.tags(n,p) ~= 'n'

Dstar.cost(n,p) = Dstar.cost(Dstarmovinggoal.recalc(z,1),
   Dstarmovinggoal.recalc(z,2)) +1;
Dstar.pointer(n,p,1) =Dstarmovinggoal.recalc(z,1);
Dstar.pointer(n,p,2) =Dstarmovinggoal.recalc(z,2);

Dstarmovinggoal.recalccount = Dstarmovinggoal.recalccount+1;
Dstarmovinggoal.recalc(Dstarmovinggoal.recalccount,1) = n;
Dstarmovinggoal.recalc(Dstarmovinggoal.recalccount,2) = p;

Dstarmovinggoal.parentblock(n,p) = Dstarmovinggoal.parentblock(
   Dstarmovinggoal.recalc(z,1),Dstarmovinggoal.recalc(z,2));
```

```
checker = 1;

if Dstar.tags(n,p) == 0
Dstar.tags(n,p) = 'r';
end
end

end
end

end

end
Dstar.tags(Dstarmovinggoal.recalc(z,1),Dstarmovinggoal.recalc(z
   ,2)) = 0;
end
z=z-1;

end
Dstarmovinggoal.recalccount = size(Dstarmovinggoal.recalc,1);
oldz = toldz;
Dstarmovinggoal.recalc = unique(Dstarmovinggoal.recalc,'rows');
end

for z = 1:Dstarmovinggoal.recalccount
count = count+1;
for n = Dstarmovinggoal.recalc(z,1)-1:Dstarmovinggoal.recalc(z
   ,1)+1
```

```
for p = Dstarmovinggoal.recalc(z,2)-1:Dstarmovinggoal.recalc(z
   ,2)+1
if abs(n-Dstarmovinggoal.recalc(z,1))+abs(p-Dstarmovinggoal.
   recalc(z,2)) <=1
if n > 0 && p >0 && n ~= planesize+1 && p ~=planesize+1
if Dstarmovinggoal.parentblock(n,p) ~= Dstarmovinggoal.
   parentblock(Dstarmovinggoal.recalc(z,1),Dstarmovinggoal.
   recalc(z,2)) && Dstarmovinggoal.parentblock(n,p)~= 0
%Dstar.tags(n,p) ~= 'n' && Dstar.tags(n,p) ~= 'R' && Dstar.tags
   (n,p) ~= 'O'
newrecalc(newrecalccount,1) = Dstarmovinggoal.recalc(z,1);
newrecalc(newrecalccount,2) = Dstarmovinggoal.recalc(z,2);
newrecalccount = newrecalccount+1;
newrecalc(newrecalccount,1) = n;
newrecalc(newrecalccount,2) =p;
newrecalccount = newrecalccount+1;
end
end
end
end
end
end


Dstarmovinggoal.recalc = unique(newrecalc,'rows');
Dstarmovinggoal.recalccount = size(Dstarmovinggoal.recalc,1);

count;
```

```
end


end
```

Dstarreplangoalmove.m

APPENDIX D

Path Finding Code for optimal algorithm

```
function [ Dstar, count,size,heur,pathfound ] = Dstarpathfind(
   Dstar, count,size,heur)
pathfound =0;
heur = 20000000000;
setcount=1;
% calc Dstar.cost around Dstar.goal
%set parrent Dstar.pointers
x=1;
for n = Dstar.goal(1)-1:Dstar.goal(1)+1
for p = Dstar.goal(2)-1:Dstar.goal(2)+1
if n > 0 && p >0 && n < size+1 && p < size+1
if abs(n-Dstar.goal(1))+abs(p-Dstar.goal(2)) <=1
if Dstar.tags(n,p) ~= 'O' && (n ~= Dstar.goal(1) || p ~= Dstar.
   goal(2));
if Dstar.tags(n,p) == 'n' && Dstar.tags(n,p) ~= 'R';
%set Dstar.cost, tag as calculated
Dstar.cost(n,p) = 1;
Dstar.tags(n,p) = 'p';

%point to Dstar.goal
Dstar.pointer(n,p,1) = Dstar.goal(1);
Dstar.pointer(n,p,2) = Dstar.goal(2);
%add to current wavefront and increase setcount
Dstar.openset(setcount,1) = n;
Dstar.openset(setcount,2) = p;
```

```
setcount = setcount+1;
%add block to calculation Dstar.costs
end
end
end
end
end
end
```

```
for n = Dstar.robottracky-1: Dstar.robottracky+1
for p =  Dstar.robottrackx-1: Dstar.robottrackx+1
if abs(n-Dstar.robottracky)+abs(p-Dstar.robottrackx) <=1
if n ~=Dstar.robottracky || p ~= Dstar.robottrackx
if n > 0 && p > 0 && n <= size && p <=size && Dstar.tags(n,p)
   ~= 'n' && Dstar.tags(n,p) ~= 'R' && Dstar.tags(n,p) ~= 'O'
Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,1) = n;
Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,2) = p;
pathfound=1;
Dstar.cost(Dstar.robottracky,Dstar.robottrackx) = Dstar.cost(n,
   p)+1;
oldcost = Dstar.cost(n,p)+1;
break
end
end
end
end
```

```
end

%create a open set to be determined
newcurrentset = zeros(1,2);
newsetcount =1;
%add current open set to list of blocks
Dstar.closedset = Dstar.openset;
setcount = setcount-1;
%find neighbors
%calculate Dstar.cost of neighbors
%repeat untill path is found
%need to add a heuristic here

while pathfound ==0
for m = 1:min(setcount,heur)
for n = Dstar.openset(m,1)-1:Dstar.openset(m,1)+1
for p = Dstar.openset(m,2)-1:Dstar.openset(m,2)+1
if abs(n-Dstar.openset(m,1))+abs(p-Dstar.openset(m,2)) <=1
if n > 0 && p >0 && n <= size && p <= size
if Dstar.tags(n,p) ~= 'O' && (n ~= Dstar.openset(m,1) || p ~=
   Dstar.openset(m,2));
if (Dstar.cost(n,p) > Dstar.cost(Dstar.openset(m,1),Dstar.
   openset(m,2))+1 || Dstar.tags(n,p) == 'n') && Dstar.tags(n,p
   ) ~= 'G';
%set Dstar.pointer to lowest Dstar.cost
%future: add random here if multiple possible Dstar.pointers
Dstar.pointer(n,p,1) = Dstar.openset(m,1);
```

```
Dstar.pointer(n,p,2) = Dstar.openset(m,2);



%add block to new set
newcurrentset(newsetcount,1) =n;
newcurrentset(newsetcount,2) =p;
newsetcount = newsetcount+1;
% tag as calculated and add up Dstarmovinggoal.recalculations
if Dstar.tags(n,p) == 'n'
Dstar.tags(n,p) = 0;
end

% give new Dstar.cost
Dstar.cost(n,p) = Dstar.cost(Dstar.openset(m,1),Dstar.openset(m
   ,2))+1;


end
end
end
end
end
end


end
%make current set new set and clear new set
%     Dstar.closedset = [Dstar.closedset;Dstar.openset];
%     Dstar.closedset = unique(Dstar.closedset,'rows');
%     Dstar.openset = unique(newcurrentset,'rows');
```

```
%       setcount = length(Dstar.openset);


% heuristic
Dstar.closedset = [Dstar.closedset;Dstar.openset(1:min(setcount
   ,heur),1:2)];
Dstar.closedset = unique(Dstar.closedset,'rows');
Dstar.openset = [Dstar.openset(min(setcount+1,heur+1):end,1:2);
   unique(newcurrentset,'rows')];
Dstar.openset = unique(Dstar.openset,'rows');
distance = zeros(1);
tempopen = zeros(1,2);




% check if path has been found to robot
%if so break from the loop
for n = Dstar.robottracky-1: Dstar.robottracky+1
for p =  Dstar.robottrackx-1: Dstar.robottrackx+1
if abs(n-Dstar.robottracky)+abs(p-Dstar.robottrackx) <=1
if n ~=Dstar.robottracky || p ~= Dstar.robottrackx
if n > 0 && p > 0 && n <= size && p <=size && Dstar.tags(n,p)
   ~= 'n' && Dstar.tags(n,p) ~= 'R' && Dstar.tags(n,p) ~= 'O'
Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,1) = n;
Dstar.pointer(Dstar.robottracky,Dstar.robottrackx,2) = p;
pathfound=1;
```

```
Dstar.cost(Dstar.robottracky,Dstar.robottrackx) = Dstar.cost(n,
   p)+1;
oldcost = Dstar.cost(n,p)+1;
break
end
end
end
end

end

if length(Dstar.openset) == 2
break
end

for z = 1:length(Dstar.openset)
distance(z) = abs(Dstar.robottracky-Dstar.openset(z,1))+abs(
   Dstar.robottrackx-Dstar.openset(z,2));
end

[tempsort,Index] = sort(distance,2);
for z = 1:length(Index);
tempopen(z,1) =  Dstar.openset(Index(z),1);
tempopen(z,2) =  Dstar.openset(Index(z),2);
end
Dstar.openset = tempopen;
setcount = length(Dstar.openset);
clear newcurrentset;
```

```
newsetcount = 1;


end
Dstar.closedset = [Dstar.closedset;Dstar.openset];
Dstar.closedset = unique(Dstar.closedset,'rows');

Dstar.robottracky = Dstar.robottracky;
Dstar.robottrackx = Dstar.robottrackx;
Dstar.robottrackxold = Dstar.robottrackx;



end
```

Dstarpathfind.m

BIBLIOGRAPHY

[1] X. Sun, W. Yeoh, and S. Koenig, "Moving target d* lite," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2010, pp. 67–74.

[2] J. N. Eagle, "The optimal search for a moving target when the search path is constrained," *Operations research*, vol. 32, no. 5, pp. 1107–1115, 1984.

[3] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 127–132.

[4] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, "Autonomous mobile robots," *Massachusetts Institute of Technology*, 2004.

[5] T. Ishida and R. E. Korf, "Moving target search." in *IJCAI*, vol. 91, 1991, pp. 204–210.

[6] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of game development*, vol. 1, no. 1, pp. 7–28, 2004.

[7] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, p. 27, 1995.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[9] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of a," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 505–536, 1985.

[10] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*. IEEE, 1994, pp. 3310–3317.

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[12] J. P. Heuristics, "Intelligent search strategies for computer problem solving addison," 1984.

[13] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The international journal of robotics research*, vol. 5, no. 1, pp. 90–98, 1986.

[14] O. Souissi, R. Benatitallah, D. Duvivier, A. Artiba, N. Belanger, and P. Feyzeau, "Path planning: A 2013 survey," in *Industrial Engineering and Systems Management (IESM), Proceedings of 2013 International Conference on.* IEEE, 2013, pp. 1–8.

[15] A. Stentz *et al.*, "The focussed d^* algorithm for real-time replanning," in *IJCAI*, vol. 95, 1995, pp. 1652–1659.

[16] S. Koenig and M. Likhachev, "D^* lite," in *AAAI/IAAI*, 2002, pp. 476–483.

[17] F.-M. Adolf and F. Andert, "Rapid multi-query path planning for a vertical take-off and landing unmanned aerial vehicle," *Journal of Aerospace Computing, Information, and Communication*, vol. 8, no. 11, pp. 310–327, 2011.

[18] J. Hagelbäck, "Multi-agent potential field based architectures for real-time strategy game bots," Ph.D. dissertation, Blekinge Institute of Technology, 2012.

[19] B. Anguelov, "Video game pathfinding and improvements to discrete search on grid-based maps," Ph.D. dissertation, University of Pretoria, 2011.

[20] S. Nutanong and H. Samet, "Memory-efficient algorithms for spatial network queries," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on.* IEEE, 2013, pp. 649–660.