Sheffield Hallam University

SHEFFIELD HALLAM UNIVERSITY

SCHOOL OF ENGINEERING

Implementation of Path Finding Techniques in Homeland Security Robots

BY

Anand Veeraswamy

MSc in COMPUTER AND NETWORK ENGINEERING

FULL TIME
2004-2005

Sheffield Hallam University

PREFACE

This report describes project work carried out in the school of engineering at Sheffield Hallam University between June 2005 and September 2005.

The submission of the report is in accordance with the requirements for the award of the degree of **MSc in computer and Network Engineering**, under the auspices of the University.

ACKNOWLEDGMENT

I consider it an honour to present this thesis as a student of MSc in Computer and Network Engineering of the School Of Engineering, Sheffield Hallam University, Sheffield, UK. I express my heartfelt gratitude to the school for giving me this opportunity.

I express my special thanks to:

- Dr Bala Amavasai who has been my inspiration for this project. He was like a guiding light in an ocean of darkness. He was always there to help me out whenever I knocked on his door despite his busy schedule.
- Mr. Jonathan Klein, the founder of Breve, who took the time to clear my doubts, from the simplest to the most involved, promptly and efficiently. He is responsible for the speed in which I became familiar with the Breve environment.
- My mother, my first teacher in life and the best I have ever seen and the best I will ever see.
- My father, whose philosophy of life has affected me and plays an important part in my spiritual life.
- My brother, Selvaraj and his wife, Sylvia who supported and encouraged me through my masters. Very special thanks to my brother who is my friend, philosopher and guide.
- Kavin and Shirom, my brother's children aged 2 and 4, the two little diamonds in my life who add a sparkle to my life through their laughter and innocence.
- Shabbir, for his moral support and belief in me.
- My friends, who have encouraged me when I lacked confidence and instilled courage to face the world when I lacked it. Special thanks go to my few select friends who I consider to be my life long mates.
- Above all to GOD, who works through me as he does through all of us!

ABSTRACT

Path finding techniques is a vital sub task in mobile robotics that has been subjected to extensive research. Navigation issues also assume a lot of importance in homeland security applications. Among the various path finding techniques, the use of search algorithms is a very promising area for research. Numerous search algorithms exist. Using the Breve simulation environment, first the general search algorithm and then the A* algorithm has been implemented. An improvement to the A* algorithm has been implemented. In this thesis it has been proved through experimental results that the performance of the A* algorithm improves drastically after adding an additional heuristic.

Dynamic path finding techniques is an extension of normal path finding. It can be quite challenging for a vehicle which already has set forth on a pre calculated safe path to keep abreast of the changing environment and recalculate the safe paths during the course of its movement. Dynamic path planning has been implemented in this project with the vehicle checking for changes in the environment at every simulation time step and recalculating paths if there is a change in the environment.

Ad-hoc sensor networks are used in homeland security. Ad-hoc sensor networks have been modelled using the patch class in Breve and the path finding techniques have been used in this environment. The path finding technique developed in this project has been found to calculate safe paths for any given start node and target node and for any given configuration of sensor (obstacle) placement. Time analysis of the various algorithms implemented in this project has been presented. Homeland security has also been discussed briefly.

TABLE OF CONTENTS

List of figures

Chapter 1:

Chapter 2:

Chapter 3:

Chapter 4:

Chapter 5:

Chapter 1

Introduction

1.1 Introduction

The aim of the thesis is investigate and develop path finding techniques for autonomous homeland security robots. The following issues will be addressed:

- enable robots to take the safest path to destination
- implement dynamic path finding techniques
- investigate existing path planning methods
- compare the performance of the different methods
- study the Breve simulation environment
- study the Steve language used in Breve
- design and create a model robot vehicle
- teach the robot to navigate in the artificial world
- use the patch class in Steve to replace the sensors used in [1].

1.2 Background

The problem of dynamic collision free path planning is vital to mobile robots and robots used in homeland applications. An attempt was made to create more versatile information systems by using adaptive distributed sensor networks [1]. Hundreds of small sensors, equipped with limited memory and multiple sensing capabilities which autonomously organize and reorganise themselves as ad hoc networks in response to task requirements and to triggers from the environment. A collection of active sensor networks can follow the movement of a source to be tracked, for example, a moving vehicle. It can guide the movement of an object on the ground, for example, a surveillance robot. Or it can focus attention over a specific area, for example, a fire in order to localize its source and track its spread.

Path finding techniques can be grouped into local methods and global methods [2]. One well known local planning method is the potential field method. In this method the robot follows the gradient of a force field. The field is generated by attractive potentials from a start position towards a target and by repulsive potentials that point away from obstacles. The potential field method has a low computational load. In

contrast, the global methods need complete information about the world and hence would require relatively large computational load.

1.3 Motivation

Homeland robotics has assumed a great importance in the present age. Robots are now being used extensively to rescue survivors from hazardous environments. For e.g. when there is a fire robots can be made to go through the fire and rescue. A serious limitation is the lack of sensors that can be mounted on the robots. In 1999, both the American Association for Artificial Intelligence and the RoboCup Federation for Artificial Intelligence and the RoboCup Federation started rescue robot competitions to foster research in this humanitarian application of mobile robots. But good theory does not necessarily lead to good practice. None of the algorithms demonstrated by CRASAR or other groups at various rescue robot competitions or at related DARPA programs were actually usable on robots that could withstand the rigors of real rubble. As a result all the robots used at the WTC site had to be teleoperated [3].

1.4 Thesis Description

The most basic form of tree/network search algorithm has been implemented which does not have any heuristics and finds the first possible path. Next, the A* search, one of the most well known search algorithms has been implemented. This algorithm evaluates the nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n). Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n[4]. The results of the blind search and the A* search has been compared and studied. An additional heuristic was added and experimental data has been provided that this heuristic improves the efficiency of the A* algorithm drastically.

1.5 Methodology

In Breve sensors can be modelled by using the patch class. In the patch class it is possible to create patches which can sense the presence of obstacles (representing dangerous objects). We can assume that the sensors are as sensitive and efficient as

the patches created in the simulation program. But when put to practical use it may be necessary create very efficient sensors or to compensate for the lack of efficiency of the sensors. Once safe patches and unsafe patches are found we should be able to build maps and find safe paths by keeping as far away from the obstacles as possible.

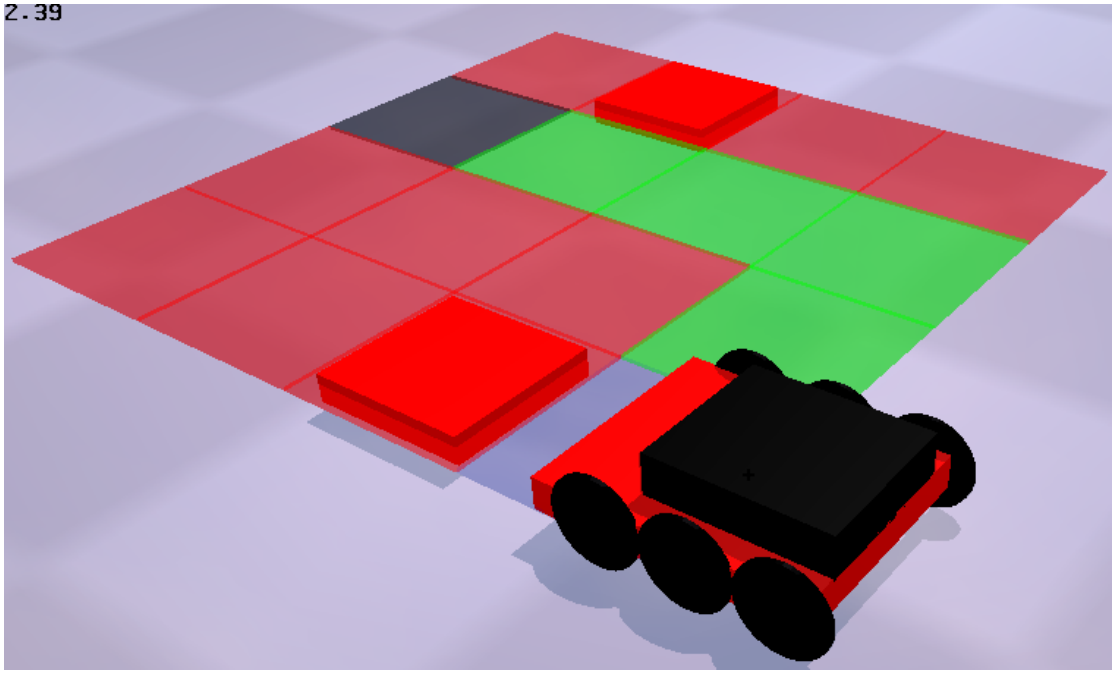

Figure 1.1: Simulation Snapshot. The light blue patch is the starting position of the robot. The dark blue patch is the target. Patches containing the red square blocks represent the danger patches. The empty and pink coloured patches are safe patches. The patches coloured green shows the path to be taken by the vehicle.

1.6 Deliverables

The results obtained in this simulation can be tested and implemented practically using Lego robots. The performance of the various path finding techniques are compared and contrasted. It should be possible to combine the strengths of various algorithms and come with a novel and more efficient algorithm for path planning.

1.7 Thesis Formulation

This section discusses constraints such as time, technical limitations etc. and develops a work plan for the entire thesis.

1.7.1 Time/Schedule

Table 1.1 shows the definitions of the tasks and timescale of the thesis.

Task	**Time Period**
Literature study and investigation of path finding techniques	June 2005
Familiarization of the Breve simulation environment Learn and experiment the Steve language	July 2005
Create a vehicle design in Breve Teach the vehicle to navigate in the artificial world	August 2005
Implementation of path finding techniques	September 1st 2005 to September 15th
Report writing	September 15th 2005 to September 29th 2005

Table 1.1: Tasks to be completed for the thesis and their schedule

1.7.2 Technical Limitations

This section focuses on various limitations of the thesis as listed below

- Difficulty in implementing C, C++ plugins.
- Difficulty in navigating the vehicle.
- Lack of data structures like stacks and queues in Steve.

1.7.3 Potential Hazards

The precautions that were strictly followed during the entire thesis period were

- An erect sitting posture was maintained while working on the computer
- Breaks were taken at regular intervals to avoid cramps and sprains that can be caused due to sitting in front of the computer for long hours.

1.8 Report Guideline

Chapter 2 contains the theory behind path finding which is very essential to this thesis. This chapter discusses most of the theory used in the project. Chapter 3 is on the Breve simulation environment. An overview of the Steve language has also been presented here. Chapter 4 talks about homeland security, why is it so important and why has it suddenly received so much of attention in the research circle. Chapter 5 shows the implementation done in the project. It also shows various experimental data collected in the project. Finally Chapter 6 concludes the thesis work and has some very interesting future work that can be done.

1.9 Summary

This chapter contains the background and motivation behind this project. It also briefly summarises the path finding techniques and homeland security issues.

Chapter 2

Relevant Theory and Analysis

2.1 Introduction

This chapter discusses the basic theory behind path planning and analyses the various methods used in path planning. Path planning using various search algorithms are first analysed. Cell Decomposition and Skeletonisation methods are then discussed. The most popular method of path finding using Potential fields is discussed next. Lastly an interesting method i.e. pathfinding using Pheromones is touched upon briefly.

2.2 Search Algorithms

The output of a search algorithm is either a failure or a solution. The efficiency of a search algorithm can be evaluated in four ways [4]

- **Completeness:** Does the algorithm guarantee to find a solution when it exists
- **Optimality:** Does the strategy find the optimal solution?
- **Time Complexity:** Time taken to find the solution
- **Space Complexity:** Memory needed to perform the search

Search algorithms can be classified as Uninformed and Informed (heuristic) search.

2.2.1 General Search Algorithm

In case of the general search algorithm the agent does not know the full search space. Instead the agent knows the initial state, and it knows operators. An operator is a function which expands a node. "Expanding" a node means computing the node that the agent could move to using the operator. With this available knowledge, the general search algorithm that the agent can use is:

function GENERAL-SEARCH(*problem, stratergy*) returns a solution, or failure
initialise the search tree using the initial state of *problem*
loop do
if there are no candidates for expansion then return failure
choose a leaf node for expansion according to stratergy
if the node contains a goal state then return the corresponding solution
else expand the node and add the resulting nodes to the search tree
end

The Romanian travel example

Imagine using a map to plan a trip from Arad to Bucharest. The Romanian road map is shown in the figure2.1[4].

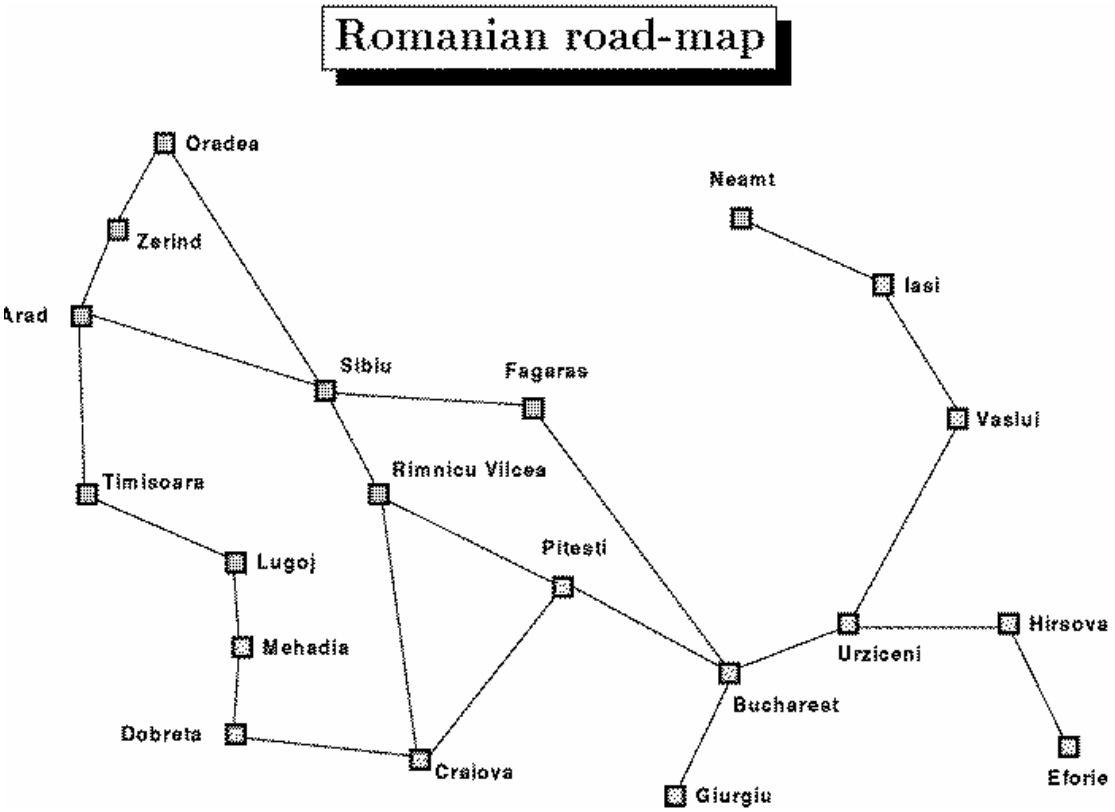

Figure 2.1: Romanian road map

Referring to the figure2.1 the following points can be noted

- Arad is not a goal node so let us expand it to find all its successors: Sibiu, Timisoara, and Zerind.
- Now lets make a choice of one of these to investigate next, say Sibiu.
- Sibiu is not a goal node so let us expand it to get its successors: Arada, Fagaras, Oradea, and Rimnicu Vilcea.
- Now let us make a choice of any one of the current leaf nodes to expand next, i.e. one of Arada, Fagaras, Oradea, Rimnicu Vilcea, Timisoara, and Zerind.
- And so on, until the goal node is arrived.

Figure 2.2 shows this search tree

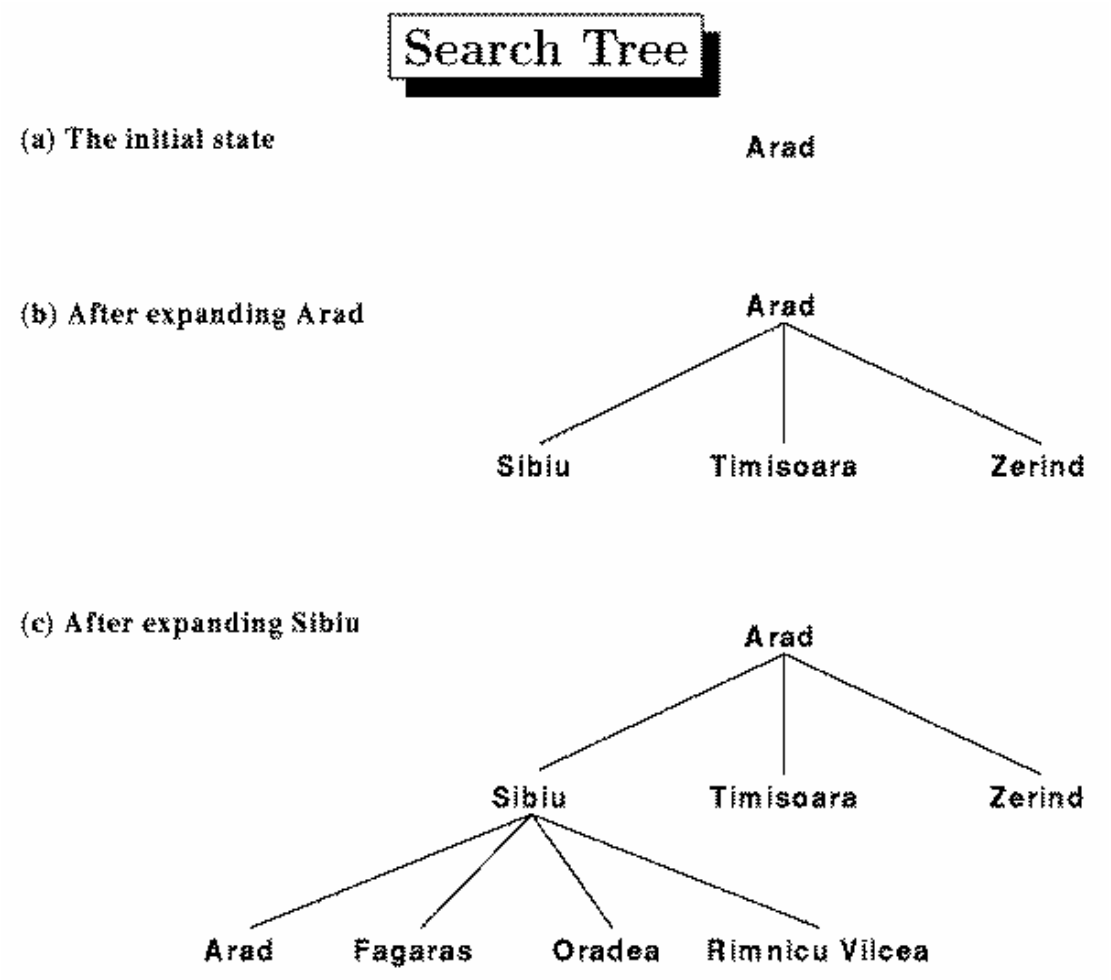

Figure 2.2 Search tree generated after a few iterations of the general search

Here it can be noticed that expanding the node Sibiu gives Arad, which has already been visited. In general the search algorithm does not "know" this.

2.2.2 Uninformed Search Strategies

The uniform search strategies use a blind search technique where each and every possible goal state is analysed. The advantage of this method is that they form a very easy to implement algorithm. The disadvantage is that they are highly inefficient and could result in very high time and space complexities. The Uninformed search strategies are

2.2.2.1 Breadth-first search(BFS)

BFS is the general search algorithm where the "insert" function is "enqueue-at-end". This means that newly generated nodes are added to the fringe at the end, so they are expanded last. BFS first considers all paths of length 1, then all paths of length 2, and so on. This is why it is called "breadth-first". Figure 2.3 shows how BFS works.

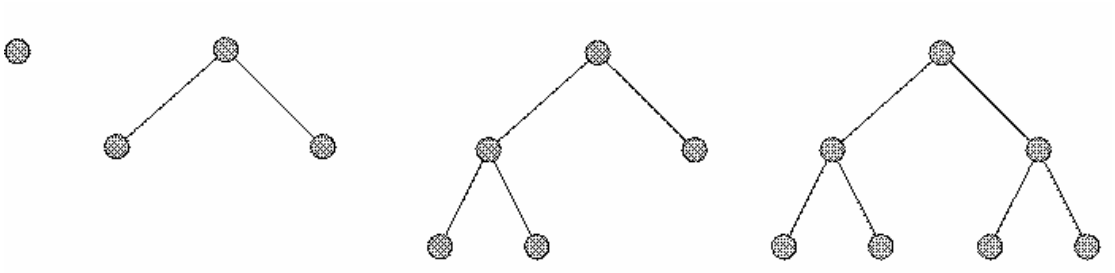

Figure 2.3: Order of expansion of the nodes in the Breadth-first search

2.2.2.2 Depth-first search(DFS)

DFS is the general search algorithm where the "insert" function is "enqueue-at-front". This means that newly generated nodes are added to the fringe at the beginning, so they are expanded immediately [4].

DFS goes down a path until it reaches a node that has no children. Then DFS backtracks and expands a sibling of the node that had no children. If this node has no siblings, then DFS looks for a sibling of the grandparent, and so on. Figure 2.4 illustrates DFS

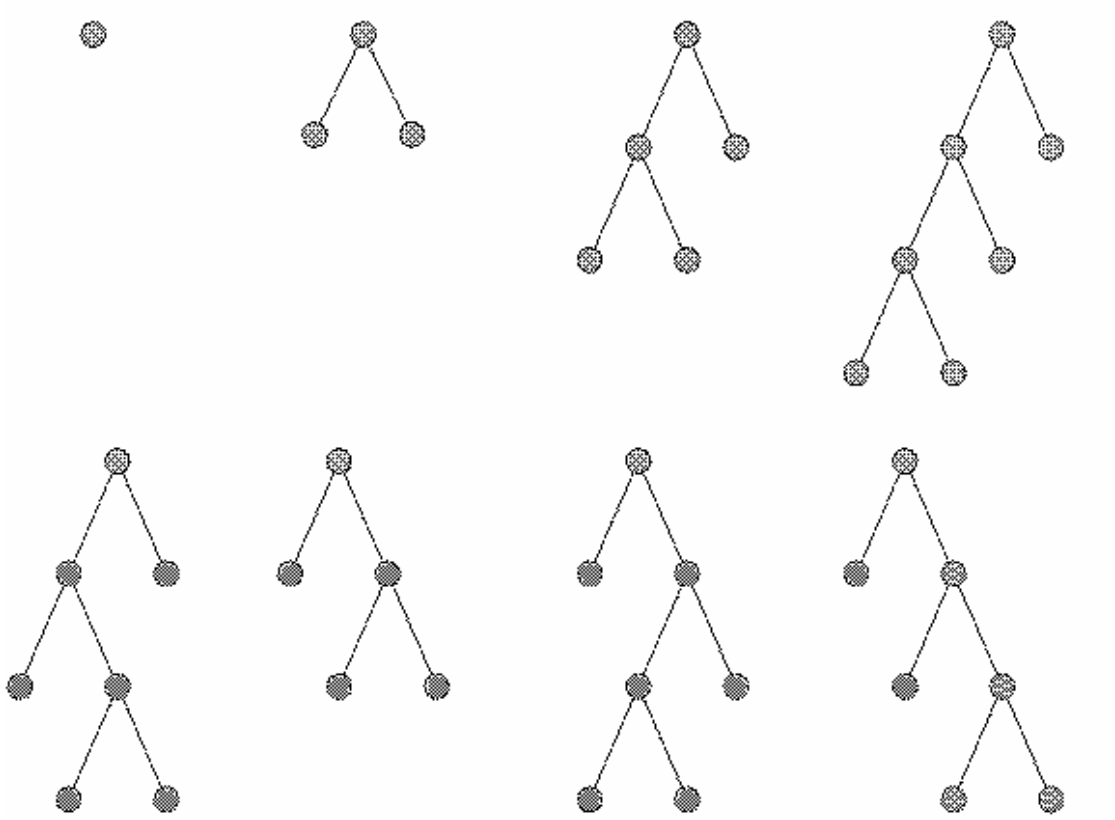

Figure 2.4: Order of expansion of the nodes in Depth First Search(DFS)

2.2.2.3 Iterative deepening depth first search

Iterative deepening is a very simple, very good, but counter-intuitive idea that was not discovered until the mid 1970s. Then it was invented by many people simultaneously. The idea is to do depth-limited DFS repeatedly, with an increasing depth limit, until a solution is found[4].

Intuitively, this is a dubious idea because each repetition of depth-limited DFS will repeat uselessly all the work done by previous repetitions.

Iterative deepening simulates BFS with linear space complexity.

For a problem with branching factor b where the first solution is at depth d, the time complexity of iterative deepening is O(b^d), and its space complexity is O(bd).

2.2.3 Informed (Heuristic) Search strategies

Informed search strategies use problem specific knowledge beyond the definition of the problem itself and can hence find solutions more efficiently than an uninformed strategy. This section explores the various informed search strategies.

Greedy best-first search

The greedy best-first search expands the node that is closest to the destination first. It tries to follow a single path all the way to the goal but will back up when it hits a dead end. It is not optimal and is incomplete. The worst-case time and space complexity is $O(b^m)$, where m is the maximum depth of the search space[4]. Figure 2.5 shows how the greedy search does find a path.

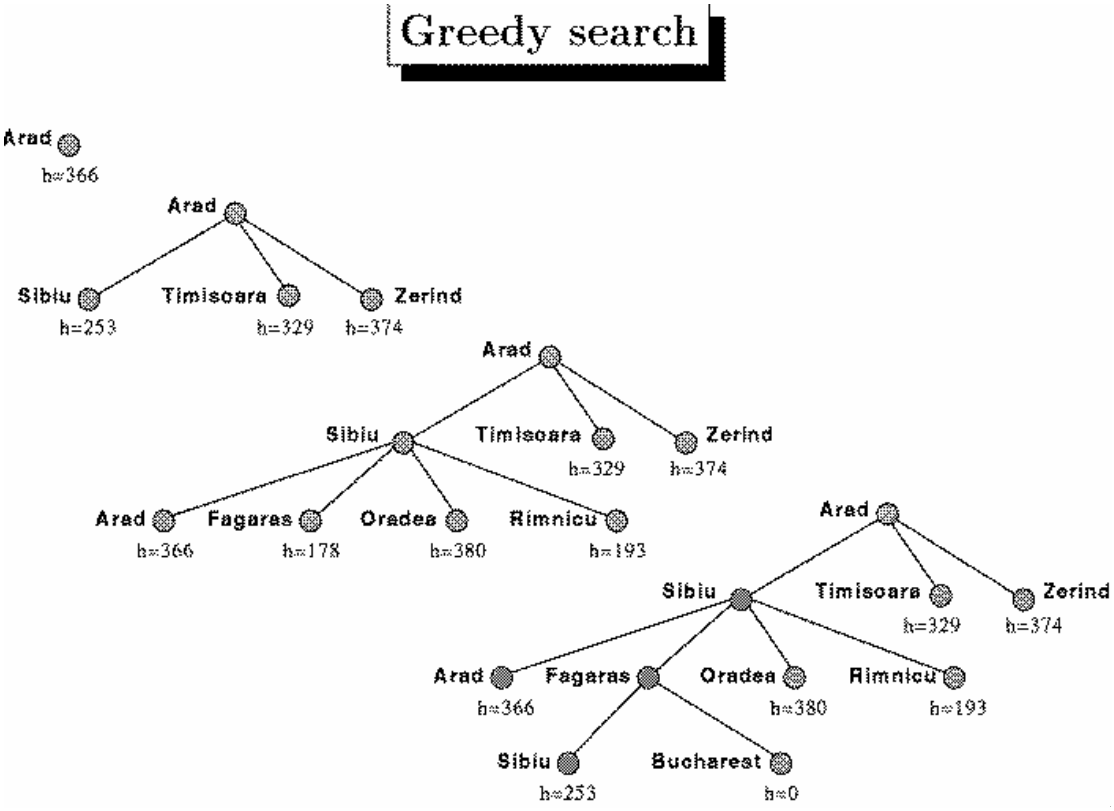

Figure 2.5 Order of expansion in the Greedy Search

A* search

A* is one of the most widely used search techniques. The nodes are evaluated by combining g(n), the cost to reach the goal, and h(n), the cost to get from the node to the goal:

$f(n) = g(n) + h(n)$.

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have

f(n) = estimated cost of the cheapest solution through n .

Hence if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). Provided that the heuristic function h(n) satisfies certain conditions, A* search is both complete and optimal[4]. Figure 2.6 shows how A* search finds an optimal path

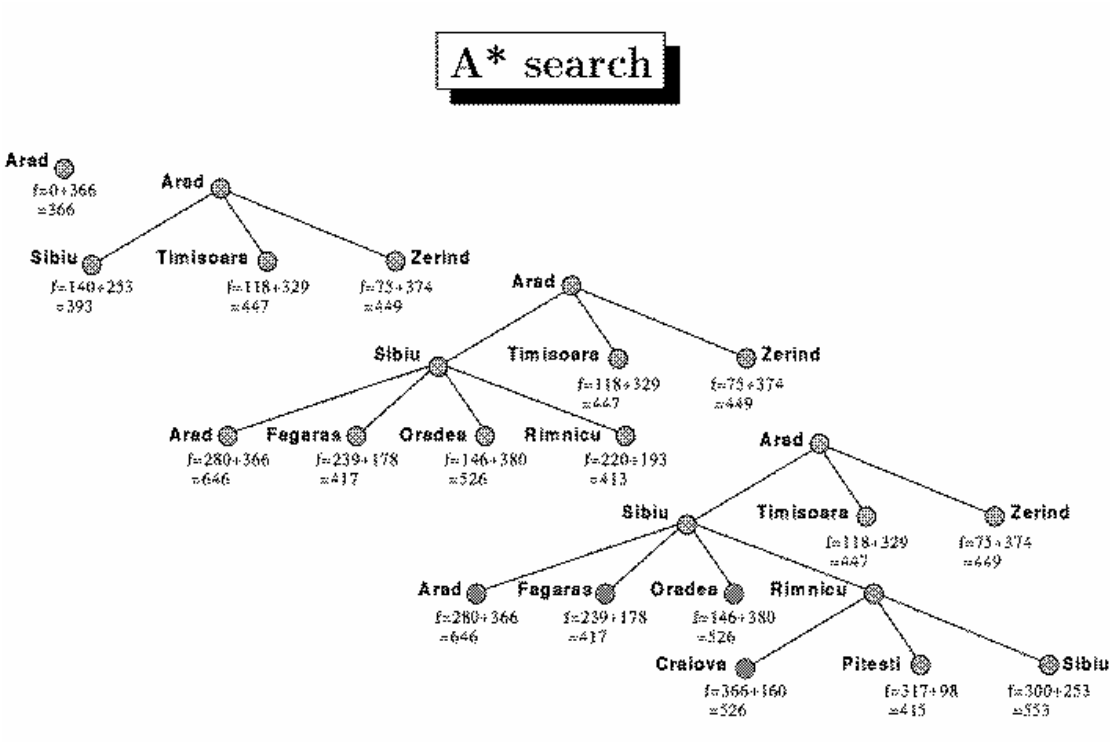

Figure 2.6 Order of expansion in the A* Search

Memory-bounded heuristic search

A simple way of to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A*(IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cut-off used is the f-cost (g+h) rather than the depth. This section briefly examines two more resent memory bounded algorithms

Recursive best-first search(RBFS) is a simple recursive algorithm uses the standard best first strategy using only linear space. RBFS is more efficient than IDA* but still suffers from excessive node re-generation. Like A*, RBFS is an optimal algorithm if the heuristic function h(n) is admissible. Its space complexity is O(bd), but its time complexity is rather difficult to characterize.

Simplified memory-bounded A* proceeds just like A* expanding the best leaf until memory is full. When it finds that it cannot add a new node to the search tree without dropping an old one, SMA* drops the worst leaf node which is the one with the highest f-value. But it backs up the value of the forgotten node to its parent to come back to it if there is no better alternative.

2.3 Cell Decomposition Methods

One approach to path planning is to decompose the entire free space into a finite number of contiguous regions called cells [5]. The simplest cell decomposition consists of a regularly spaced grid. In Breve the patch and patch-grid classes were used to create regularly spaced grid. This method has the advantage that it is very simple to implement. There are 2 disadvantages of this method

- It is workable for low dimensional configuration spaces only, as the number of grid cells increases exponentially with d, the number of dimensions.
- There is a problem of what to do with cells that are mixed i.e. neither entirely within the free space not entirely within occupied space. A solution path includes such a cell may not be a real solution, because there may be no way to cross the cell in the desired direction in a straight line. This would make the path planner unsound. On the other hand, if we insist that only completely free cells may be used, the planner will be incomplete, because it might be the case that the only paths to the goal may go through mixed cells. This problem was encountered in the thesis.

There are 2 solutions to this problem

- Recursively sub dividing the cells until a path is found.
- Exact cell decomposition of the free space. This was the solution used in the thesis. The obstacles were made the same size as the cells and also care was taken to place the obstacles in such a way that they fit the cell exactly without overlapping on the other cells. But an advanced way of solving this issue would be to allow the cells to be irregularly shaped where they meet the boundaries of free space. This technique requires advanced geometric ideas and so was not followed in this thesis.

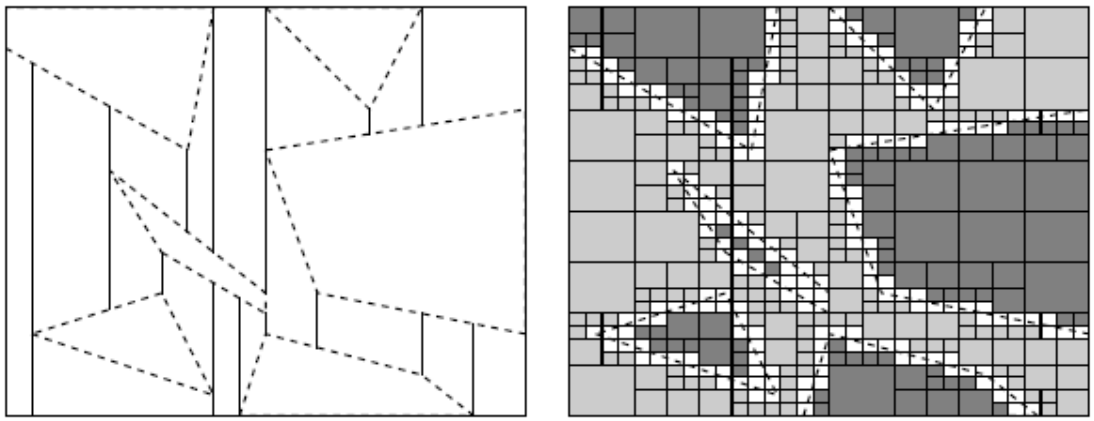

Figure 2.7: Left Figure: Exact cell decomposition, space exactly decomposed into trapezoids. Right Figure: Approximate cell decomposition, mixed cells are divided until a series of free cells connects the start with the goal cell free cells: light grey, obstacle cells: dark grey, mixed cells: white, obstacles (- -).

2.4 Skeletonization Methods

The skeletonization methods reduce the robot's free space to a one-dimensional representation for which the planning problem is easier. This lower-dimensional representation is called a skeleton of the configuration space. This section analyses the two most used methods of skeletonization

2.4.1 Voronoi Graph

Voronoi graph is drawn by connecting the set of all points that are equidistant to two or more obstacles. Path planning using Voronoi graphs involves making the robot move to the closest point in the graph and follows the Voronoi graph until it reaches the point nearest to the target. Disadvantages of Voronoi graph techniques are that they are difficult to be applied to higher dimensional configuration spaces, and that they tend to induce unnecessarily large detours where the configuration space is wide open[6].

2.4.2 Probabilistic Roadmap

Probabilistic roadmap, a skeletonization method offers more possible routes and thus deals better with wide open spaces. The graph is created by randomly generating a large number of configurations, and discarding those that do not fall into free space. We then join any nodes by an arc if it is easy to reach one node from the other .

Theoretically this method is incomplete, because a bad choice of random points may leave us without any paths from the start to the target [7].

2.5 Path Planning Using Potential Fields

In path planning using potential fields, an artificial potential field, where robots move under the actuation of artificial forces is created [1]. The goal generates an attractive potential which pulls the robot towards it. The obstacles generate a repulsive potential which pushes the robot away from the goal. The attractive potential is higher near the goal and lower away from the goal. Hence the robot will naturally move towards the goal from any position. The potential field method has a low computational load and generates smooth paths that stay away from obstacles. However, the greedy gradient descent may get trapped in local minima [3]. It is hence most useful in environments where local minima are unlikely. Furthermore, it can be used for fast reactive obstacle avoidance.

2.6 Path Planning Using Pheromones

Ant colonies are able to quickly adapt to changing food sources and obstacles without the need for centralization [8]. They construct networks of paths with pheromones (evaporative scent markers) that connect their nests with food sources. Mathematically, these networks form minimum spanning trees [9], minimizing the energy ants expend in bringing food into the nest. Agents guided by synthetic pheromones can imitate the behaviour of insects in tasks such as path planning. These systems are well suited to problems such as path planning for unmanned robotic vehicles.

2.7 Summary

A brief overview of the various path planning techniques has been presented in this chapter. Path planning using searching algorithms was discussed first followed by roadmap techniques. The more advanced method of potential fields for finding path was illustrated next. Finally a group path planning method using pheromones was suggested.

Chapter 3

The Breve Simulation Environment

3.1 Introduction

This chapter introduces the Breve simulation environment. The Steve language used in Breve is explored next. The features of the Steve language used in the thesis are discussed. Braitenberg vehicles are discussed briefly. How the robot vehicle used in this thesis evolved using a basic braitenberg vehicle template is illustrated with figures. The patch and patch-grid classes of Steve and their features are mentioned in brief.

3.2 What is Breve

The simulation software Breve, was initiated by Jon klein as a thesis at Hampshire College and was developed further into a Master's thesis at Chalmers University. The software is actively being developed as a platform for a thesis building large scale simulations of evolutionary dynamics, but is also used for many other applications. Breve is a free, open-source software package which makes it easy to build 3D simulations of decentralized systems and artificial life. Users define the behaviours of agents in a 3D world and observe how they interact. Breve includes physical simulation and collision detection so one can simulate realistic creatures and an OpenGL display engine so one can visualize ones simulated worlds. While Breve is conceptually similar to existing packages such as Swarm and StarLogo, the implementation of Breve which simulates both continuous time and continuous 3D space is quite different such that the environment is suited to a different class of simulations[10]. Breve is available for Mac OS X, Linux and Windows. The Breve mailing list was used extensively during the course of this thesis.

3.3 Writing Simulations in Breve

Breve simulations are written in an easy to use language called Steve. The language is object-oriented and borrows many features from languages such as C, Perl and Objective C and even users without previous programming experience will find it easy to program using it[11].

3.4 Use of Plugins in Breve

Breve features an extensible plugin architecture which allows us to write our own plugins and interact with our own code. Writing plugins is simple and allows us to expand Breve to work with existing code. Plugins have been written in Breve to generate MIDI music, download web pages, interact with a Lisp environment and interact with the “push” language [11]. It is also possible to write plugins in C, C++.

3.5 Versions of Breve

The latest version of Breve at the time of writing this report is Breve 2.3. This version includes a genetic algorithm class.

The thesis was initially done using Breve 2.1 and then was upgraded to Breve 2.2 version as soon as it was released.

3.6 Features of Breve

Following are some of the important features of Breve

- Object oriented language: Steve
- An OpenGL display engine
- Collision detection
- Genetic algorithm class
- Braitenberg class
- Push language
- Plugins
- Record movies of the simulation
- Save snapshots of the simulation

3.7 Braitenberg Vehicles

In the book [12], Valentino Braitenberg describes a series of thought experiments in which "vehicles" with simple internal structure behave in unexpectedly complex ways. He describes simple control mechanisms that generate behaviours that, if we did not already know the principles behind the vehicle’s operation, we might call aggression, love, foresight and even optimism. Braitenberg gives this as evidence for the "law of uphill analysis and downhill invention," meaning that it is much more difficult to try to guess internal structure just from the observation of behaviour than it is to create the structure that gives the behaviour.

3.7.1 Braitenberg Vehicle implemented in Breve

The template of the Braitenberg vehicle in Breve is shown is Figure 3.1

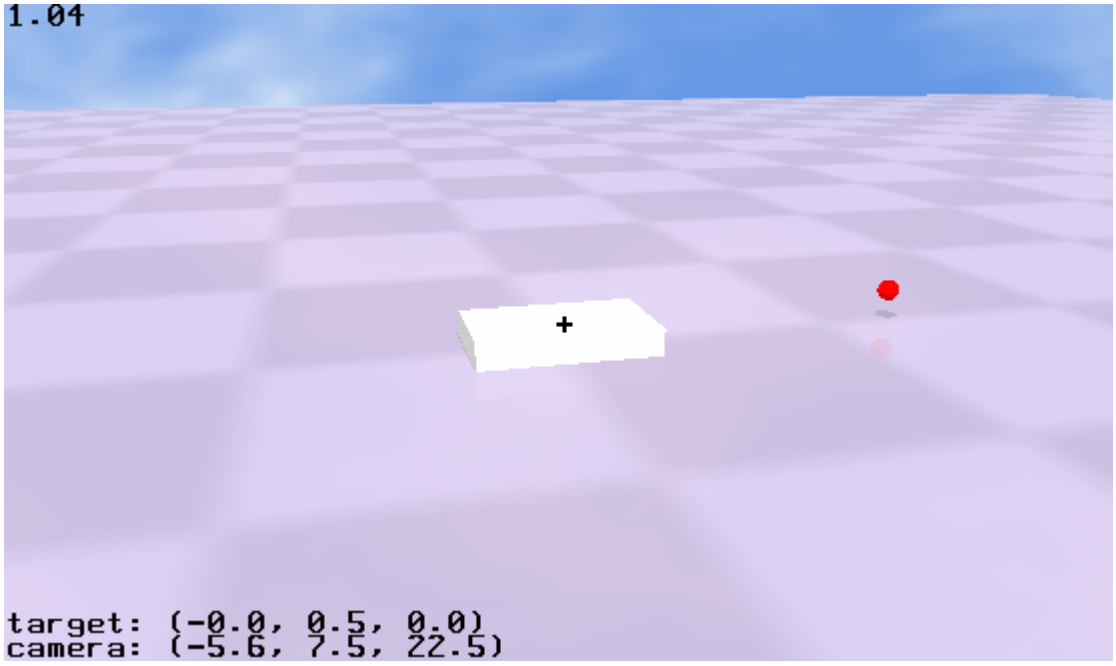

Figure 3.1: Snapshot taken from the Breve simulation shows the rectangular white object which is the body of the vehicle and a red light object which can move about.

This template of the braitenberg vehicle has been taken and the modifications done to it will be shown subsequently.

3.7.2 Features of the Braitenberg Vehicle

There are 5 classes implemented in the Braitenberg class.

BraitenbergControl: In order to create a Braitenberg vehicle simulation, we need to subclass BraitenbergControl and use the init method to create BraitenbergLight and BraitenbergVehicle objects.

BraitenbergLight: A BraitenbergLight is used in conjunction with BraitenbergControl and BraitenbergVehicle. It is what the BraitenbergSensor objects on the BraitenbergVehicle detect.

3.7.3 Braitenberg vehicle with wheels and sensors added

Two wheels were added to the vehicle to enable it to move around. Two sensors were also added to the vehicle to enable it to sense the light objects. Figure 3.2 shows this design.

It was possible to move the vehicle by setting a velocity to the wheels. But the vehicle would move and fall off the edge and it was not possible to study the motion of the vehicle.

The method (vehicle get-position) was used to get the position of the vehicle at each iteration and whenever there was the possibility of the vehicle falling down, it was made to move back and turn right or left.

Now that the vehicle was able to move about the vehicle without falling off the edge, it was time to add sensors and introduce obstacles in the vehicles path.

Using this design an interesting study of the braitenberg principles was observed. Both the left and right sensors were coupled to just one of the wheels. This caused the vehicle to move about the world avoiding the obstacles. But it was noticed that this method was not perfect. The light obstacles would be avoided when the obstacles were either to the left or right of the vehicle. But the vehicle would run right through the obstacle if it were in front of the vehicle equally in between the two sensors. Various combinations of coupling the sensors with the wheels were tried. Adding two more sensors was also tried. But none of the methods could give a prefect obstacle avoidance technique.

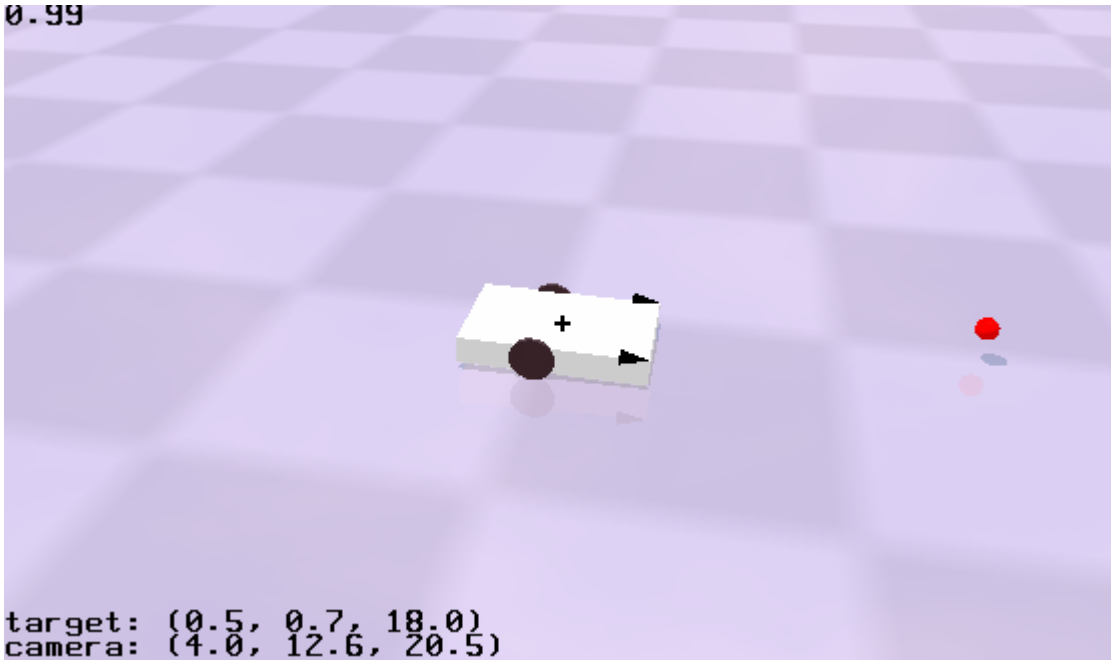

Figure 3.2: Two wheels and two sensors have been added to the vehicle

3.7.4 Introduction of Patches

The introduction of the patches in the simulation led to a change in the dimensions of the braitenberg vehicle. Due to change in the dimensions of the vehicle it was necessary to add more wheels. After the introduction of the patches the use of sensors had become redundant and hence was removed. The final design of the vehicle used is shown in Figure 3.3.

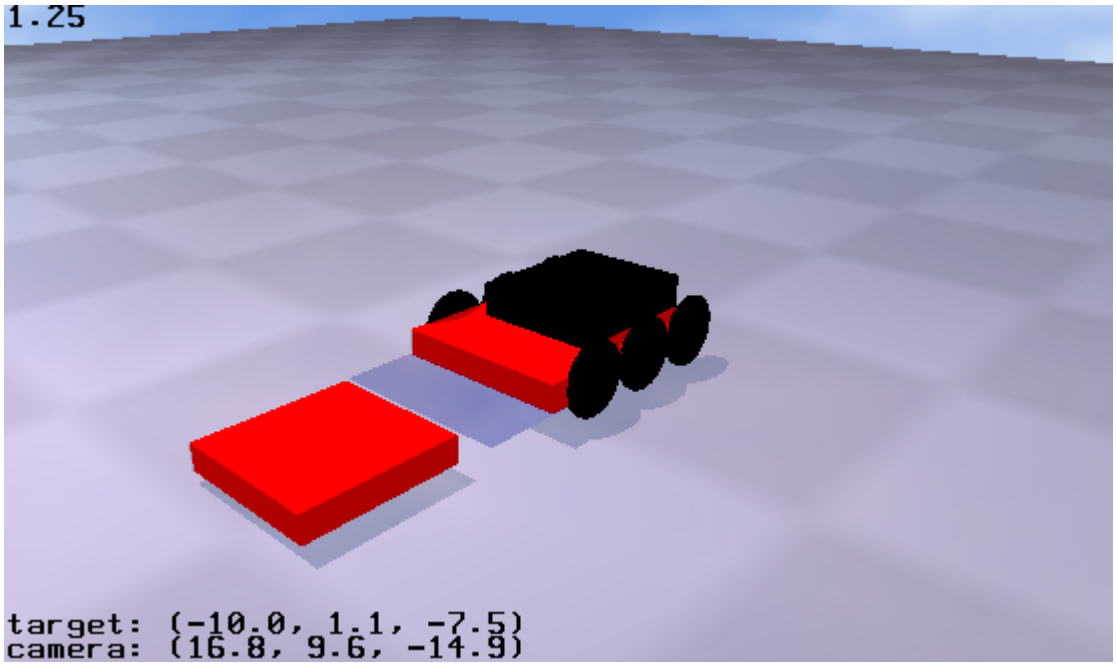

Figure 3.3: Final modified design of the vehicle and obstacle

3.8 Patch Class

In accordance with the cell decomposition method the world was decomposed by equal sized patch grids. The size of the patch grids and the vehicle were made the same in order to ease the navigation of the vehicle from one patch grid to another.
This section discusses the features of the patch class.

3.8.1 Features of the Patch Class

The patch class has a lot of useful features which eased programming task a lot in this thesis.

- patch get-location: returns a vector location of the patch object
- get-patch-at-location: returns the patch at the given vector location
- get-patch-above: Returns the patch towards (0, 1, 0)
- get-patch-below: Returns the patch towards (0, -1, 0)

- get-patch-towards-plus-z: Returns the patch towards (0, 0, 1)
- get-patch-towards-minus-z: Returns the patch towards (0, 0, -1)
- patch set-color to (value) : Sets the color of the patch to value
- patch set-transparency to value : Sets the transparency to value

There are a lot more features in the Patch class but only those above were mentioned since they were used quite often in the thesis.

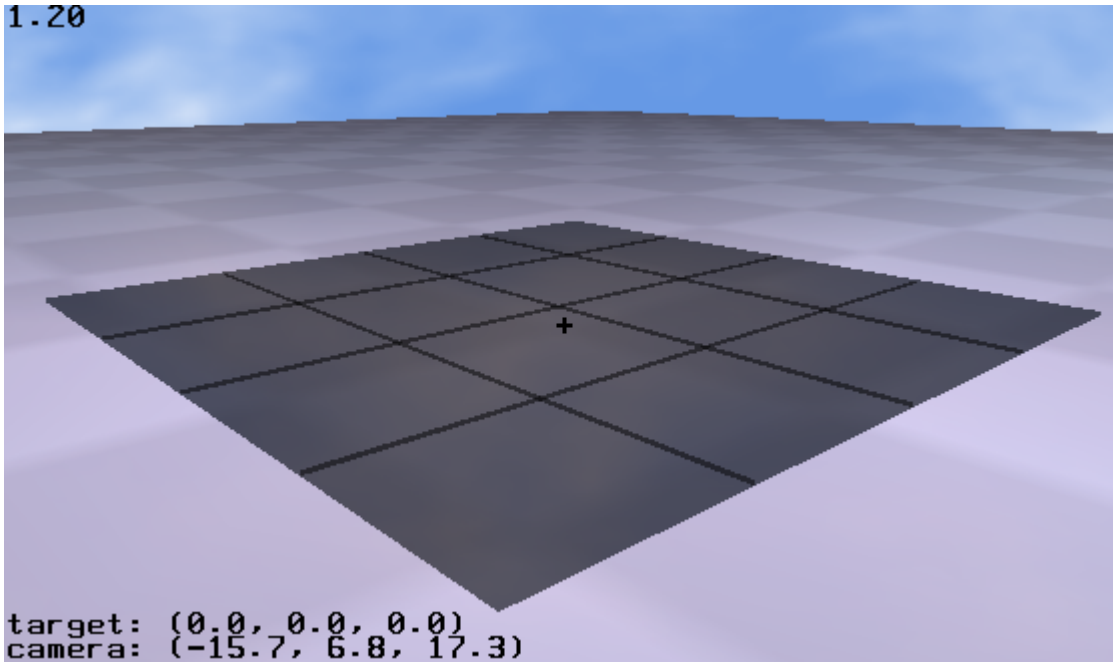

Figure 3.4: Shows a simple 4X4 patch

The following function was used to get the adjacent neighbors of any patch

Figure 3.5 shows the above code in action. Given the patch at the centre(Pink) this function returned the 4 neighbors(Green) surrounding it. The set-color function was used just for demonstration purposes.

```
+to get-neighbors of patch (object):
    neighborList (list).
    i (int).
    i=0.
    if (patch get-patch-to-left) : {neighborList{i}= (patch get-patch-to-left). i++. }
    if (patch get-patch-to-right) : {neighborList{i}= (patch get-patch-to-right). i++. }
    if (patch get-patch-towards-plus-z) : {neighborList{i}= (patch get-patch-
      towards-plus-z). i++. }
    if (patch get-patch-towards-minus-z) : {neighborList{i}= (patch get-patch-
      towards-minus-z). i++. }
    return neighborList.
```

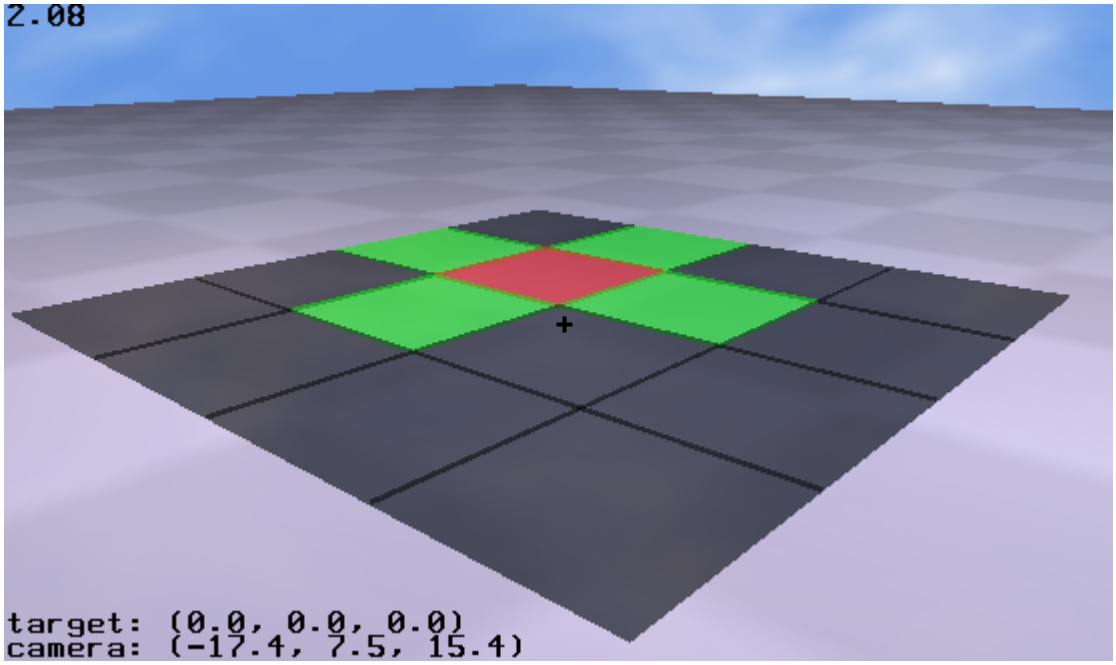

Figure 3.5: Given any patch it is possible to obtain its neighbouring patches.

The following function was used to delete patches which contained an obstacle from the neighbors list. The patch on which the red rectangular object resides on is not coloured green as can be seen in Fig3.6.

```
+ to deleteobs neighList neighList (list) :
        obsList, tempList (list).
        i, j, k, length2 (int).
        patch (object).
        j=0. k=0.
        obsList = self getObstacleList.
        for each patch in neighList: {
                i=0.
                length2 = |obsList| - 1.
                while(length2 >= 0) : {
                   if (patch == obsList{length2} ) : {i=1.}
                    length2--.
                 }
         if(i==0): {
              tempList{j} = neighList{k}.
              j++.
          }
           k++.
        }
        return tempList.
```

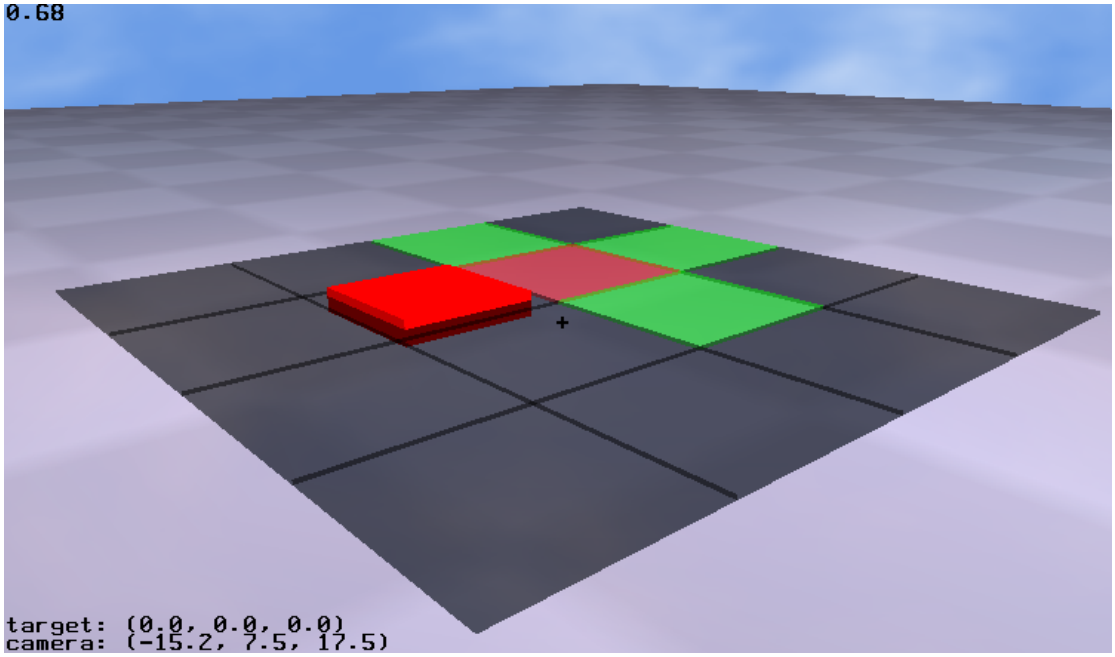

Figure 3.6: The patch containing the obstacle has been deleted from the neighbour list

There are a lot more functions of the patch class used which cannot all be covered here.

3.9 List Data Type

The data type used to store the patches was the list data type in Steve. Using the list data type it is possible to keep a list of other variables of any type including other lists.

3.9.1 List Operators

Some of the list operators used often in the thesis are

- **insert expression at list{ index }:** inserts expression at the specified index in the list, shifting up other list elements with higher indices
- **remove list{ index }:** removes the element of list at the specified index and returns it, shifting down other list elements with higher indices
- **push expression onto list:** appends expression onto the end of list
- **pop list:** removes the last element of list and returns it
- **prepend expression onto list:** prepends expression onto the start of list

- **list{ expression } = value:** sets an element of the list at offset expression to value.
- **`copylist list`:** copies the entire list.
- **`| list |`:** gives the length of a list.

3.10 Summary

The Features of Breve was covered briefly in the beginning of this chapter. The basics of the Steve language was also touched upon. Some important classes that were used in the thesis like Braitenberg class, Patch class was covered and 2 functions used in the patch class was demonstrated. The chapter ended with a brief description of the List datatype which shows the strength of this data type in Steve.

Chapter 4

Homeland Security

4.1 Introduction

This chapter first discusses the basics of homeland security and why it is necessary. The sensor based networks used for homeland applications are discussed next. How the sensor based ad-hoc networks have been simulated using Breve is illustrated with figures. The placement of sensors and algorithms to place sensors are discussed. Finally dynamic path finding where the sensor values keep changing over time which was implemented in this thesis is discussed.

4.2 What is Homeland Security?

Homeland security or homeland defence is a neologism referring to domestic governmental actions justified by potential guerrilla attacks or terrorism. The term became prominent in the United States following the September 11, 2001. Terrorist Attack, although it was used less frequently before that incident.

Such domestic governmental actions include [13]:

- Emergency mobilization, including volunteer medical, police, and fire personnel
- New domestic surveillance and spying efforts, particularly with respect to immigration, transportation, military installations, and utilities
- Infrastructure protection
- Border control

4.3 Scope of Homeland Security

The six main mission areas considered critical for Homeland Security are:

- Intelligence and warning;
- Border and transportation security
- Domestic counterterrorism
- Protecting critical infrastructures
- Defending against terrorism
- Emergency preparedness and response

The first three areas focus on, among other things, preventing terrorist attacks against the U.S., the next two on reducing vulnerabilities within the U.S., and the last area on minimizing the damage and recovering from terrorist attacks that have occurred in the U.S [14].

4.4 Navigation Techniques in Sensor Networks

The application of path finding techniques for homeland applications used in this thesis was inspired by work done in [1]. Here a versatile information by using distributed sensor networks: hundreds of small sensors, equipped with limited memory and multiple sensing capabilities which autonomously organize and reorganize themselves as ad hoc networks in response to task requirements and to triggers from the environment. Distributed adaptive sensor networks are reactive computing systems, well suited for tasks in extreme environments, especially when the environmental model and the task specifications are uncertain and the system has to adapt to them. A collection of active sensor networks can follow the movement of a source to be tracked, for example, a moving vehicle. It can guide the movement of an object on the ground, for example, a surveillance robot. Or it can focus attention over a specific area, for example, a fire in order to localize its source and track its spread. A sensor network consists of a collection of sensors distributed over some area that form an ad hoc network. Each sensor is equipped with some limited memory and processing capabilities, multiple sensing modalities, and communication capabilities. These sensors are capable of detecting special events called “danger” (e.g. temperature, biochemical agents, etc.) that are above a particular threshold. The sensors that have triggered the special events are considered to be obstacles.

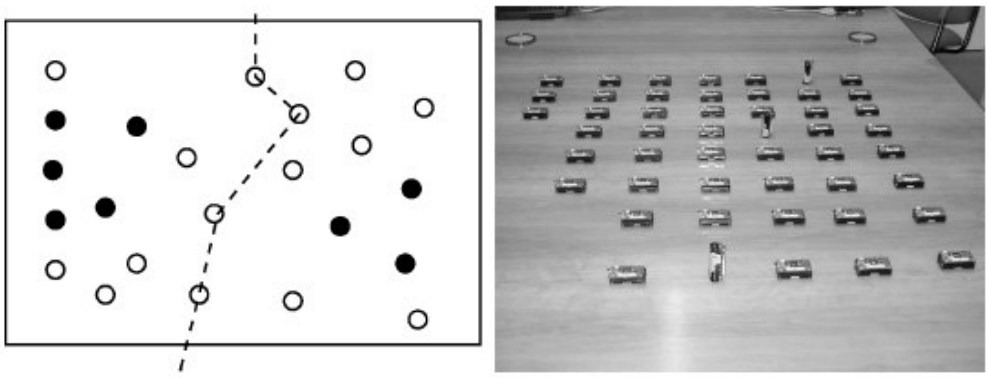

Figure 4.1[15]: The left figure shows a typical setup for the navigation guiding task. The solid black circles correspond to sensors whose sensed value is danger. The white circles correspond to sensors that do not sense danger. The dashed line shows the guiding path across the area covered by the sensor network. Note that the path travels from sensor to sensor and preserves a maximal distance from the danger areas while progressing to the exit area. The right picture shows some Mote sensors used for our experiments. The three sensors placed in the upright position denote two obstacles (i.e., danger areas) and one goal.

4.5 Sensor Network modelled in Breve

The sensor network mentioned in the previous section was modelled in Breve using patches in the patch class. In this thesis the patches was equally distributed over the entire region. But efficiency could have been improved by placing the patches in a certain pattern to minimize the number of patches and maximising the safety of a vehicle navigating through the area infested with “danger” (obstacles). In the sensor network the sensors would sense the special events electronically. This is simulated in Breve by placing light obstacles over the patches. By getting the location of the light object and by finding the patch present at that location we can determine the patches which have obstacles or “danger” and patches which are safe.

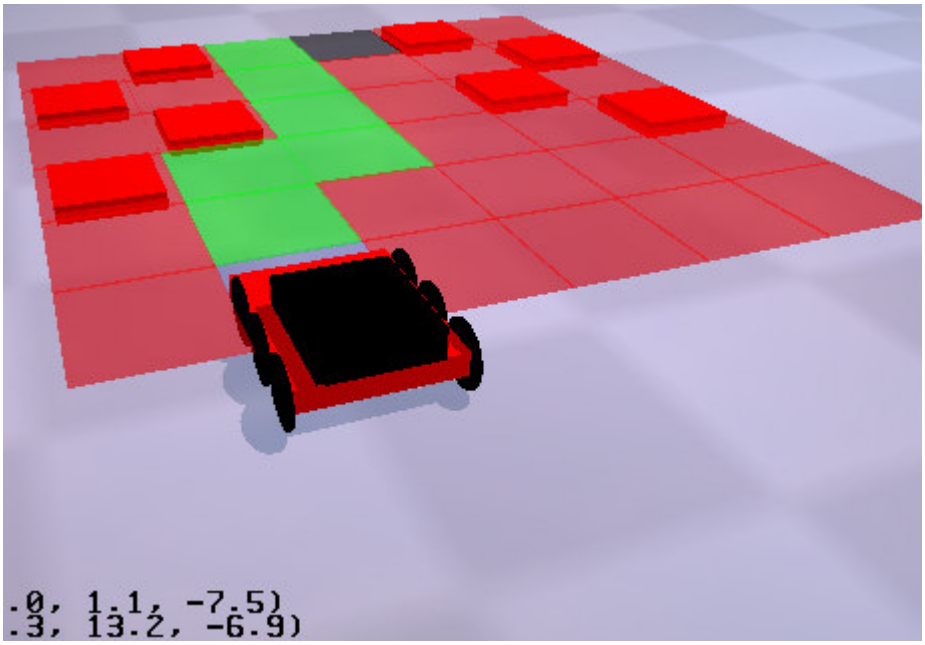

Figure 4.2: The sensor network shown in fig3.1 has been modelled using patches in Breve. The patches containing the red light objects represent danger areas. The patch the vehicle is on is the start patch. The dark blue patch is the target patch. The green coloured patches show the path the vehicle needs to take.

4.6 Sensor Communications

The US Army's Future Combat Systems (FCS) will rely heavily on the use of unattended sensor networks to detect, locate and identify enemy targets in order to survive with less armour protection on the future battlefield. The latest Homeland Security (HLS) counter-terrorist measures will also rely heavily on unattended sensor networks to detect, locate and identify terrorist attacks on critical civilian infrastructure. Successful implementation of these critical communication networks will require the collection of the sensor data, processing and collating it with available intelligence, then transporting it in a format conducive to make quick and accurate decisions. The networked communications must support secure, stealthy, and jam resistant links for sensor data fusion for both tactical and HLS missions[16].

S&TCD is working closely with the U.S. Army Research Laboratory to mature specialized networked communications technologies for the Networked Sensors for the Objective Force (NSOF) Advanced Technology Demonstration (ATD) that is led by the CECOM RDEC Night Vision Electronic Sensors Directorate. The NSOF communications can apply to tactical deployments as well as scenarios for HLS when

normal civilian communications infrastructure are not available due to terrorist attack or stress from the chaotic scenario. Successful use of these critical UGS networks requires the development of complementary communication networks to interconnect the UGS networks within a sensor field and to connect the UGS networks field back to higher level data fusion and Command and Control (C2) elements. The envisioned UGS supporting communications architecture consists of two layers, the Lower Sensor Layer (LSL) and the Higher Sensor Layer (HSL). Figure 4.3 shows a depiction of the NSOF Notional System Architecture.

The LSL consists of, but is not limited to, small, (close to the ground) low data rate sensor nodes known as Pointer Nodes (P-nodes). P-nodes may contain a combination of sensor types, such as acoustic, magnetic and seismic detection sensors, micro-radar motion detectors, and potentially Nuclear, Biological, and Chemical (NBC) sensors. These sensors are integrated with a low power, low data rate radio and associated battery and antenna, which communicate at ranges of up to four hundred meters. P-nodes perform functions such as target detection, simple target classification, and simple Line-of Bearing (LOB) determination and in some cases data fusion. The HSL consists of higher data rate/bandwidth sensor nodes known as Recognition Nodes (R-nodes). R-nodes may contain some of the aforementioned types of sensors, combined with video imagery and/or uncooled InfraRed (IR) imagery capabilities. The R-nodes generally are larger, higher power nodes having advanced processing. capabilities. These nodes may perform data fusion and correlation, image processing, advanced

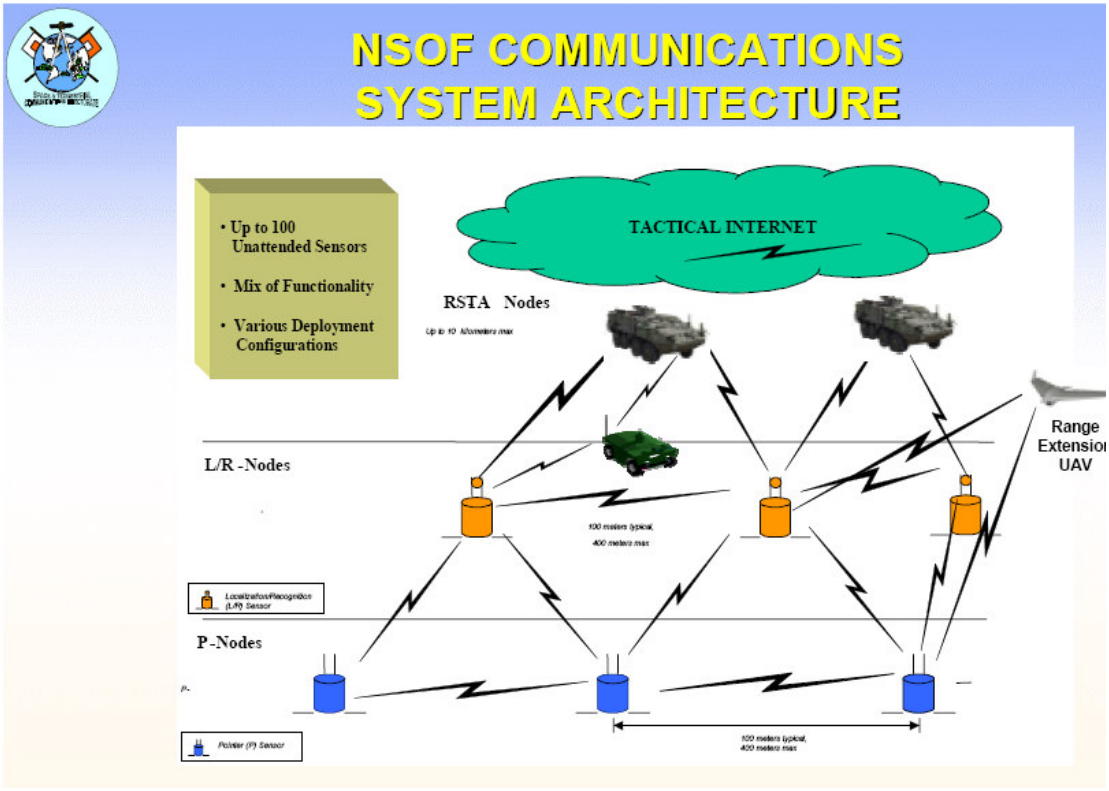

Figure 4.3: Notional NSOF System Architecture

target classification, and act as the gateway that allows the UGS networks to communicate with the upper echelon C2 elements. To facilitate the transfer of imagery data and the longer-range communications requirement (up to 10 Km), a higher power, higher data-rate radio is required, along with the appropriate batteries and antennae. In addition to the HSL functions just identified, the R-node is networked together with the LSL P-node network. For an HLS application the architecture is very similar. The drivers are the quantity of sensor data and the distance range that the data needs to be sent to various decision authorities. The quantity of data will be determined by the type of sensor. Acoustic, magnetic and seismic detectors will have a minimum amount of data while Infra-red or video imagers will require more data. The HLS range needs are somewhat more complex. Numerous civilian decision authorities are involved in the event of an attack on a critical target are numerous. Even when the lead authority is identified, the data must be shared with the other national, regional, state and local agencies which also need the data to coordinate their specialized activities in a timely manner.

4.7 Sensor Deployment Techniques

In this thesis sensors(patches) were placed at a uniform distance and connected to each other. In [17] the sensor deployment problem in the context of uncertainty in sensor locations subsequent to airdropping was considered. Sensor deployment in such scenarios is inherently non-deterministic and there is a certain degree of randomness associated with the location of a sensor in the sensor field.

Wireless sensor networks that are capable of observing the environment, processing the data, and making decisions based on these observations, have recently attracted considerable attention [18]. Such networks can be used to monitor the environment, detect, classify and locate specific events, and track targets over a specific region. The topology of the sensor field, i.e., the locations of the sensors, determines to a large extent the quality of the coverage provided by the sensor network. However, even if the sensor locations are precomputed for optimal coverage and resource utilization,

there are inherent uncertainties in the sensor locations when the sensors are dispersed, scattered, or airdropped. Thus a key challenge in sensor deployment is to determine an uncertainty aware sensor field architecture that reduces cost and provides high coverage, even though the exact location of the sensors may not be controllable. In applications such as battlefield surveillance and environmental monitoring, sensors may be dropped from airplanes. Such sensors cannot be expected to fall exactly at predetermined locations; rather there are regions where there is a high probability of a sensor being actually located (Figure. 4.4). In underwater deployment, sensors may move due to drift or water currents. Thus the position of sensors may not be exactly known and for every point in the sensor field, there is only a certain probability of a sensor being located at that point.

In [17], two algorithms for sensor deployment were presented wherein it was assumed that sensor positions were not exactly predetermined. It was also assumed that the sensor locations were calculated before deployment and an attempt was being made during the airdrop to place sensors at those locations; however, the sensor placement calculations and coverage optimization were based on a Gaussian model, which assumes that if a sensor is intended for a specific point P in the sensor field, its exact location can be anywhere in a “cloud” surrounding P. The sensor field was represented as a grid (two or three dimensional) points. A target in the sensor field is therefore a logical object, which is represented by a set of sensors that see it. An

irregular sensor field is modelled as a collection of grids. The optimization framework is however inherently probabilistic due to the uncertainty associated with sensor

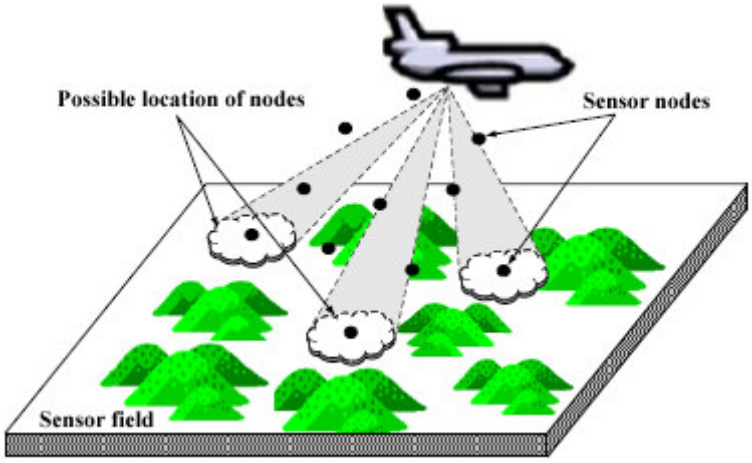

Figure 4.4: Sensors dropped from airplanes. The clouded region gives the possible region of a sensor location. The black circle shows the mean (intended) position of a sensor.

detections. Two algorithms were proposed for sensor placement that addressed coverage optimization under the constraints of imprecise detections and terrain properties. The placement algorithms gave the sensor positions prior to actual placement and we assume that sensors are deployed in a single step.

4.8 Moving Sensors

In a dynamic environment where the sensor values may change from "danger" to "safe" or vice versa, care should be taken to recalculate the path as the vehicle is navigating. Instead of calculating the entire path again, algorithms like the dynamic A* or lifelong A* can be used.

4.9 Summary

This chapter gives a brief introduction to homeland security. The sensor network used in homeland security was illustrated and then a snapshot and explanation of how this network was modelled in Breve is shown. Sensor deployment techniques not used in the project was also discussed. Dynamic path planning was discussed in brief.

Chapter 5
Implementation and Results

5.1 Introduction

The coding involved in the development of the vehicle design used in this thesis is discussed. The patches and light objects modelled as obstacles and some functions using these have been illustrated. The implementation of the general search algorithm, the A* algorithm has been shown along with the pseudocode and experimental results obtained from the thesis. An additional heuristic which was added to the A* algorithm is discussed. Experimental data to prove that this heuristic improves the A* algorithm is provided. Dynamic path finding and path finding techniques used in homeland robotics are then illustrated with snapshots taken from actual simulation performed in the thesis.

5.2 Design and implementation of the Vehicle

The evolution of the design of the vehicle used in this thesis has already been demonstrated in Chapter 3 section 3.7 of this report. In this section the coding involved in the design and implementation of the vehicle will be discussed.

5.2.1 Changes made to the Braitenberg Class

The following piece of code was added to the class file in order to add a black protrusion on the top.

```
+ to add-body at location (vector):
            % Adds a top body at location on the vehicle.

            bodyAdd, joint, addLink (object).

            bodyAdd = new Shape.
            bodyAdd init-with-cube size (4, 1.0, 3.5).

            addLink = new Link.
            addLink set-shape to bodyAdd.

            addLink set-color to (0,0,0).
            joint = new FixedJoint.

            joint link parent bodyLink
                          to-child addLink
                          with-parent-point location
                          with-child-point (0,0,0).

            self add-dependency on joint.
            self add-dependency on bodyAdd.

            return bodyAdd.
```

```
A lot of care had to be taken when creating joints and adding
dependencies to the main vehicle.
```

There are four joints available in Breve. They are

- **PrismaticJoint** for linear sliding joints between links
- **RevoluteJoint** for rotational joints between links
- **FixedJoint** for static joints between links
- **BallJoint** for ball joints between links
- **UniversalJoint** for ball joints between links

For adding a body on top of the vehicle implemented in this thesis a Fixed joint was used as can be seen in the sample code listing.

5.2.2 Coding Used to Implement the Vehicle

A Class implementing the Braitenberg class was created and the following piece of code was added to create the vehicle.

```
% Creates new braitenberg vehicle instance
vehicle = new BraitenbergVehicle.

% Sets the colour of the vehicle to (1.0, 0.0, 0.0)
vehicle set-color to (1.0, 0.0, 0.0).

% Makes the camera focus on the vehicle as it moves about
self watch item vehicle.

%Adds the wheels at the vector position and sets a color
leftbackWheel1 = (vehicle add-wheel at (-2.5, 0, 2.5)).
leftbackWheel1 set-color to (0, 0, 0).
rightbackWheel1 = (vehicle add-wheel at (-2.5, 0, -2.5)).
rightbackWheel1 set-color to (0.0, 0, 0).

leftbackWheel2 = (vehicle add-wheel at (-0.5, 0, 2.5)).
leftbackWheel2 set-color to (0, 0, 0).
rightbackWheel2 = (vehicle add-wheel at (-0.5, 0, -2.5)).
rightbackWheel2 set-color to (0.0, 0, 0).

leftfrontWheel = (vehicle add-wheel at (1.5, 0, 2.5)).
leftfrontWheel set-color to (0, 0, 0).
rightfrontWheel = (vehicle add-wheel at (1.5, 0, -2.5)).
rightfrontWheel set-color to (0.0, 0, 0).

%Makes use of the add-body class the coding of which was shown in the
%previous section.
Addbody = (vehicle add-body at (-1,0.8,0) ).
```

Figure 5.1 shows the final design of the vehicle.

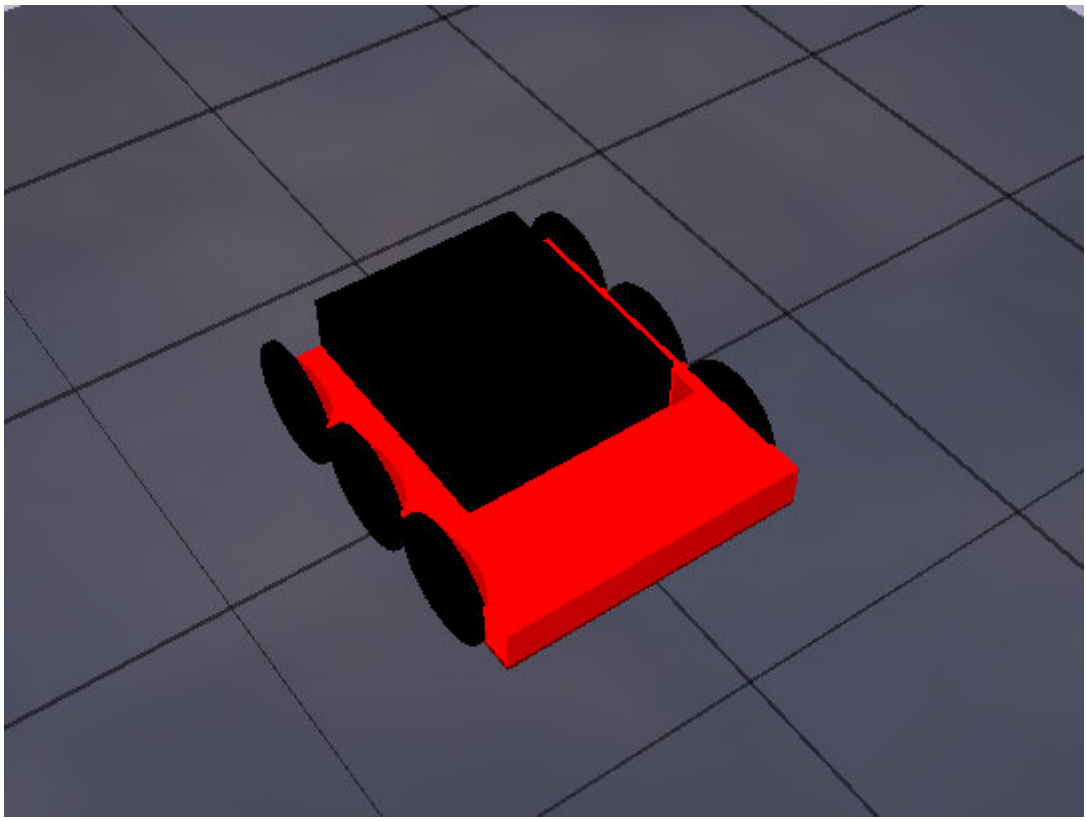

Figure 5.1: Vehicle Design used in the Thesis.

5.3 Patches

According to the principle of Cell Decomposition the entire space was covered by equal size patches. The Patch and PatchGrid classes in Steve was used for this. These patches have a sense of location. That is given a location it is possible to obtain the patch object residing in that location. Given a patch it is possible to get its location. Figure 5.2 shows the patches covering the entire work space. It was not possible to capture the entire work space and hence only a partial view has been shown.

The patches were created using

```
patches = (new PatchGrid init-at location (0,0.75,0) with-patch-size
(5, 0.1, 5) with-x-count X_SIZE with-y-count Y_SIZE with-z-count 6
with-patch-class "LifePatch").
```

By changing the x and z values it was possible to create patches of different dimension like 4X4, 6X6 etc….A 32X32 patch would fill the entire work space created in this simulation.

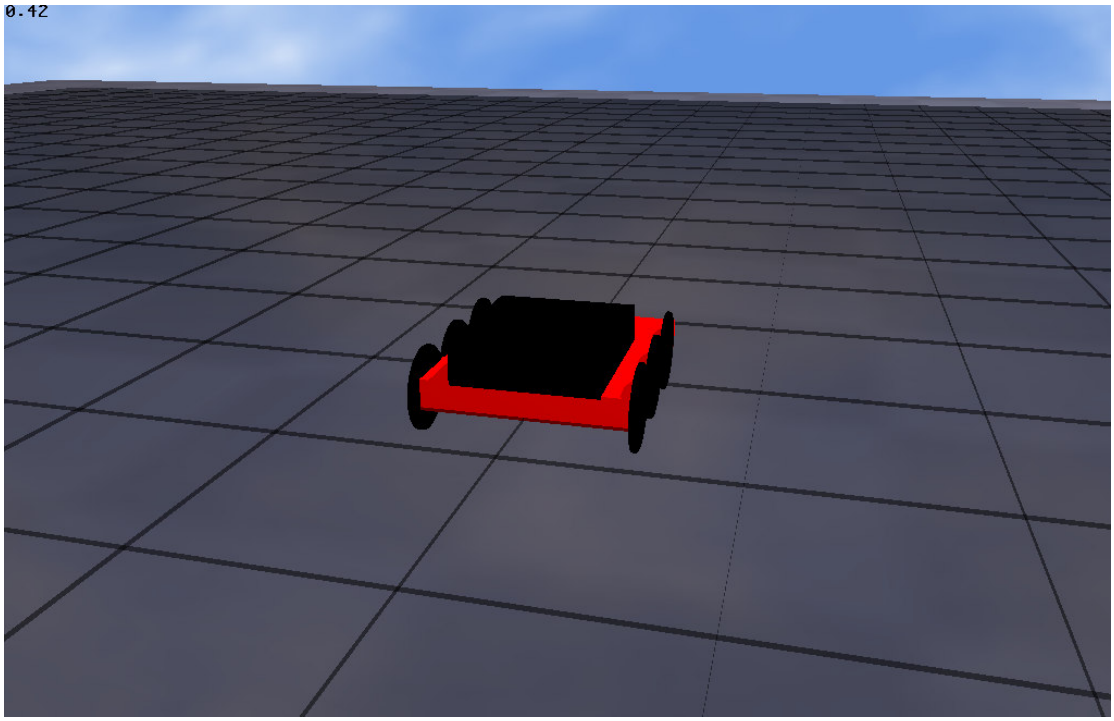

Figure 5.2: Shows the patches used in the simulation

5.4 Light Objects modelled as Obstacles

The light objects which are part of the braitenberg class were used as obstacles. The light objects are mobile objects and could be moved around during the simulation which gives us dynamic obstacles that is obstacles that would change the position with time and in this thesis it was able to take the dynamic nature of the obstacles into consideration and get a safe path avoiding the obstacles at each iteration of the simulation cycle. Figure 5.3 shows the patches, vehicle and obstacles and an interesting observation can be made. The size of the patches, vehicle and the obstacles are all the same. The vehicle size and the patch size were made same in order to ease the vehicle navigation when it moves from one patch to the neighbouring patch. The size of the obstacle and patch were made the same in order to solve the problem discussed in Chapter 2 Section 2.4. Also care was taken to place the obstacles centrally in the patch since this prevents overlapping of a single obstacle with many chapters which was also discussed in Section 2.4.

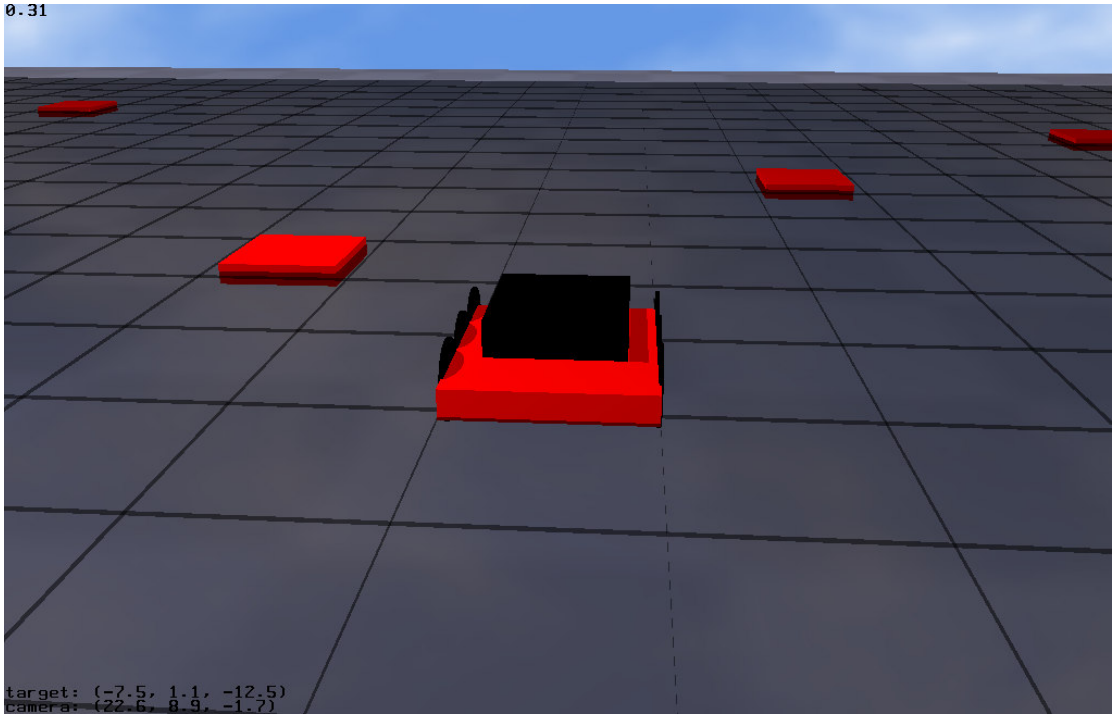

Figure 5.3: The patch, vehicle and obstacle size are all same.

The following piece of code shows how the light obstacles were placed in the simulation.

```
+ to place-obstacle:
            i(int).
            patch (object).
            loc (vector).
            patchList (list).

            patchList = (self getPatchList).
            for each patch in patchList: {

            dic{i++} = patch.

            }
            for i=1, i<=35, i++: {
                if (i==13||i==18||i==20) : {
                    patch = dic{i}.
                    loc = (patch get-location).
                    loc += (2.5,0,2.5).
                    obj add-light at loc.
                }
            }
```

The "if" statement "`if (i==13||i==18||i==20)`" places the obstacle on patches numbered 13, 18, 20. So using this piece of code it is possible to add any number of obstacles and on any patch quite easily.

5.5 Getting the List of Patches Containing Obstacles

The following piece of code shows how patches containing obstacles were identified.

```
+ to getObstacleList:
    obstacleList, obspachList, patchList (list).
    light,patch (object).
    i,length, count(int).
    obstacleList = obj getlightpositions.
    patchList = (self getPatchList).
    foreach patch in patchList: {
            length = |obstacleList| - 1.
          while(length>=0): {
                    if ( (patch get-location)+(2.5,0,2.5) )==   obstacleList{length} : {
                              obspachList{i}=patch.
                               i++.
                    }
         length--.
         }
     }
     return obspachList.
```

This function was written in the main controller class. The getlightpositions function talks to the braitenberg class and gets the vector position of all light objects which were created in the braitenberg class. Next the list of all patches is obtained. For every patch existing in the simulation the vector position of the patch was compared with all the positions of the light objects obtained previously as a list and those patches which resided in the same position as the light objects were stored and returned as obspachlist.

5.6 Implementation of the General Search Algorithm

The first algorithm that was implemented in this thesis was the general search algorithm. This random search algorithm was implemented in order to make sure that it would be possible to implement a search algorithm using the Steve language. The Pseudocode used for this algorithm is:

```
create a list P
 add the start node S, to P giving it one element
 Until first path of P ends with G, or P is empty
        extract the first path from P
        extend the path one step to all neighbors creating X new paths
        reject all paths with loops
        add each remaining new path to P
If G found -> success. Else -> failure.
```

5.6.1 Time Complexity Analysis

Manhattan Distance	Time taken to calculate path in seconds
40	0
45	1
50	11
55	116

Table 5.1: Time taken to calculate paths in seconds

This algorithm crashes for a manhattan distance of 60 or above.

5.6.2 Space Complexity Analysis

The algorithm crashes due to a limit on the number of paths or the number nodes that this software can compute at a given time, considering the memory available. Hence a space complexity study has been done. The maximum paths and maximum nodes are defined as follows:

Maximum paths: - The maximum no of possible paths that the algorithm considers during any iteration as it goes on to find the final safe path.

Maximum nodes: - The number of nodes in the path which contains the maximum number of nodes before a valid safe path is found

Manhattan Distance	Maximum paths	Maximum nodes
30	110	7
40	447	9
45	2282	10
50	5614	11
55	13722	12

Table 5.2: Maximum paths and maximum nodes computed.

Thus it is found that this algorithm crashes due to a huge number of maximum paths to calculate in a single iteration.

5.6.3 Advantages of this Algorithm

It was one of the most easiest algorithms to implement.

It calculated the paths within a manhattan distance of 40 in zero time.

5.6.4 Disadvantages of this Algorithm

The general search algorithm crashes due to space complexity when the manhattan distance is greater than 55.

5.7 Implementation of the A* algorithm

The A* algorithm improves the basic search algorithm by adding certain heuristics.

5.7.1 When two or more paths end in the same node take only the best path

When there are two or paths ending in the same node it makes sense to consider the best possible path and ignore the rest. This will help us reduce the space complexity encountered in the basic search algorithm explained in the section 5.6. So whenever we find that there are more than two paths with the same end node, we calculate the sum of the distance travelled and the remaining distance to the target node and the path with the lowest cost is selected and the rest are simply ignored.

The code used to do this is shown below

```
+ to delete-same-end-paths shortList sampleList (list):
        #sampleList (list).

        length1 (int).
        i,j,k (int).
        i1, i2 (int).

        length1 = |sampleList | - 1.
        j = |sampleList{i}| - 1.
        while (i <= (length1) ): {
                i1 = i .
                 while (i1 < length1): {
                            j = |sampleList{ i }| - 1.
                           k = |sampleList{i1+1}| - 1.
                          if (sampleList{i}{ j } == sampleList{i1+1}{k}): {
                                  if (j<k || j==k): remove sampleList{i1+1}.
                                  else: remove sampleList{i}.
                                i1=i.
                  length1 = |sampleList| - 1.
                  }
          i1++.
         }
```

The A* algorithm uses this heuristic and also sorts the paths with the lowest cost path first so that the lowest cost path is always expanded first. The pseudocode for the A* algorithm is as shown below:

```
create a list P
add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
        extract the first path from P
        extend first path one step to all neighbors creating X new paths
        reject all paths with loops
        for all paths that end at the same node, keep only the shortest one.
        add each remaining new path to of P
        Sort all paths by total underestimate, shortest first.
 If G found -> success. Else -> failure.
```

5.7.2 Time Complexity Analysis

Sample readings obtained using the A* algorithm is shown in the following table:

Manhattan Distance	**Time taken to calculate path(seconds)**	**Manhattan Distance**	**Time taken to calculate path(seconds)**
80	1	170	27
110	2	175	31
120	3	180	39
125	4	185	45
130	5	190	41
135	7	195	43
140	8	200	74
145	10	205	86
150	13	210	99
155	16	215	148
160	20	220	168
165	23	225	214

Table 5.3: Time taken to calculate paths in seconds

5.7.3 Space Complexity Analysis

Manhattan Distance	Maximum paths	Maximum nodes
30	13	7
35	18	8
40	21	9
45	27	10
50	31	11
55	37	12
60	43	13
65	51	14
85	83	18
125	171	26
150	223	31
200	321	41
220	358	45

Table 5.4: Maximum paths and maximum nodes computed.

Comparing table 5.2 with 5.4 the A* algorithm could possibly crash due to an increase in the maximum number of nodes that can be expanded, unlike the general search algorithm which crashed due to the maximum number of paths that could be expanded.

5.7.4 Advantages of the A* algorithm

Comparing the table 5.1 and 5.2 it can be concluded that the A* algorithm performs vastly better than the general search algorithm. Since A* algorithm expands the lowest cost paths first, it can be expected to give the optimal path.

5.7.5 Disadvantages of the A* algorithm

Given a list which contains paths ending in the same node the A* algorithm chooses the lowest cost one and ignores the rest. This methods is not very efficient because of 2 reasons

- The path which has more cost presently may in the longer run be a cheaper cost.

- Such rash deletion of paths may even delete the only path that may be available in certain situations.

5.8 Adding the additional heuristic to the General Search Algorithm

Deleting paths that exceed the maximum manhattan distance

The maximum distance is the manhattan distance between the start node and the end node. It is quite obvious that any path which is greater than the manhattan distance will not an efficient path.

5.8.1 Time Complexity Analysis

The sample readings using this additional heuristic to the is shown below

Manhattan Distance	Time taken to calculate path(seconds)
40	1
45	1
50	3
55	57
60	Infinite

Table 5.5: Time taken to calculate paths in seconds

Comparing the values in the table 5.5 and the table 5.1, we see that adding this heuristic has certainly improved the performance of the general search algorithm by decreasing the time taken to calculate paths. Hence if we add this heuristic to the A* algorithm, it should improve the performance of the A* algorithm.

5.8.2 Space Complexity Analysis

Manhattan Distance	Maximum paths	Maximum nodes
30	80	7
35	201	8
40	452	9
45	1020	10
50	2198	11
55	4776	12
60	10106	13

Table 5.6: Maximum paths and maximum nodes computed.

Comparing the values obtained in table 5.1 and 5.6, it can be seen that the addition of this heuristic has decreased the space complexity of the general search algorithm. Lets now apply this heuristic to the A* algorithm and check how it improves the A* algorithm.

5.9 Adding an additional heuristic to the A* algorithm

After adding the heuristic discussed in section 5.8 to the A* algorithm the new algorithm will be as follows

```
create a list P
 add the start node S, to P giving it one element
 Until first path of P ends with G, or P is empty
      extract the first path from P
      extend first path one step to all neighbors creating X new paths
      reject all paths with loops
           reject all paths exceeding maxdist
           for all paths that end at the same node, keep only the shortest one.
      add each remaining new path to P
      Sort all paths by total underestimate, shortest first.
 If G found -> success. Else -> failure.
```

The change from the A* algorithm has been shown in italics

5.9.1 Time Complexity Analysis

The readings obtained using this modification is shown in table 5.4.

Manhattan Distance	**Time taken to calculate path(seconds)**
160	11
180	25
210	70
215	91
220	123
225	140
230	165
235	220
240	238

Table 5.7: Time taken to calculate paths in seconds

Comparing the table 5.3 and the table 5.7 it can be concluded that adding this additional heuristic has certainly improved the performance of the A* algorithm in terms of time taken.

5.9.2 Space Complexity Analysis

Manhattan Distance	Maximum paths	Maximum nodes
35	9	8
40	11	9
45	12	10
50	13	11
55	15	12
60	19	13
65	21	14
70	24	15
75	28	16
80	30	17
85	33	18
125	58	26
150	115	31
200	273	41
220	347	45
225	351	46
230	374	47

Table 5.8: Maximum paths and maximum nodes computed.

Comparing the table 5.4 and table 5.8 it can be seen that the additional heuristic has decreased the number paths expanded by the A* algorithm even though there is no improvement in the maximum number of nodes expanded.

5.9.3 Advantages of the Modified A* Algorithm

The addition of a simple common sense heuristic has improved the performance of the A* algorithm by deleting inefficient paths.

5.9.4 Disadvantages of the modified A* algorithm

Along with the disadvantages mentioned for the A* algorithm, the addition of the new heuristic introduces another small issue when obstacles are present. The presence of obstacles will make paths, which are greater than the maximum manhattan distance, valid paths. So when obstacles are present this new heuristic might never be able to give us a solution since it deletes all paths greater than the manhattan distance between the start node and the target node. But this issue can be resolved by making the manhattan distance greater by the number of obstacles present.

5.10 Dynamic path finding

Dynamic path finding is where the obstacles move and change position with time. This poses a challenge to the moving vehicle, since it needs to change its path as the obstacles change their position. The snapshots taken from a single run of the simulation at different time steps is shown next

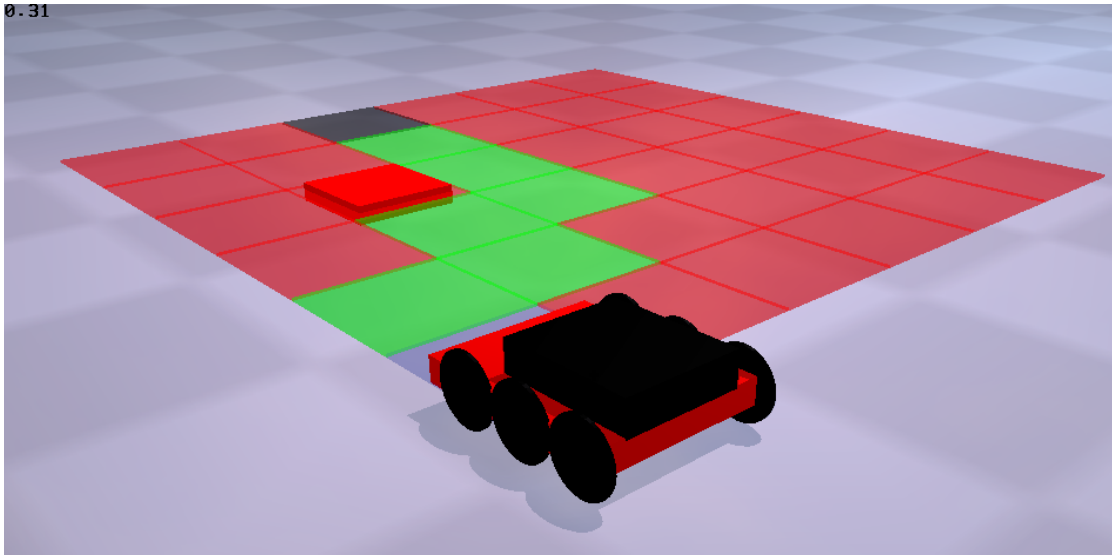

Figure 5.4: Shows the initial path found.

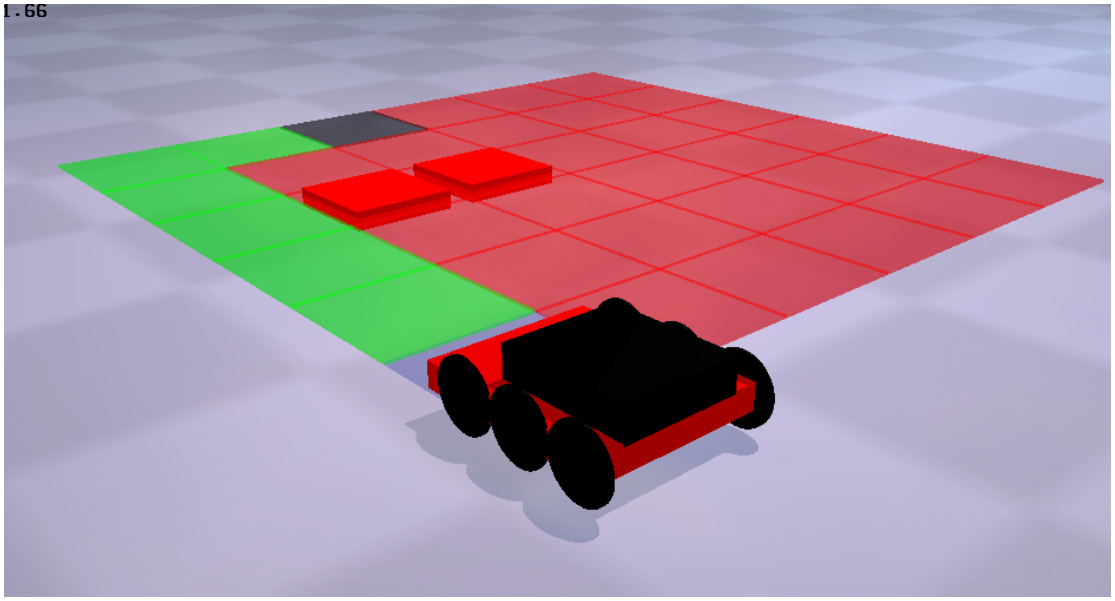

Figure 5.5: Shows the introduction of a new obstacle, which was in the path of the original path found. The algorithm has found a new safe path avoiding the newly introduced obstacle.

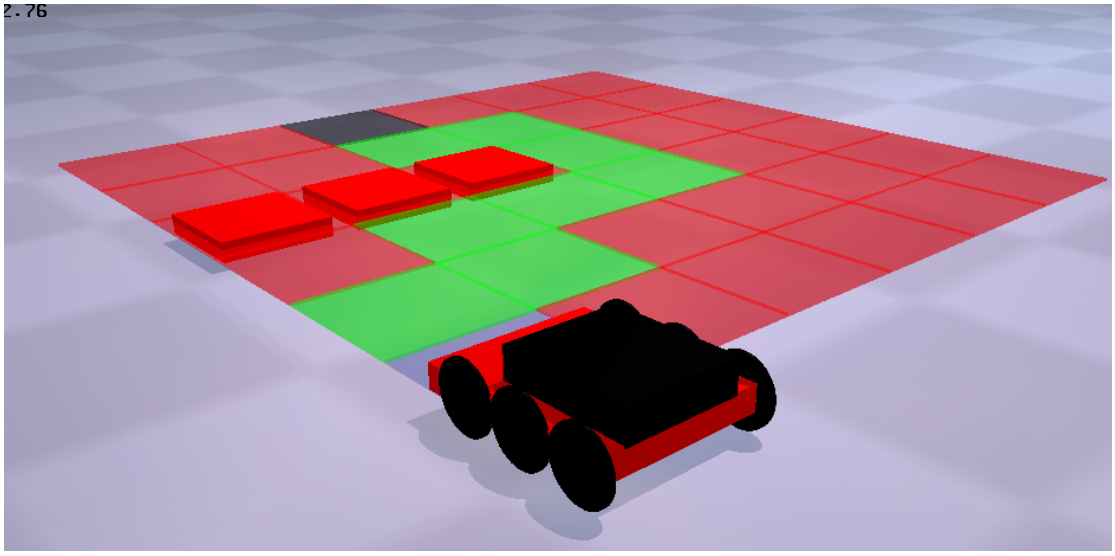

Figure 5.6: Shows the introduction of another new obstacle which was again in the path of the vehicle and a new path was found.

5.11 Path Finding Used in Homeland Security Robots

Finally we can now test path finding techniques for robots used in homeland security. The following snapshots show the vehicle taking safe paths avoiding the "danger spots".

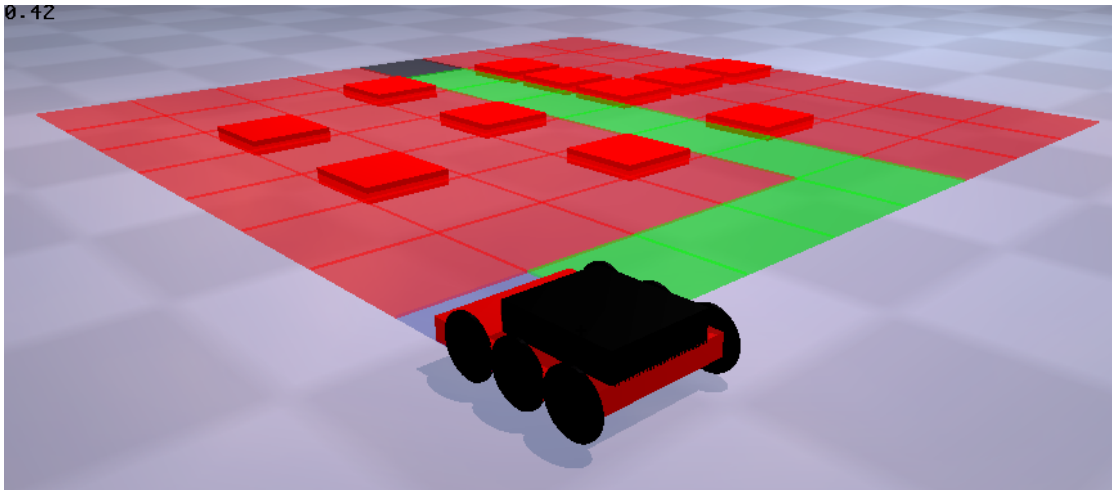

Figure 5.7: A safe path is found for the robot to navigate

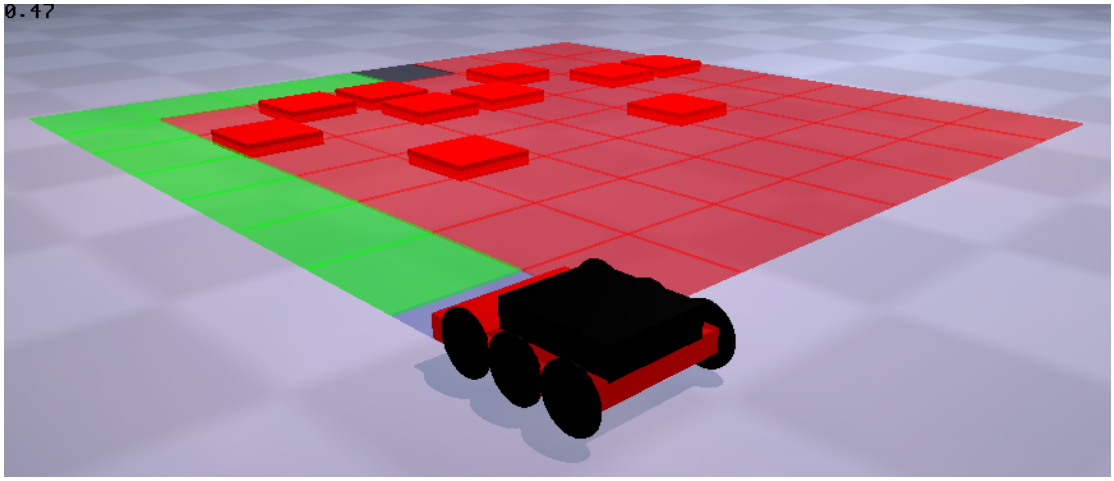

Figure 5.8: Another snapshot showing a safe path found.

The snapshots shown in Figure 5.7 and Figure 5.8 show the safe paths found. The snapshot shown in Figure 5.9 is very interesting because although the vehicle has found a safe path it has not found the safest path. That is on careful scrutiny it can seen the path found does not take care to keep the vehicle at a maximum distance from the target. This could be a serious concern. Consider for eg an army unit moving through a battle field avoiding enemy camps, in this situation it would be highly desirable to navigate the army unit or vehicle as far away from the “danger spots” as possible.

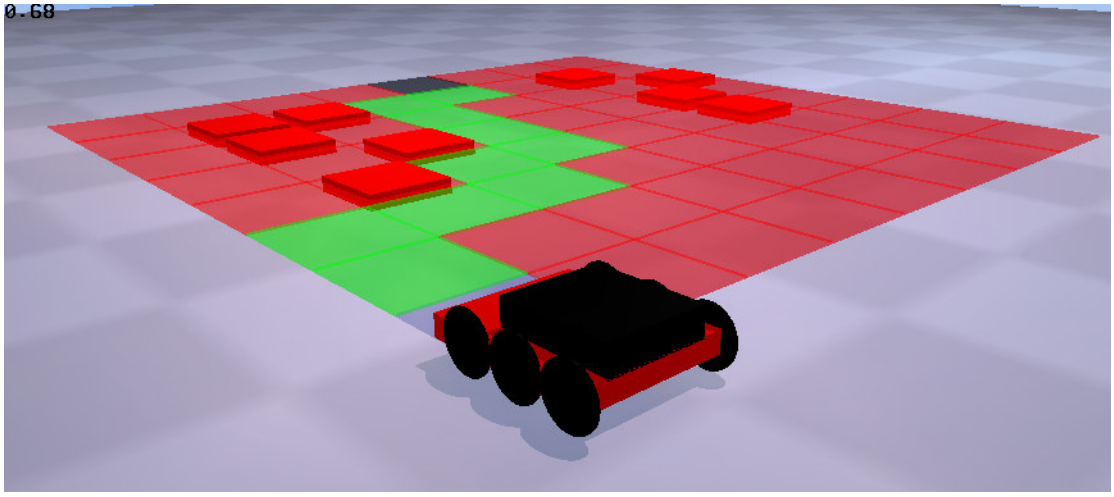

Figure 5.9 A safe path but not the safest path is found.

5.12 Placement of Obstacles

The following snapshots show the paths found for a complex arrangements of obstacles.

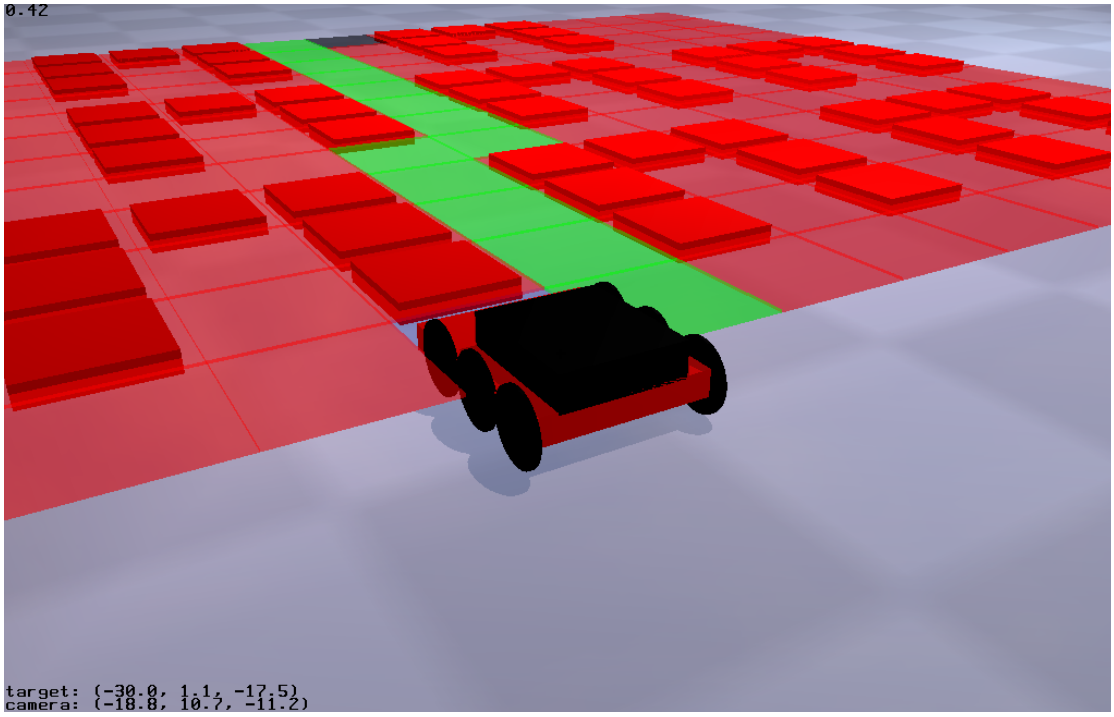

Figure 5.10: Safe path taken in a 12X12 matrix with obstacles places in a open square fashion.

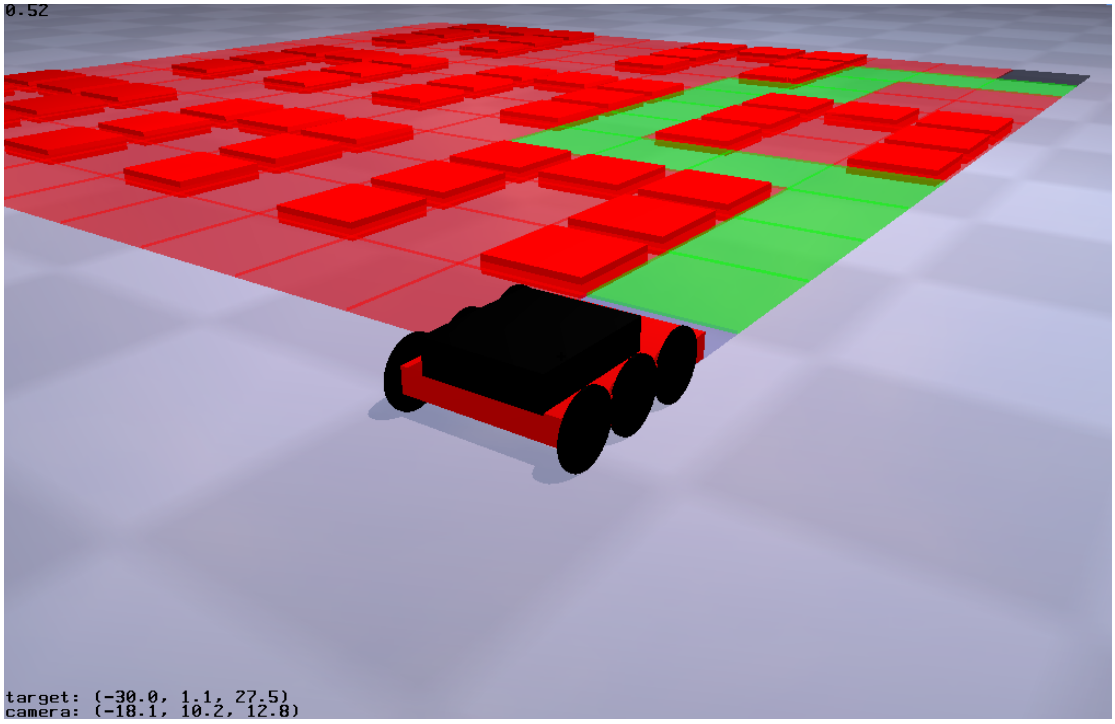

Figure 5.11: Safe path taken in a 12X12 matrix with obstacles places in a open square fashion.

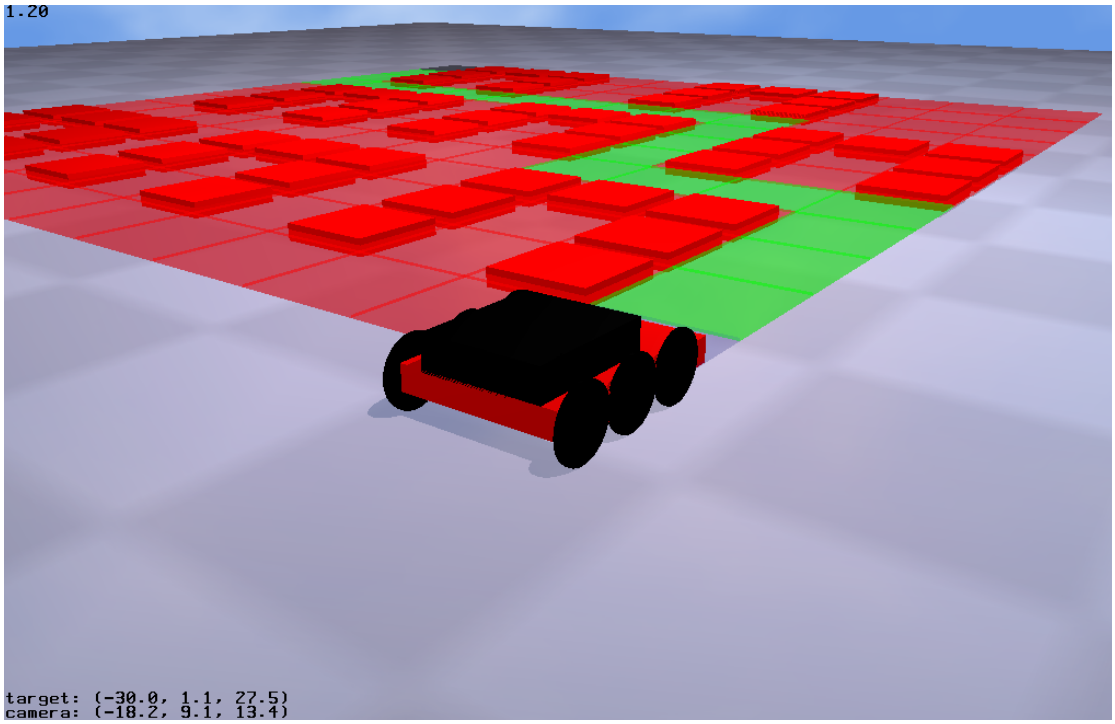

Figure 5.12: Safe path taken in a 12X12 matrix with obstacles places in a open square fashion.

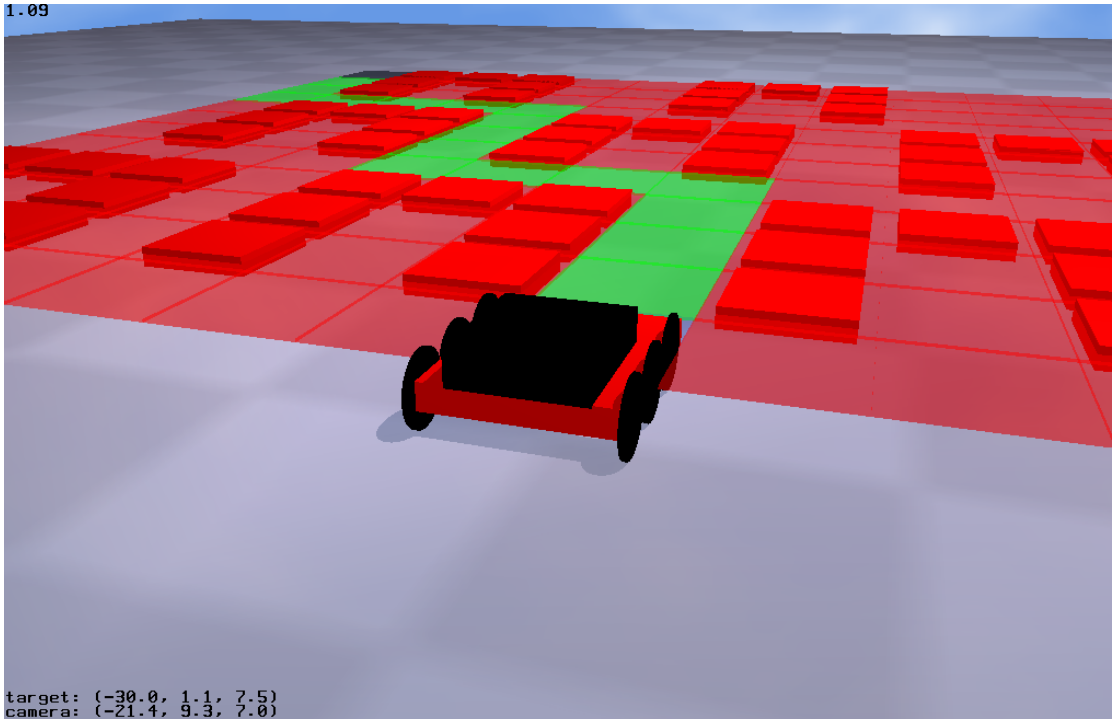

Figure 5.13: Safe path taken in a 12X12 matrix with obstacles places in a open square fashion.

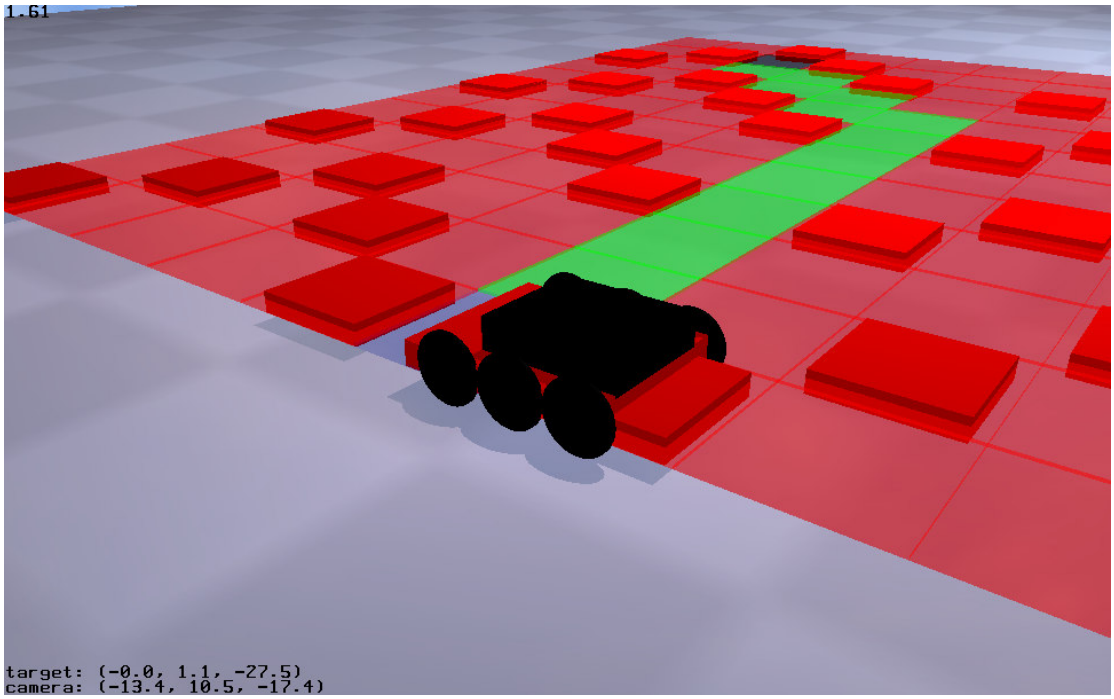

Figure 5.14: Safe path taken in a 12X12 matrix with obstacles places in a triangular fashion.

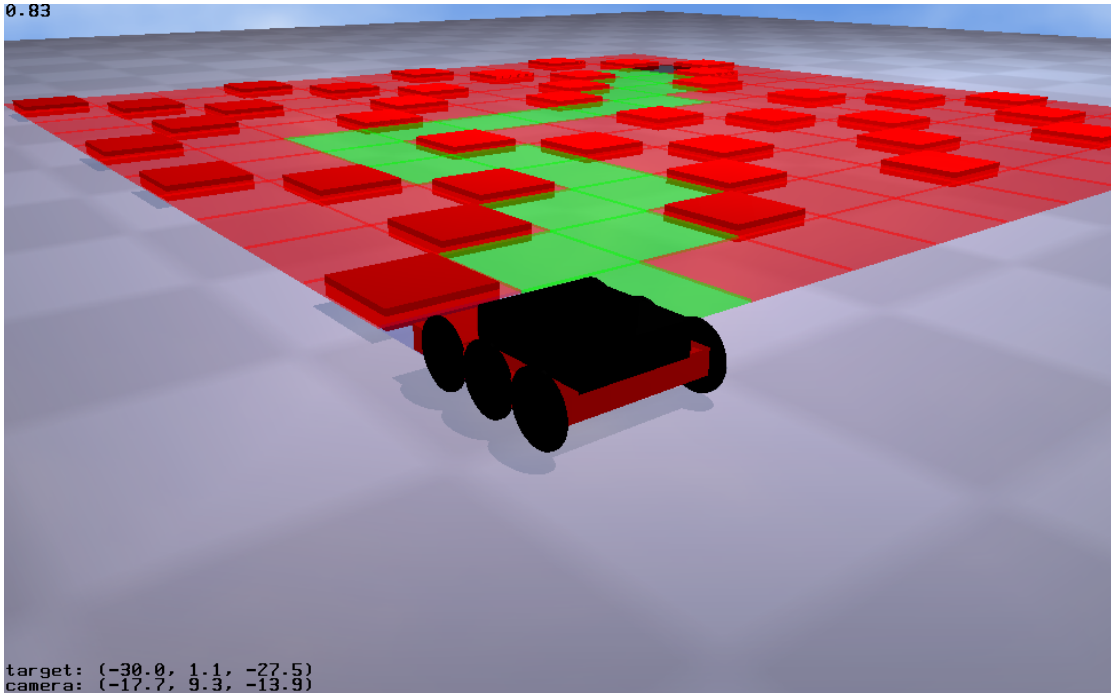

Figure 5.15: Safe path taken in a 12X12 matrix with obstacles places in a triangular fashion.

5.14 The longest path calculated by the Improved A* algorithm

Fig5.14 shows the longest path calculated by the A* algorithm. It took 2.6 minutes to calculate this path. This was done for a 26X26 matrix containing 676 nodes totally.

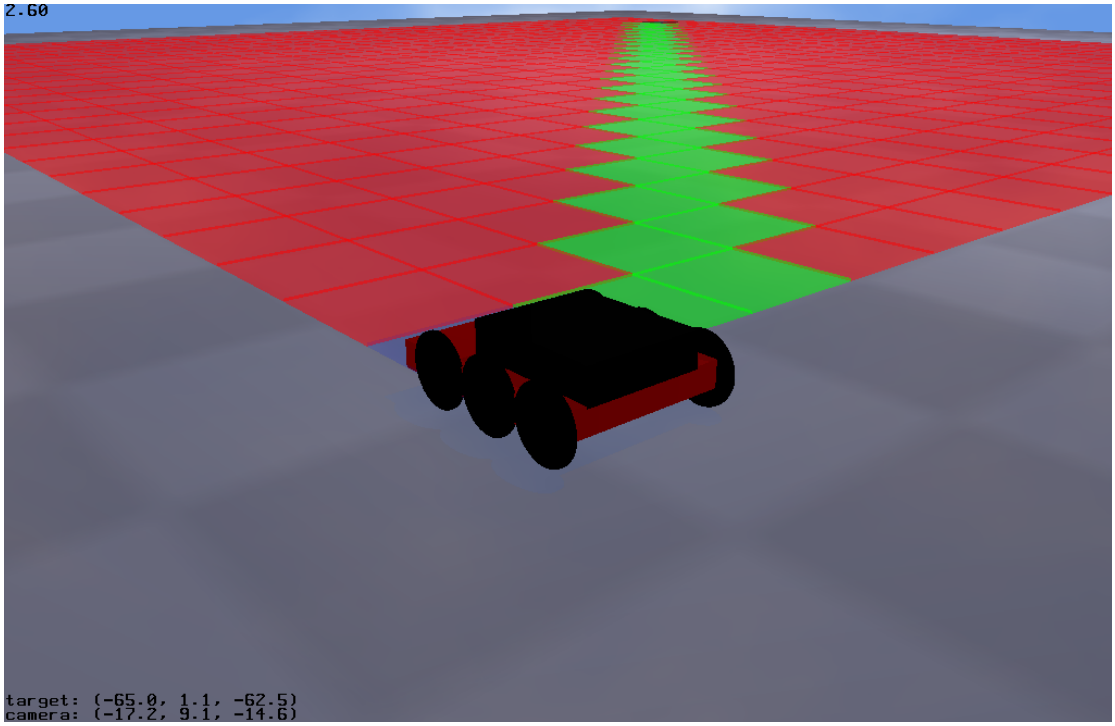

Figure 5.14: Shows the longest path calculated by the improved A* algorithm

5.15 Summary

This chapter started with more discussion about Breve and then showed the performance of the general search algorithm. The advantages and disadvantages of the general search algorithm was discussed. The performance of the A* algorithm was then illustrated and shown to be better than the general search algorithm. Finally it was proved through experimental data that an additional heuristic could improve the performance of the A* algorithm. Dynamic path finding implemented in the thesis was illustrated with snapshots. Finally how the path finding techniques implemented could be used in homeland robotics was shown.

Chapter 6

Conclusion and Further Work

6.1 Discussion

This thesis has developed and analysed the use of different path finding techniques in the same work space which is of importance to the robots used in homeland applications.

The objective of the project was to analyse the performance of various path finding techniques and to come up with a better technique by combining their strengths and nullifying their drawbacks.

Different search algorithms were used in the same work space and the time taken to calculate the paths using the different algorithms were recorded most efficiently in Breve.

The general search algorithm contains no heuristics or assumptions. The A* search algorithm uses a very effective heuristic and also sorts the paths so that the path finding is more efficient. The performance of the general search with the A* search was compared and it was proved that the general search algorithm is no match to the A* search. Finally an additional heuristic was added to the A* algorithm and it was proved through experimental data that this heuristic improves the performance of the A* algorithm.

This improved A* algorithm was then used for robots used in homeland applications. It was found that our algorithm was very efficient in terms of finding a safe path avoiding all “danger spots”. It was also found that this algorithm sometimes finds the shortest path instead of the safest path as would be the requirement in most homeland applications. Hence it can be concluded that although the algorithm developed in this work can be used in homeland applications it cannot be used in sensitive mission critical homeland applications where it is a matter of life and death.

Another interesting path finding technique using potential fields as discussed in section 2.6 can be used. This method has been used in [15]. But potential field methods tend to suffer from local minima i.e. the navigating vehicle may get stuck in regions of local minima. Care should be taken to avoid this.

The advantages and disadvantages of all the algorithms used were clearly stated. This should enable us to explore more path finding techniques and come up with a more efficient technique by combining the strengths of various techniques and correcting the drawbacks in each algorithm.

6.2 Further Work

6.2.1 Sensor Deployment

In this project where an ad-hoc sensor network was modelled using the patch class, the patches representing the sensors were distributed uniformly. In [17] algorithms for sensor deployment were discussed. It would be interesting to study and implement the sensor deployment algorithm.

6.2.2 Map Building

This project deals only with path finding. The algorithms assume that the sensor locations are known. Breve has a very efficient collision handling methods. Using this it should be possible to implement map building techniques. The robot can be made to explore the environment and mark the safe spots and the danger spots. Map building in mobile robotics is still under developmental stages. Simultaneous Localization and Mapping (SLAM) [19] is one of the most used techniques in map building. So another interesting further work would be to first implement map building techniques to build a map of the environment and then use path finding techniques using this information.

6.2.3 SMA* Algorithm

The disadvantages of the algorithms implemented in this thesis as described in section 5.7.2 and section 5.9.2 can be overcome by using the SMA* algorithm which is a modification of the A* algorithm. The algorithms implemented in this thesis consider only the best path and do not give a chance for a path which is presently bad to develop into an efficient path in future. The SMA* algorithm on the other hand

remembers the best path of the forgotten paths and comes back to the forgotten path recursively when it finds that the present path is not doing all that very well.

6.2.4 Cell Decomposition

The cell decomposition method used in this project divides the work space into equal sized cells. The obstacles are made the same size of the cells and are placed exactly in any one cell at any time. This is a luxury when compared to real life scenarios where the obstacles can be of different shapes and same obstacles can reside in different squares. In other words some part of the cell may contain an obstacle and other parts of the cell may be free. If we consider the entire cell to be unsafe then we would not get an efficient path. So one way to resolve this would be further sub divide the cell to find the region in the cell which is free. This issue discussed in section 2.4 can be resolved using more advanced geometric technique which has not been done in this project.

6.2.5 Terrains

In this project a plain terrain was used. Using Breve it is possible to model different kinds of terrains like hills. The robots used in the WTC were unable to withstand the rigors of rubble. The effect of using the path finding techniques on different terrains can be analysed.

6.3 Conclusion

The path finding technique developed in this work was implemented in a simulated homeland security environment. This technique was proved to take lesser time than the A* algorithm. Given any start node, target node and positioning of obstacles the algorithm always returned a safe path avoiding all danger spots. The only drawback found in this algorithm was that it calculates optimal paths instead of safest paths which may be critical in some homeland security applications. Hence it can be concluded that this algorithm can be used in non critical homeland applications and the algorithm needs to be modified for use in critical homeland applications.

Reference

[1] ACM Transactions on Sensor Networks (TOSN) Volume 1 , Issue 1 (August 2005) table of contents Pages: 3 - 35 Year of Publication: 2005 ISSN:1550-4859

[2] Sven Behnke "Local Multiresolution Path Planning",In Proceedings of 7th RoboCup International Symposium, Padua, Italy, 2003. RoboCup-2003: Robot Soccer World Cup VII, LNCS 3020, pp. 332-343, Springer, 2004

[3] COMMUNICATIONS OF THE ACM March 2004/Vol. 47, No. 3

[4] Stuart Russell, Peter Norvig, A Modern Approach, Pearson Education, Inc. 2003, ISBN (0-131-03805-2), January, 1995

[5] R. Brooks and T. Lozano-P´erez. A subdivision algorithm in configuration space for findpath with rotation. In Proceedings of the 8th International Conference on Artificial Intelligence (ICAI), pages 799–806, 1983.

[6] C. O'Dunlaing and C. K. Yap. A 'retraction' method for planning the motion of a disc. Journal of Algorithms, 6:104–111, 1986.

[7] L. Kavraki, P. Svestka, J.C. Latombe, and M. Overmars. Probabilistic road maps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation, 12(4):566–580, 1996.

[8] H. Van Dyke ParunaK , Sven Brueckner , John Sauter, Digital pheromone mechanisms for coordination of unmanned vehicles, Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1, July 15-19, 2002, Bologna, Italy

[9] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels. Selforganized Shortcuts in the Argentine Ant.Naturwissenschaften, 76:579-581, 1989.

[10] Klein, J. 2002. Breve: a 3D simulation environment for the simulation of decentralized systems and artificial life. Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems. The MIT Press.

[11] http://www.spiderland.org/Breve/

[12] Valentino Braitenberg, Vehicles: Experiments in Synthetic Psychology, MIT Press, 1986

[13] http://en.wikipedia.org/wiki/Homeland_security

[14] John Yen, Communications of the ACM March 2004/Vol. 47,No 3.

[15] ACM Transactions on Sensor Networks (TOSN) Volume 1 , Issue 1 (August 2005) table of contents Pages: 6 Year of Publication: 2005 ISSN:1550-4859

[16] Dan Hampel and Stefano DiPierro, Booz Allen Hamilton, "Networked Sensor Communications", 2002 IEEE.

[17] Y. Zou and K. Chakrabarty, "Uncertainty-Aware Sensor Deployment Algorithms for Surveillance Applications", Ad Hoc Networks, vol. 1, pp. 261-272, 2003.

[18] G. J. Pottie and W. J. Kaiser, "Wireless sensor networks", Communications of the ACM, vol. 43, pp. 51-58, May 2000.

[19] Stefan B.Williams, Hugh Durrant-Whyte, Gamini Dissanayake "Constrained Initialization of the Simultaneous Localization and Mapping Algorithm" in The International Journal of Robotics Research Vol. 22, No. 7–8, July–August 2003, pp. 541-564,©2003 Sage Publications

[20] Senior Diablo, http://ai-depot.com/Tutorial/PathFinding.html

Appendix A:

Source Code used in this Thesis

```
@use Braitenberg.
@use Link.
@include "Patch.tz"

Controller main.

@define X_SIZE 8. #determines the matrix of patches
@define Y_SIZE 1.

#The main class where the execution starts
Control : main {
	+ variables:
	   obj (object).
	   patches (object).
	   x, flag, maxdist (int).
	   time1, time2, time (float).
	   startTargetList (list).
	   dic (hash).
	+ to init:
	   self set-integration-step to 1.0.
	   self set-iteration-step to 1.0.
	   #Creates an instance of the main vehicle class
	     obj = new myBraitenbergControl.
	   #creates the patches
	   patches = (new PatchGrid init-at location (0,0.75,0)
	            with-patch-size(5, 0.1, 5) with-x-count X_SIZE with-y-count Y_SIZE
	            with-z-count 8 with-patch-class "LifePatch").

	   x=1.
	#self schedule method-call "findPath" at-time 1.0.
	#self schedule method-call "findPath" at-time 2.5.

	    #Given a patch object this function shows its neighbors
	+ to showneigbors:
	   patch (object).
	   i (int).
	   patchList, neighborList (list).
	   patchList = (patches get-patches).
	   for each patch in patchList: {

	   if (i==9): {
	   neighborList = patch get-neighbors of patch.
	   neighborList = self deleteobs neighList neighborList.
	   patch set-color.
	   for each patch in neighborList: patch set-path-color.
```

```
    }
    i++.
    }
     # When there are 2 are more paths ending in the same node
     # only the best path is taken
+ to delete-same-end-paths shortList sampleList (list):
   #sampleList (list).

   length1 (int).
   i,j,k (int).
   i1, i2 (int).

   length1 = |sampleList | - 1.
   j = |sampleList{i}| - 1.
   while (i <= (length1) ): {
    i1 = i .

    while (i1 < length1): {
     j = |sampleList{ i }| - 1.
     k = |sampleList{i1+1}| - 1.
     if (sampleList{i}{ j } == sampleList{i1+1}{k}): {
      if (j<k || j==k): remove sampleList{i1+1}.
      else: remove sampleList{i}.
      i1=i.
      length1 = |sampleList| - 1.
     }
     i1++.
    }
    i++.

   }
   return sampleList.

     # Deletes a any path that exceeds the maximum manhattan
     # distance
+ to delete-long-paths list1 list1 (list):
  length1, length2, i (int).
  covered, tocover, total (int).

  length1 = |list1| - 1.
  while(i<length1): {
   length2 = |list1{i}| - 1.
   covered = self get-manhattan-dist patch1

            list1{i}{length2} patch2 list1{i}{0}.
   tocover = self get-manhattan-dist patch1
            list1{i}{length2} patch2                startTargetList{1}.
   total = covered + tocover.

   if (covered+tocover) >maxdist: {
```

```
      remove list1{i}.
      i--.
      length1 = |list1| - 1.
      }
      i++.
    }
    return list1.

      #Function which return the list of all patches
+ to getPatchList:
    patchList (list).
    patch (object).

    i(int).

    patchList = (patches get-patches).
    for each patch in patchList: {
    patch set-color.
    dic{i++} = patch.
    }
    return patchList.

#Function to place obstacles
      + to place-obstacle:
    i(int).
    patch (object).
    loc (vector).
    patchList (list).

    patchList = (self getPatchList).

    for each patch in patchList: {

    dic{i++} = patch.

    }
    for i=1, i<=64, i++: {
       if (i==27||i==30||i==42) : {
          patch = dic{i}.
          loc = (patch get-location).
          loc += (2.5,0,2.5).
          obj add-light at loc.
       }
    }

# Function to calculate the manhattan distance between 2
      # patches
+ to get-manhattan-dist patch1 patch1 (object) patch2 patch2
            (object):
    point1, point2 (vector).
```

```
    a,b (float).
    point1 = (patch1 get-location).
    point2 = (patch2 get-location).
    #print "point1 = $point1".
    #print "point2 = $point2".
    a = point1::x - point2::x.
    b = point1::z - point2::z.
    if a<0: a = -a.
    if b<0: b = -b.
    return a+b.
    #print a+b.


    # Returns a list of all the patches containing obstacles
+ to getObstacleList:
    obstacleList, obspachList, patchList (list).
    light,patch (object).
    i,length, count(int).
    #patch = new PatchLife.
    obstacleList = obj getlightpositions.
    #print "obs = $obstacleList".
    patchList = (self getPatchList).
   # #print "patches = $patchList".

    foreach patch in patchList: {
        length = |obstacleList| - 1.
        while(length>=0): {
            if ( (patch get-location)+(2.5,0,2.5) )==
                obstacleList{length} : {
            obspachList{i}=patch.
            i++.
            }
            length--.
        }
    }

      #foreach patch in obspachList: patch set-path-color.
      #print "obs = $obspachList".

       return obspachList.


       #Set the Start and Target Patch and return list containing #them
+ to setStartTarget:
    patchList  (list).
    patch (object).
    count (int).
    loc (vector).

    patchList = (self getPatchList).
```

```
    foreach patch in patchList:  { #print (patch get-
            location).

    if count==0 : {
       startTargetList{0} = patch.
       patch set-state-start.
       loc = (patch get-location) + (0,0.7,2.5).
    }

    if count==60: { startTargetList{1} = patch. patch set-
            state-target.}
    count++ .
    }
    obj move to loc.
    maxdist = self get-manhattan-dist patch1
            startTargetList{0} patch2 startTargetList{1}.
    print "maxdist=$maxdist".
    return startTargetList.

#Delete the patches which contain obstacles from the
      neighbors List
+ to deleteobs neighList neighList (list) :
    obsList, tempList (list).
    i, j, k, length2 (int).
    patch (object).
    j=0. k=0.

    obsList = self getObstacleList.

    for each patch in neighList: {
       i=0.
       length2 = |obsList| - 1.
       while(length2 >= 0) : {
          if (patch == obsList{length2} ) : {i=1.}
          length2--.
        }

       if(i==0): {
          tempList{j} = neighList{k}.
          j++.
       }
       k++.
    }

    return tempList.

     #Function to sort a list of possible paths with the
     #lowest cost first
     + to sortList tosortList tosortList (list) targetPatch
            targetPatch object):
```

```
    length1, length2, length3, x, y, index_of_min (int).
    distance1, distance2 (float).
    tempList (list).
    for x=0, x<length1, x++: {
      index_of_min = x.
       for y=0, y<length1,y++ : {
          length2 = |tosortList{index_of_min}| - 1.
          length3 = |tosortList{y}| - 1.

          distance1 = self get-manhattan-dist
                      tosotList{index_of_min}{length2}.
          distance2 = self get-manhattan-dist
                        tosotList{y}{length3}.

          if distance1 > distance2 : {
             index_of_min = y.
          }
       }
       tempList = tosortList{x}.
       tosortList{x} = tosortList{index_of_min}.
       tosortList{index_of_min} = tempList.
    }
    return tosortList.
+ to display the listtodisplay (list):


     #Function which finds the path from the start patch to the
     #target patch
+ to findPath:
    pathList, tempList1,tempList2, tempList3, tempList4,
              neighborList,startTargetList (list).
    obstacleList, tempList5 (list).
    index, length1, length2, length3, length4, i , j, k,
              found, terminate (int).
    forindex (int).
    patch, dummy(object).

    self place-obstacle.

    obstacleList = self getObstacleList.
    startTargetList = self setStartTarget.

    pathList{0} = startTargetList{0}.

    tempList1{0} = pathList{0}.
    neighborList = tempList1{0} get-neighbors of
              tempList1{0}.
    neighborList = self deleteobs neighList neighborList.

    length3 = |neighborList| - 1.
```

```
j=0.
while(length3>=0): {
   i=0.
   tempList2{i} =  tempList1{0}. i++.
   tempList2{i} = neighborList{length3}.
   pathList{j} = copylist tempList2.
   length3--.
   j++.
}

index=0. found=0.

while(found==0): {
   length1 = |pathList| - 1.
while(length1 >=0) : {
   tempList1 = pathList{length1}.
   length2 = |tempList1| - 1.
   neighborList = tempList1{length2} get-neighbors of
         tempList1{length2}.
   neighborList = self deleteobs neighList neighborList.

   length3 = |neighborList| - 1.
   while (length3>=0) : {
    if(self not-circular mainList tempList1 item
                  neighborList{length3}): {
      tempList2 = copylist tempList1.
      push neighborList{length3} onto tempList2.

      tempList3{index} = copylist tempList2.
      index++.
      }
      length3--.
   }

   length1--.
   }

   pathList = copylist tempList3.
            pathList = self delete-same-end-paths shortList

   pathList = self delete-long-paths list1 pathList.

   index=0.

   length4 = |pathList| - 1.

   while (length4 >=0) : {
      tempList4 = copylist pathList{length4}.
      length2 = |tempList4| - 1.
```

```
            if (tempList4{0}==startTargetList{0} &&
        tempList4{length2}==startTargetList{1}): {
                pathList = copylist tempList4.
                found=1. length4=-1.
                print (controller get-time).
                length3=|pathList| - 1 .
                                foreach patch in pathList:  patch set-path-color.
                pathList{0} set-state-start.
                pathList{length3} set-state-target.
            }
            length4--.
        }
    }

  #Function to delete circular paths
            + to not-circular mainList mainList (list) item item (object):
    flag, length (int).
    flag=1.
    length = |mainList| - 1.
    while(length>=0): {
        if(mainList{length}) == item: flag=0.
        length--.
    }
    return flag.

#Iterate function of the main function
    #Used to get the time elapsed in calculating paths
    + to iterate:

    if flag==0: {
        time1 = (controller get-real-time).

        self findPath.
        time2 =  (controller get-real-time).
        time = time2 - time1.
        print time.

        flag=1.
    }

  }

    #Braitenberg class which created the vehicle
    BraitenbergControl : myBraitenbergControl {
    + variables:
            vehicle, Addbody (object).
            leftbackWheel1, rightbackWheel1, leftbackWheel2,
            rightbackWheel2,leftfrontWheel, rightfrontWheel (object).
    obj (object).
    light1, light2, light3, light4 (object).
```

```
            start, loc (vector).
            location (float).
    flag,superb (int).
    time (float).

    #Function to move vehicle straight
    + to moveStraight:
            leftbackWheel1 set-natural-velocity to 15.0.
      rightbackWheel1 set-natural-velocity to 15.0.
    #Function to stop the vehicle
+ to Stop:
      leftbackWheel1 set-natural-velocity to 0.0.
      rightbackWheel1 set-natural-velocity to 0.0.

    #Function to move the vehicle right
    + to moveRight:
            rightbackWheel1 set-natural-velocity to 5.0.
      leftbackWheel1 set-natural-velocity to 0.0.

#Function to move the vehicle left
    + to moveLeft:
      rightbackWheel1 set-natural-velocity to 0.0.
      leftbackWheel1 set-natural-velocity to 5.0.
      #self sleep for-seconds 10.

#Function to move the vehicle backwards
    + to moveBack:
      #print "moving Back".
      leftbackWheel1 set-natural-velocity to -15.0.
      rightbackWheel1 set-natural-velocity to -15.0.

#Function to make the vehicle to loc
    + to move to loc(vector):
    vehicle move to loc.

    #Function to get the vehicle position
    + to getvehPos:
            return (vehicle get-location).

#Function to get the position of the light obstacles
    + to getlightpositions:
    lights, lightPos(list).
    light (object).

    lights = all BraitenbergLights.

    foreach light in lights: {
     insert (light get-location) at lightPos {0 }.
    }
    return lightPos.
```

```
#Function which returns all the light objects
     + to get-all-lights:
    lights(list).
    lights = all BraitenbergLights.
    #print lights.
    return lights.

#Function which adds a light obstacle
     + to add-light at loc(vector):
    lightobj (object).
    lightobj = new BraitenbergLight.
    lightobj move to loc.
    #loc = (2.5,0.7,-15+2.5).

     #Init function of the braitenberg class which
     #created the vehicle
     + to init:
    #self schedule method-call "add-light" at-time 1.0.
    #self schedule method-call "add-light" at-time 2.0.
    vehicle = new BraitenbergVehicle.
    vehicle set-color to (1.0, 0.0, 0.0).
    self watch item vehicle.
    loc = (0.00000+2.5, 0.70000, -5.00000+2.5).

              leftbackWheel1 = (vehicle add-wheel at (-2.5, 0, 2.5)).
              leftbackWheel1 set-color to (0, 0, 0).
              rightbackWheel1 = (vehicle add-wheel at (-2.5, 0, -2.5)).
              rightbackWheel1 set-color to (0.0, 0, 0).

    leftbackWheel2 = (vehicle add-wheel at (-0.5, 0, 2.5)).
              leftbackWheel2 set-color to (0, 0, 0).
              rightbackWheel2 = (vehicle add-wheel at (-0.5, 0, -2.5)).
              rightbackWheel2 set-color to (0.0, 0, 0).

    leftfrontWheel = (vehicle add-wheel at (1.5, 0, 2.5)).
              leftfrontWheel set-color to (0, 0, 0).
              rightfrontWheel = (vehicle add-wheel at (1.5, 0, -2.5)).
              rightfrontWheel set-color to (0.0, 0, 0).


    vehicle move to (-7.50000, 1, -12.50000).
    flag=0. superb=0.

    Addbody = (vehicle add-body at (-1,0.8,0) ).

     #Iterate function of the braitenberg class
+ to iterate:

       super iterate.
```

```
}


# Patch class
Patch : LifePatch {
   # Declare variables used in this class
   + variables:
      neighbors (4 objects).

   # Set the color of the start patch
   + to set-state-start:
      self set-color to (0,0,5).
      self set-transparency to 0.1.

   # Set the color of the target patch
   + to set-state-target:
      self set-color to (0,0,0).
      self set-transparency to 0.5.

   # Set the color of the patches in the path
   + to set-path-color:
      #self set-transparency to 1.0.
      self set-color to (0,1,0).

   # Set color of the other paches
   + to set-color:
      self set-color to (5,0,0).

   # Gets the patch at a location
   + to get-pach at location (vector):
      patch (object).
      patch = new PatchGrid.
      patch get-patch at-location location.
      return patch.

   # Gets the neighbors of the patch specified
   +to get-neighbors of patch (object):
      neighborList (list).
      i (int).
      i=0.
           if (patch get-patch-to-left) : {
     neighborList{i}= (patch get-patch-to-left). i++.
      }
      if (patch get-patch-to-right) : {
      neighborList{i}= (patch get-patch-to-right). i++.
      }
      if (patch get-patch-towards-plus-z) : {
      neighborList{i}= (patch get-patch-towards-plus-z). i++.
      }
      if (patch get-patch-towards-minus-z) : {
```

```
    neighborList{i}= (patch get-patch-towards-minus-z). i++.
        }

    return neighborList.

}
```