AN ANALYSIS OF CURRENT ALGORITHMS FOR PATH PLANNING IN THE RETURN AND DELIVERY JOURNEY OF GROUND BASED COURIER ROBOTS

By

Joshua Daly

Supervisor(s): Mr. Arnold Hensman

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
B.SC. (HONOURS) IN COMPUTING
AT
INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN
DUBLIN, IRELAND
April 25, 2015

Declaration

I herby certify that this material, which I now submit for assessment on the programme of study leading to the award of **B.Sc. (Honours) in Computing** in the Institute of Technology Blanchardstown, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fulfillment of the requirements of that stated above.

Dated: April 25, 2015

Author: ________________________________

Joshua Daly

Abstract

Abstract should be clear, concise, and should cover the entire project in a fraction of the space, consider:

- Keep the word count low around 250 words.
- Avoid an jargon or ambiguous language.
- Do not use abbreviations.
- Do not reference anything.
- Briefly cover the motivation, problem statement, approach, results and conclusions.

Acknowledgements

Remember to thank the following people:

- My family and friends for putting up with me during the course of this project.
- Arnold Hensman for providing supervision and leading me into robotics.
- Tucker Balch for producing an excellent tutorial on grid based navigation.
- Sven Koenig for taking the time to respond to my queries.
- The Arduino, Raspberry Pi, and DFRobot communities for being awesome.

Table of Contents

List of Tables

List of Figures

Abbreviations

DARPA	Defence Advanced Research Project Agency
SLAM	Simultaneous Localisation and Mapping

Chapter 1

Introduction

1.1 Background

Provide a context to the reader, talk about the project, where the idea came from (extension of 3rd Project), some examples of courier robots today. Summary of the big industry players like Google and Amazon. Why is the project unique. Most of this can be taken from the background provided with the project proposal and literature review submissions for Research Skills.

The ability to select a path to a destination, negotiate obstacles, and anticipate the movement of other people is something that you probably take for granted. To a human these skills are a part of everyday life and do not at first appear to be particularly special, until you try to replicate them. Programming a machine to deal with rush hour traffic or road closures is surprisingly difficult but that has not stop us from trying [9].

1.2 Motivation

This section is a combination of the *Justifications and Benefits* and *Feasibility* sections from the project proposal and should state the following:

- Primary reason for undertaking this project, probably academic.
- Any potential benefits to society and ITB developing this technology has.
- The fact it evolved from a 3rd project and reuses existing equipment.
- Back all this up with sound evidence that suggests that this project has an achievable goal.

1.3 Scope Definition

What is the actual scope of this project? Defining what to exclude is just as important as what is included so state both of these here in bullet points:

The scope of this project **includes**:

- Path Planning: calculating the shortest and safest path to the delivery point.
- State everything else...

The following are **outside** the scope of this project:

- Collection: purely automated collection using beacons or computer vision. Goods will be manually handled by a human operator.
- State everything else...

1.4 Research Question(s)

State the primary research question of this project on its own line possibly in bold or italics. Then follow it with other sub research questions in bullet points:

- Where can mobile delivery robots be used outside of research laboratories?
- State everything else...

1.5 Objectives

One line introducing the purpose of these objectives followed by the objectives themselves again in bullet points:

- To evaluate the effectiveness of path planning techniques in known and partially known environments.
- State everything else...

Chapter 2

Literature Review

In this literature review we will discuss path planning with respect to courier robots, mobile agents created for the sole purpose of shipping goods. Courier robots pose a unique challenge because for the companies that use them time is literally money, so they must be able to compute these paths fast. Our main research question is how efficient are the current path planners at performing this task and how do they compare to each other. What we will analysis is the length of time it takes to produce a path, its tolerance to change, and the natural appearance of the path.

The scope of this document progressively narrows, to begin with it looks at the wider application of courier robots which includes their potential benefits and economic impact. It moves on to define path planning in robotics using a grid based approach. Then three unique path planners are analysed in depth, they include *Dijkstra's Algorithm* [2] a *static* planner, *D* Lite* [10] a *dynamic* replanner, and the main algorithm in this project *Field D** an *interpolation* based replanner. Finally the findings of this literature review are discussed and presented in the conclusion.

2.1 Courier Robots

2.1.1 Field of Application

Until recently robots were mostly confined to industrial and military applications such as working on car assembly lines or delivering hellfire missiles to unsuspecting terrorists [11]. These 'robots' have not changed society as visionaries like *Isaac Asimov* predicted being effective in only narrow situations. Courier and delivery robots are part of an emerging branch of automation known as *service robotics* [12]. This new breed of robots are set to change the world that we live in today because unlike their predecessors they are becoming; (a) relatively affordable, (b) more intelligent, and (c) ubiquitous.

Examples of service robots include robotic vacuum cleaners, lawn mowers, and of course self-driving cars. Every major auto manufacturer in the world is currently developing self-driving prototypes including: BMW, Mercedes, Volvo, GM, Ford, and even Google [9]. The benefits for society span well beyond shipping goods faster and cheaper, it has the potential to save lives. Around *93%* of road traffic accidents in the US are the result of human error, driver assisted technologies can help reduce this figure [13].

2.1.2 Economic and Social Impact

No new technology comes without potential negatives, take courier robots in the transport industry which at its core involves moving items from point *a* to *b*, be it goods, post, or people. The incentive to adopt these new technologies is huge(!) self-driving auto-mobiles do not; tire, blink, fall a sleep, act irrationally, or demand the minimum wage. A study conducted by the University of Oxford predicted that *47%* of all jobs in the US are under threat from this new type of automation with 'Cargo and Freight Agents' facing a *99%* probability that their jobs will be computerised [14].

2.2 Path Planning

2.2.1 Defining the Problem

Now that we have established the context for this study it is time to look at a specific task that courier robots need to accomplish. For a mobile agent to be considered useful, whether it is a vacuum cleaner, lawn mower, self-driving car, or a robotic arm, it must be able to navigate its environment. Autonomous navigation in robotics is made up of three components [15]:

1. Mapping
2. Localisation
3. Path Planning

Mapping refers to a robots ability to discover the make-up of its environment using some form of external sensor. Localisation involves accurately establishing where a robot is in relation to its environment using a combination of internal and external sensors. Lastly there is path planning, the classic ‘how do I get from point *a* to point *b*?’ problem.

Path planning is the name given to the set of steps required to reach a desired goal state from a start state [5]. The term can be applied to a broad domain and covers everything from solving crossword puzzles to artificial intelligence, what varies is the type of states. In mobile robotics the states are typically expressed in the form of a coordinate and the task can be defined simply as ‘given a start state reach the goal efficiently’. By far one of the most popular ways of expressing points in relation to a robot is the *Cartesian System* [15] which is well suited to grid based path planning.

2.2.2 Representing the World as a Grid

There are many ways in which a robot can navigate the world around it be it using landmarks, beacons, or grids. What differs between them all is how they represent that world in memory [4], what we will focus on here is grid based path planners for 2D environments. A grid based path planner uses an occupancy grid made up of individual cells that contain a probability of whether or not it is occupied.

The main advantage of using a 2D grid is that it can be easily traversed as every cell is connected to eight other cells to which the robot can progress, the robot simply follows the path produced by the planner point by point (Figure: 2.1). Updating the occupancy grid is also straight forward as the change can be mapped to the cell(s) using x and y coordinates. The disadvantage to this approach is memory consumption, high resolution grids require a lot of memory as each cell contains its own data used by the planner [4].

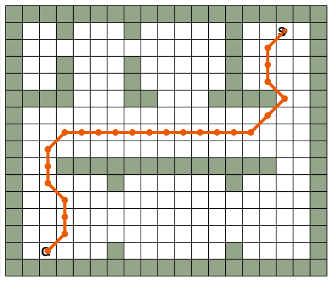


Figure 2.1: An example of planning a path through a grid, every cell is conveniently mapped to an *x* and *y* position. [1]

For example, if each cell represented $0.01m$ squared, consumed $12bytes$ each, and the resolution of the grid was $10000cells^2$ it would consume $1.12GB$ of memory. That is only operating in an area that is $100m^2$, it is easy to see how fast a robot's resources could be exhausted and so care most be taken with this approach.

2.3 Path Planning Algorithms

2.3.1 Dijkstra's Algorithm

In the classic path planner *Dijkstra's Algorithm* [2] the environment (state space) is represented as a node based graph (Figure: 2.2) defined by *G = (S, E)*, where *S* is all the possible start locations, and *E* all of the traversable edges [5]. The algorithm works by expanding outwards from the desired goal until it has computed an estimated cost for travelling from any node to the goal. It is possible to reach the desired goal from a node by iteratively traversing the edge with the lowest cost, always producing a shortest path [3].

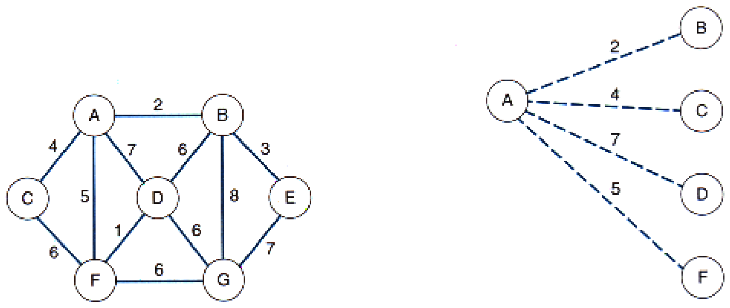
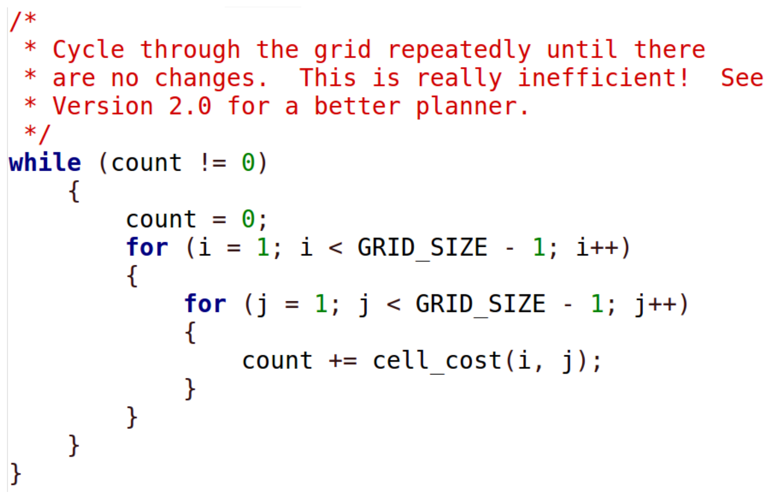


Figure 2.2: (left) An example of a graph that has been traversed by *Dijkstra's Algorithm* [2]. (right) All of the possible transitions from node *A*, traversing the edge connected to node *B* yields the lowest cost. [3]

How it computes the cost for traversing an edge is dependent on the application, in the *GridNav* [4] system (version 1.0 and 2.0) developed by Tucker Balch horizontal and vertical transitions are assigned a constant cost of *1* and diagonal transitions $\sqrt{2}$. The cost of traversing an edge (cell) is the sum of all the other nodes that must be traversed to reach the goal. In his paper Balch examines path planning techniques using a go-to-goal scenario which is the basic problem that any courier robot will have to solve.

Every cell in *GridNav* is mapped to a node in Dijkstra's graph representation as in Figure 2.7(a). Each cell contains the estimated cost of travelling from that cell to the goal, occupied cells are not traversable and are assigned a high cost of *500*. This approach works fine in the event that both sides of the grid are less than *500* cells long. If however this were not the case it is possible that the planner would treat an occupied cell as traversable.



```
/*
 * Cycle through the grid repeatedly until there
 * are no changes.  This is really inefficient!  See
 * Version 2.0 for a better planner.
 */
while (count != 0)
    {
        count = 0;
        for (i = 1; i < GRID_SIZE - 1; i++)
        {
            for (j = 1; j < GRID_SIZE - 1; j++)
            {
                count += cell_cost(i, j);
            }
        }
    }
}
```

Figure 2.3: Nested loop approach to computing the path in version *1.0*. [4]

While version *1.0* (Figure: 2.3) is simpler to code it is *extremely inefficient* [4] and in its worst case scenario it has an *O(n)* rating of n^4. This naive approach evaluates a cell multiple times, in the second planner a cell is only evaluated after the cost of one of its neighbours has been lowered. Balch proves this by testing the two planners on a *10x10* grid, the first planner evaluated each cell *10* times performing *1000* evaluations compared to the enhanced planners *85* evaluations. Running the planners against any grid proves that the second planner is always *N* times faster, where *N* is the longest side of the grid [4].

The main advantage of *Dijkstra's Algorithm* is its simplicity, Balch covers the algorithm in a novel way using real code and sample results, making it a lot easier to understand compared to a purely algorithmic approach. However *Dijkstra's Algorithm* is far from perfect and comes with a significant drawback, when a change is detected every edge cost must be recalculated [4]. These recalculations can be very expensive when using high resolution grids [5] as the planner considers every cell and not just the ones the robot is likely to travel through.

2.3.2 D* Lite

As we have just previously discussed the task of having to solve every search from scratch, as with *Dijkstra's Algorithm*, is far from practical in the dynamic environment that a courier robot is likely to operate in. Our primary concern is that such operations can be extremely computationally expensive [5], taking several seconds to complete when performed in large state spaces [10]. A better solution would be to take the previous path and repair it based on the changes to the graph, this is the approach taken by *incremental replanning algorithms*.

Two of the most widely used replanning algorithms are *D** [16] and *D* Lite* [10] (both derived from *A** [17]) which have been tested on real robots with great success (Figure: 2.4). The *A** algorithm is considered a *static* planner meaning as with *Dijkstra's Algorithm* it too must recompute every cost when a change to the graph occurs. When building a path *A** uses heuristics to focus the search between the start and goal positions [17] which generally results in far less evaluations than *Dijkstra's Algorithm*, making *A** more efficient. *D** and *D* Lite* inherit these heuristics and are not only more efficient when searching but also *dynamic* as they do not need to solve every search from scratch.

Figure 2.4: *D** and *D* Lite* have both been tested on real robot platforms with great success including the Pioneers (left) and the E-Gator (right). [5]

Since *D* Lite* is both eaiser to understand and more efficient [5][10] than *D** we will focus on it alone. *D* Lite* uses the same underlying graph representation, blocked cells are not traversable and no cost is associated with them, unknown cells are considered traversable until proven otherwise. Initially *D* Lite* computes the cost grid just like in [4] however it stores additional state information such as the status of all the successors of a cell and estimated goal distances. Using this it can quickly repair broken paths by evaluating only those cells that have been affected by the update, see Figure 2.5 for an example.

D Lite's* authors provided a number of experimental results with their work which compared the algorithms performance to *A** and *D**. These experiments were carried out in simulated environments and included a number of common operations such as the total number of vertex expansions and heap percolates [10]. Their findings showed that *D* Lite* outperformed the other two algorithms in all of the tests and in its worst case scenario it is at least as efficient as *D**. These results are significant as *D* Lite* achieves this using less operations, making it easier to understand and extend [5].

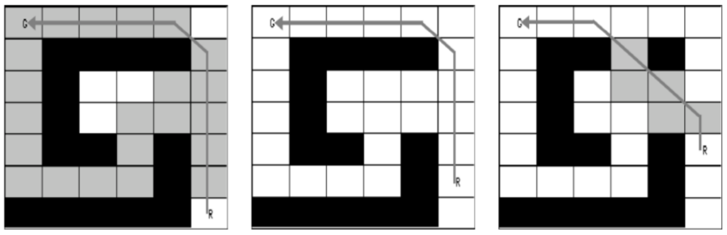


Figure 2.5: (left) The initial planning step evaluates the state of *22* cells. (center) Path produced from the initial plan. (right) When a shorter route is discovered only *5* cells need be evaluated. [5]

It is the speed and efficiency of the replanning stage that gives *D* Lite* an edge over its alternatives. It is both more efficient than *A** and *D**, and removing the need to recalculate the entire cost grid makes it considerably more efficient than *Dijkstra's Algorithm*. While this algorithm is a suitable path planner for a courier robot there is a significant disadvantage related to how it represents the world around it which we will discuss next.

2.3.3 Field D*

All of the planners that we have covered up until this point tailor an underlying graph structure to a grid, the paths they produce are therefore constrained by this limited representation. In these planners the center of every cell is connected to eight other cell centers as seen in Figure 2.7(a) and the robot must pass through one of these to progress. This limits the robot's headings to increments of $\frac{\pi}{4}$, these 'optimal' paths are often unnatural, suboptimal, and difficult to traverse in practice [6].

Consider a robot in an obstacle free environment that is facing its goal, path planning is carried out using *D* Lite*, and the robot's heading is $\frac{\pi}{8}$. The shortest path to the goal is a straight line but with planners like *D* Lite* the path contains several complicated manoeuvres, this costs a delivery robot time and energy. In Figure 2.6 the robot's heading is initially $\frac{\pi}{8}$, *D* Lite* would shift this by $\frac{\pi}{8}$, follow $\overrightarrow{e_1}$, turn $\frac{\pi}{4}$, and continue along $\overrightarrow{e_2}$ to reach g. If *D* Lite* was not limited to headings of $\frac{\pi}{4}$ the same could be achieved simply by passing through $\overrightarrow{e_0}$.

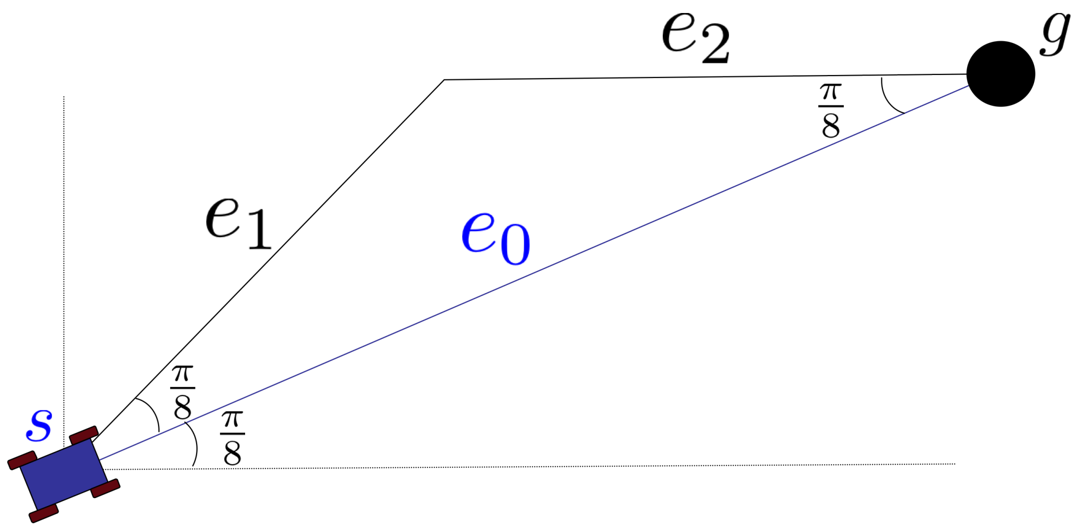


Figure 2.6: The shortest path here is along the vertex e_0, but with planners such as *D* Lite* the robot must correct its heading and pass through e_1e_2. [6]

Since courier robots typically function in human environments it is desirable that they choose what to a human would be the most natural path from point *a* to *b*. *Field D** [18] is an algorithm capable of producing these 'near natural' paths as it does not constrain a robots heading to increments of $\frac{\pi}{4}$. To achieve this *Field D** shifts the cell node from the center to the corners of each grid cell, resulting in the nodes *s* to s_8 (Figure 2.7(b)). From these nodes we can establish the following edges $\{\overrightarrow{s_1 s_2}, \overrightarrow{s_2 s_3}, \overrightarrow{s_3 s_4}, \overrightarrow{s_4 s_5}, \overrightarrow{s_5 s_6}, \overrightarrow{s_6 s_7}, \overrightarrow{s_7 s_8}, \overrightarrow{s_8 s_1}\}$ [6], and so the 'optimal' path must intersect one of these edges (Figure 2.7(c)).

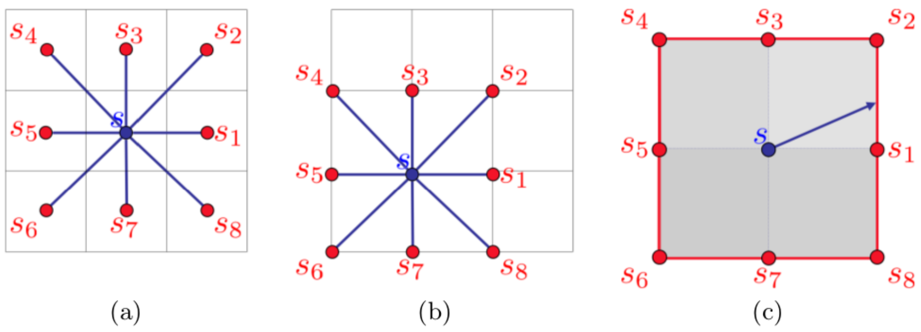


Figure 2.7: (a) The grid as a standard path planner represents it with the nodes appearing in the center. (b) How *Field D** treats cells with the nodes shifted to the corners. (c) The optimal path will intersect one of the defined edges at any heading. [6]

It was important that we first discussed *D* Lite* in Subsection 1.3.2 because in its basic unoptimised form *Field D** is simply an extension of *D* Lite*. At its core the algorithm uses *linear interpolation* when calculating the cost of traversing a node and *trigonometric maths* when computing the heading. In all of the previously mentioned path planners the cost of travelling to the goal from a node is computed as:

$$g(s) = min_{s' \in nbrs(s)} \left(c(s, s') + g(s')\right)$$

Where $nbrs(s)$ is all of the nodes neighbouring s, $c(s, s')$ is the cost of traversing s to s', and $g(s')$ is the path cost of node s' [18]. As Stentz and Ferguson explain this formula only allows for straight-line trajectories from s to a neighbouring node, which constrains the robot to headings of increment $\frac{\pi}{4}$. In *Field D** this assumption is relaxed to allow for headings that intersect a grid cell anywhere along its boundary at the point $g(s_b)$ using an approximation. In Figure 2.7(c) we see that the optimal path intersects the edge $\overrightarrow{s_1 s_2}$, to compute the cost of node s, Stentz and Ferguson use the following formula:

$$g(s_b) = yg(s_2) + (1 - y)g(s_1)$$

The path cost of $g(s_b)$ is therefore the linear combination of the costs of s_1 and s_2, where y is the distance from s_b to s_1 [18]. Stentz and Ferguson go on to discuss the theory behind *Field D** in great detail to such an extent that it cannot be fully covered here.

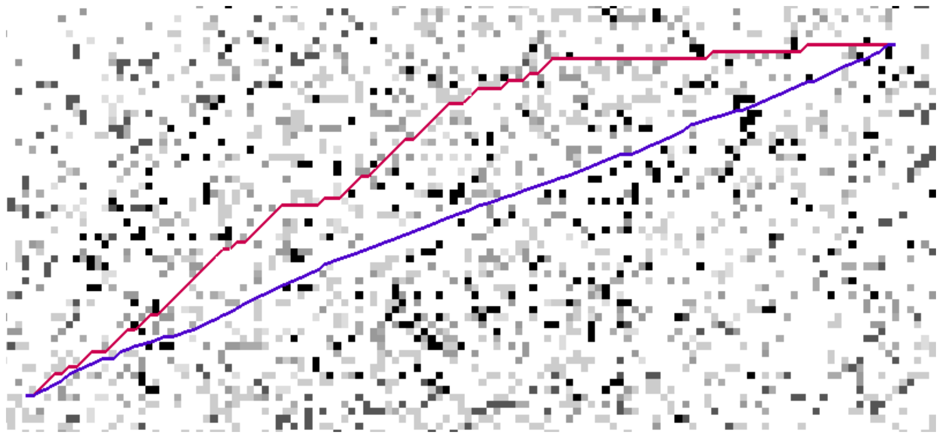


Figure 2.8: A comparison of the paths produce by *Field D** (blue) and D* Lite (red). [6]

The real value of *Field D** is the length of the paths it produces when compared to other algorithms, in Figure 2.8 the path produced by *Field D** is around *8%* shorter than *D* Lite's* result. It is also considerably smoother and contains less turning making it easier to traverse in practice. Another advantage is that it is capable of producing these smooth paths in very low resolution grids, saving on memory [6].

2.4 Discussion

During the course of this literature review we discussed courier robots in considerable detail, covering everything from their general application to specific algorithms that enable them to operate in complex environments. As we have seen path planning is an important component in courier robots because at its simplest delivering goods is all about being able to get from *a* to *b*. To achieve this we need to ensure that a courier robot can reuse the knowledge it has gained during its operation and apply it efficiently to the task at hand.

We looked at three different algorithms that can be used to produce a path through a grid, each of which have their individual strengths and weaknesses. Take *Dijkstra's Algorithm* a *greedy* algorithm that requires that the entire cost grid be recomputed every time a change occurs. Clearly this *static* planner is unsuited to the fast paced environments like the one *Kiva* operates in [19], though it is easy to implement. Then came *D* Lite* the *dynamic* replanner that is capable of dealing with broken paths or identifying short cuts efficiently, yet it too is limited not by its speed but by the headings it can deal with. Finally we arrived at *Field D**.

What is really promising about *Field D** is the scope of its application, NASA's Spirit, Opportunity, and Curiosity rovers that landed on Mars all used a flavour of *Field D** [20]. If *Field D** is capable of dealing with path planning on another planet it can then surely be applied to planning efficient routes for courier robots. As with any approach there are shortcomings to *Field D** as some of the heading changes are still unnecessary [21], but the paths are still considerably smoother than before. So as a result of undertaking this literature review this project has benefited from a clearly established context and a newly narrowed the focus, which will hopefully contribute towards its successful outcome.

Chapter 3

System Design

3.1 Requirements Specification

3.1.1 Ability to Plan a Path

The primary functional requirement for this system is the ability to plan a path through a complex environment that can then be traversed by a mobile robot. By a complex environment we refer to a 2D representation stored in some form of map that *may* represent a real world environment. The complexity of the environment is determined by the number of obstacles that the planner must successfully avoid. The success of a journey shall be measured based on the length of the planned path as a percentage when compared to other solutions.

Given a 2D map, start, and goal position, the planner using **a path planning algorithm must produce a traversable path**. A path will be made up of a series of points expressed in x and y coordinates, these can then be iteratively processed. In a case where no path to the goal exists due to obstacles the planning process must be capable of gracefully terminating which includes safely bring the robot to a halt and providing appropriate feedback.

Inputs: 2D map, start position, goal position

Outputs: A path of points OR termination feedback

3.1.2 Robot Mobility

In order to traverse the path produced as outlined in the previous requirement we need a mobile robot. **A mobile robot is one that is capable of movement in at least one axis** that will result in a change of position. For our purposes we need to be mobile in both the x and y axis. The robotic platform's drive system therefore will need to be based on a wheeled, tracked, or legged configuration which can provide the ability to travel on a level surface. It is important to note that the planning system places no restrictions on the dimensions of the robot.

However neither does it account for cases where these dimensions make traversing a path impossible due to space restrictions. The system must be able to make an intelligent decision based on the robot's current position and the next point to travel towards. The output from this requirement is an action that brings us closer towards the goal, this may involve rotating to face the next point along with travelling the calculated distance.

Inputs: current positon, next postion
Outputs: travel distance OR travel distance AND rotate

3.1.3 Data Gathering

During the execution of the plannning process the system shall generate a constant stream of data, and a requirement of this project is to capture this information for later analysis. The type of data being produced can tell us a lot about the operation including the accuracy, run time, and each planners efficiency. These results must be stored in a permanent way. The data attributes that need to be gathered include the planned path, run time, length of the path, and the total number of vertex accesses.

Inputs: planned path, run time, length of path, total vertex accesses
Outputs: time stamped directory containing datasets and plot files

3.1.4 Hardware Platform Specification

As the path planning system implemented on the Linux host is designed to be independent from the hardware specifics of the robot it is necessary that all robots communicate with the planner in a predefined way. We will discuss this mechanism under 3.3 from the software side here we will specify a generic robot that implements the interface.

In order to be compatible with the planner a robot must have software/hardware:

- Odometers - capable of establishing an (x, y) position in meters and θ in radians.
- Communications - a channel for processing text based data to and from the planner.
- External Sensors - at least one proximity sensor for detecting obstacles in the environment.

How accurate this hardware needs to be is completely dependent on the application, the planning system can only assume that any data it receives is valid. Any hardware inaccuracies i.e. drift must be dealt with at a lower level, the purpose of this approach is to decouple the planner from a robot's implementation details. This enables the planning system to be tested across a wide variety of platforms without having to account for each individual hardware configuration (Figure 3.1). It is perfectly possible that the planner and robot hardware control be implemented on the same physical system rather than externally this could be done via IP using the *localhost* address.

Motion Model

The planner will impose some restrictions on the motion model that a robot must use to be compatible with the paths it produces. It will be assumed that the robot can perform on the spot rotations, this is necessary as all of the path planning algorithms *may* produce paths that require this kind of motion. In wheeled and tracked vehicles this is typically referred to as *skid steering* [22], on the spot rotations are achieved by running the drive systems on either

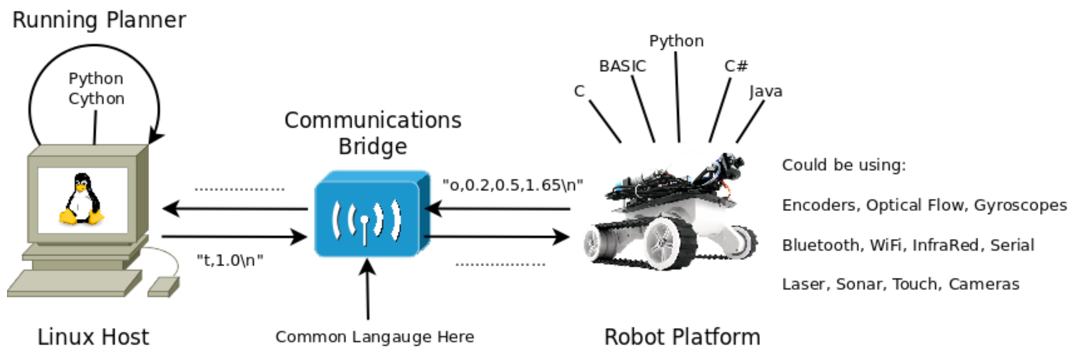


Figure 3.1: The planner on the Linux host and robot platform communicate via a common communications bridge that they both understand regardless of implementation details.

side in opposite directions, a tank is good example of this. Legged robots are also capable of performing such turns, however the heading changes can be too extreme for rack-and-pinion based agents i.e. cars and trucks.

3.2 System Modelling

3.2.1 Overview

The navigation system as a whole is built from four key components that when brought together can be used to evaluate the performance of the selected path planning algorithm. These four components are (Figure 3.2):

1. *Inputs* - operating map and a valid configuration
2. *System Core* - provides all of the functionality and the interface
3. *Pluggable Algorithms* - deal specifically with producing paths
4. *Physical Output* - actual movement of a mobile robot

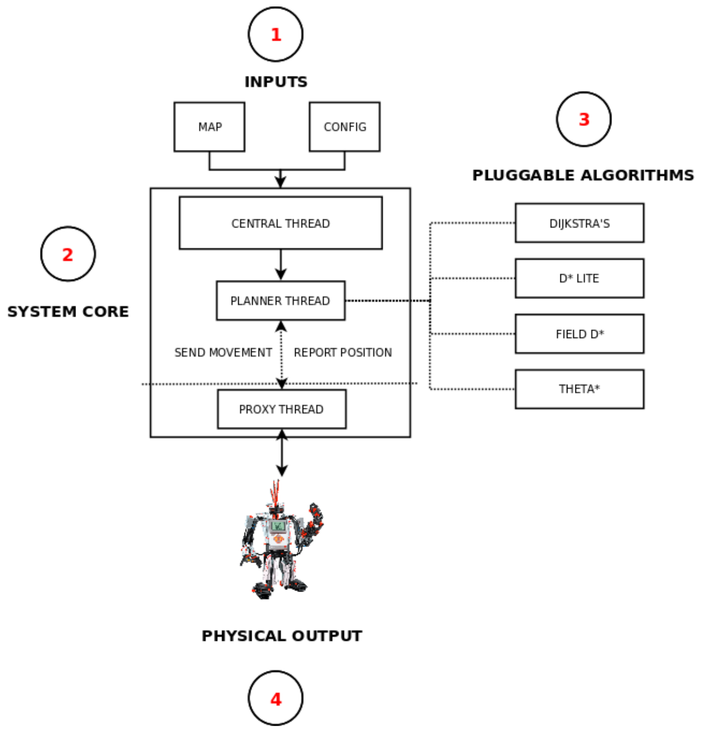


Figure 3.2: Individually each component in the system is of limited use but by bringing them together the result is a fully functional path planning system.

In essence the final component *may* not even exist as a physical robot. Instead a computer generated simulation could be used for carrying out large numbers of tests in controlled environments. Each of these components is mutually dependent on the other, we cannot start the system core without both a valid map and configuration, just in the same way without an algorithm there can be no path. This should not be mistaken for *tight coupling*, instead this is a *modular* implementation where each component interacts in an abstract way. When all of these components come together we have a path planning system not just an algorithm.

3.2.2 Threaded Architecture

The *BotNav* navigation system contains three separate threads of execution each of which perform a specific task, they are the Central, Proxy, and Planner threads. As threaded programming is notoriously difficult to understand we will cover the architecture here in detail. The task of the three threads can be summarised as follows:

- **Central** - Deals with user input, configuration, initial execution, and pre-emptive termination.
- **Proxy** - Provides a two way communications channel to a physical robot for sending commands and receiving updates.
- **Planner** - Interacts with the planning algorithm and produces a string of points that the robot iteratively navigates to.

Central Thread

The Central thread deals with user input i.e. the configuration file, based on the parameters in the configuration file it may spawn both the Proxy *and* Planner threads or just the Planner thread. In simulation mode the Central thread will spawn n Planner threads one after another, where n is the total number of traversable cells, as each Planner finishes another begins. In physical mode only one Planner thread is spawned as it is only practical to carry out one journey with a real robot. *BotNav* terminates execution after all of the Central thread's children finish. Below we can see the basic process flow for the Central thread in Figure 3.4:

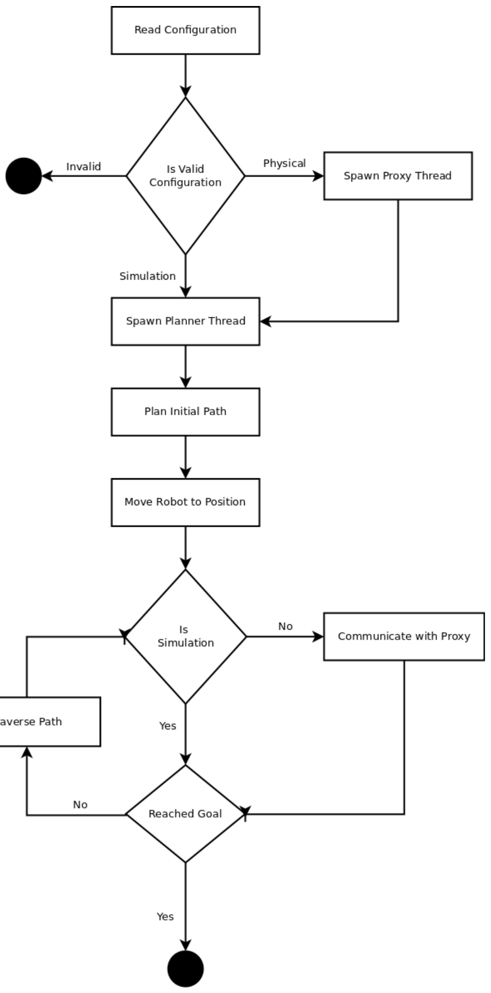


Figure 3.3: From reading the configuration file to deciding upon which child threads to spawn, the Central thread plays a very important part in the execution of the system.

Proxy Thread

Two way communication with a physical robot that implements the communications protocol outlined in 3.3 is provided via the Proxy thread. The Proxy thread's *passive* function is to process the constant stream of odometry data being generated by the robot, this data is then passed onto the system as odometry updates.

Planner Thread

It is in the execution of the Planner Thread where the must crucial work takes place, the Planner brings together a map, robot, proxy, and planning algorithm. All of this is achieved in such a way that it is possible to plug any one of the planning algorithms implemented for this project (GridNav, D* Lite, Field D*, Theta*) into it.

During a planning cycle the Planner Thread calls upon the planning algorithms *plan* routine which uses a combination of the robot's current position, a goal, and an occupancy grid to produce a path. The actual implementation behind a particular algorithm's planning functionality is hidden from the Planner Thread. If there is a valid path the result of the call to *plan* will be a trail of points which the robot can then iteratively traverse. In the case where no path exists to the goal execution is immediately terminated.

Once a path has been generated and returned it is processed as follows:

```
while we are not within 0.7 cells of the goal in both x and y:
  # Scan the immediate area.
  robot.scan()

  # Check for a change in the map.
  if there is a change:
    path = algorithm.plan()

  # Pop the next point from the current path.
  if path length is 0: # Make sure there is a point.
    break

  point = path.pop() # Get the next point.
  robot.go(point.x, point.y)
```

```
while robot is travelling:
  # Busy wait.

calculate x difference to goal
calculate y difference to goal
```

3.3 Communications Protocol

This section briefly discusses the communications protocol that a physical hardware robot must implement in order to be compatible with the planning system. The protocol covered here is an extension of the simple communications mechanism from [22] that was used to drive a tele-operated mapping robot. It is based on plain text strings terminated by new line characters ('\n') and was designed with simplicity in mind. All commands are synchronous in behaviour and cannot be completed until the appropriate response has been returned. The full specification is available in the Appendices under section **??**.

3.3.1 Travelling a Predefined Distance

To instruct the robot to travel forward for a straight line distance of 1.3 metres the command is simply:

t,1.3\n

The 't' character literally stands for *"travel"*, the distance specified as a decimal number comes immediately after the ',' character and can be any **positive** value, it must be terminated by a newline character. When the robot has completed the command it simply responses with the string:

Travelled\n

This informs the planner that the robot is no longer in motion and that the next planning step can now be processed. It is possible for a robot's local planner or obstacle avoidance system to abort the action by simply returning this response at any stage in the journey due to unforeseen obstacles. In these cases the robot should return the location of the obstacle(s) in its next scan report.

3.3.2 Rotating to a Given Heading

Valid headings run from 0 to 6.27 radians in a circular fashion, they increase from right to left around the circle. To instruct the robot to face a heading of 1.57 radians (90°) the command is:

r,1.57\n

There is no explicit return value instead the planner waits until the odometry reports from the robot are within 0.02 radians (1°) of the requested heading, the robot is then instructed to stop if it has not already done so. The planner uses both the travel and rotation commands in conjunction to navigate to specific points.

First the robot is commanded to face a given point then the distance between its position and the target are calculated, secondly the robot is sent a travel command containing this information. Using this mechanism the robot is able to navigate complex environments with only the basic knowledge it requires, the rest is done by the planner.

3.4 Programming Platforms

3.4.1 Linux Architecture

The Linux kernel is to date driving an increasing number of commercial and research robotic platforms all over the world including Google's Self Car Project, Nao, EV3 Lego Mindstorms, and the horde of robots that use ROS [9, 8, 7, 23]. Linux is unique as it is mature, stable, customisable, embeddable, and freely available, making it a suitable candidate for powering the robot revolution.

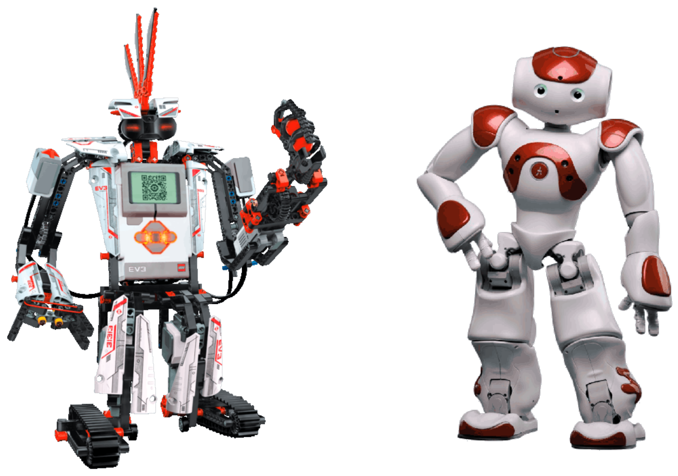


Figure 3.4: Both the EV3 Lego Mindstorms (left) and the Nao (right) run a distribution of the Linux Kernel. [7, 8]

Considering all of the points previously mentioned Linux was picked as the target programming platform over any other alternatives. Linux's embeddable nature is important to this project as at least one of the physical robots may be equipped with the *Raspberry Pi* a credit card size computer capable of running Linux. The system will be tested on two distributions of Linux, one embedded on the robot(s), the other connected externally:

- Raspbian (Embedded)
- Ubuntu Linux 14.10 (External)

3.4.2 Python3

Python version 3 (Python3) has been selected as the most appropriate core programming language for the implementation stage for the following reasons:

- Development Speed - Python is an interpreted bytecode language which is compiled "on the fly" eliminating lengthy build overheads.
- Interactive Shell - Python's shell facilitates interactive programming which lends itself to test driven coding.
- Python and Robotics - already widely used in the robotics field, Sebastian Thurn's Udacity program is done purely using Python [24].

Python's syntax is also extremely lenient when compared to compiled languages such as C where every variable must explicitly declares its type before it can be used, making the equivalent Python code more visually compact. The implementation could have been carried out using C, C++, Java, or C$\sharp$ as the author already has the experience in those languages, however using Python instead presents a unique learning opportunity.

3.4.3 Cython Modules

A major drawback of interpreted languages is the speed of their execution when compared to compiled languages, since Python is implemented using a virtual machine it will nearly always be slower than pure machine code. It is important to take this issue into account at design time as the planning algorithms are computationally complex and as the state space increases Python will become less adapt at handling it compared to a compiled language.

Fortunately as Python is wrote in C it can also be extend using C. Implementing the planning algorithms as Cython modules shall significantly increase their performance while still enabling the rest of the system to use Python. At the time of writing GCC (GNU C Compiler) version 4.9.1 has been selected for compilation purposes.

3.5 Source Control

This section covers how the project was managed, the source control system used, frequency off commits, total commits, and any branching.

3.5.1 Git

Throughout the projects development all work undertaken including everything from the code base, poster, modelling, and thesis was managed and controlled using the Git source control tool. This decision was taken at design time as it came with a number of distinct advantages such as providing a clear work history and the ability to roll back changes if the need arose. Another important reason for choosing Git over other systems is that it maintains a local copy in addition to the working head in the remote repository [25]. An open repository was set-up and maintained in the cloud at the GitHub BotNav.

Chapter 4

Core Implementation

4.1 Building the Project

4.1.1 Obtaining the Source

The latest version of the project's source code can be checked out via *git* using:

git clone https://github.com/swordmaster2k/botnav.git

Or downloaded as a ZIP file from `https://github.com/swordmaster2k/botnav`.

Alternatively the most up to date version at the time of printing is available on the CD at the front of this thesis.

4.1.2 Compiling the D* Lite Cython Module

The planning algorithm D* Lite must be compiled as a Cython module, the original source code was provided by Maxim Likhachev of CMU and Sven Koenig of USC in C. It has been modified to make it compatible with the core Python system using Cython, as Python is implemented in C [**?**] it is inherently compatible with the sample of D* Lite that is provided by its authors.

To build D* Lite you will need Python3.4, the Python3.4 headers, gcc, and make. It **must** be built for each platform on which it will execute as C compiles to machine code making it *target dependent*. From a terminal navigate to the source code directory *BotNav/algorithm/dstarlite_build/*.

The make file contains two build rules:

1. *make* - which builds the module *dstarlite_c.so*
2. *make clean* - cleans all previous output files from the build process

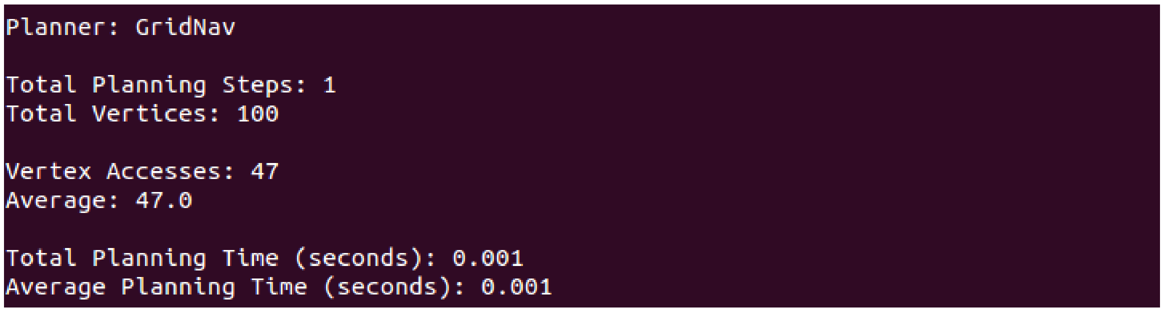
Once the module file *dstarlite_c.so* has been successfully built for the target platform it can simply be dropped into the parent directory *BotNav/algorithm/*. The Python source code contains a reference to the module and will automatically link it in at execution time.

4.1.3 Running it in Python3

By default the project is set-up to run a sample simulation with *sample.map* using the Grid-Nav path planning algorithm. It will output the results of its run to the *BotNav/maps/output/* directory and is configured to compute a path from every traversable cell to the goal. To run it simply navigate to the path containing *tester.py* and type:

```
python3 tester.py config.botnav
```

The argument passed to *tester.py* is the path to the default configuration file, the contents of which will be discussed later in this chapter. After running the above command in a terminal the result shall look similar to Figure 4.2.



```
Planner: GridNav

Total Planning Steps: 1
Total Vertices: 100

Vertex Accesses: 47
Average: 47.0

Total Planning Time (seconds): 0.001
Average Planning Time (seconds): 0.001
```

Figure 4.1: An example of the type of output generated after running GridNav over *sample.map* in simulation mode.

4.2 Running Simulations

Simulations provide an easy means of testing each planning algorithm in controllable environments, it speeds up the testing process immeasurably, and provides reproduce-able results. The most powerful feature of simulations is the ability to plan a path from every free cell in the environment, the result of each traversal is placed into a separate timestamped folder. This data can then be easily mined and analysed in order to gauge how each algorithm performed for a particular scenario.

4.2.1 Configuration

When carrying out simulated runs three parameters must be set in the corresponding configuration file they are *map*, *mode*, and *planner*. Below is an example of the configuration required to run a simulated trial using D* Lite:

```
map=maps/simple.map
planner=d_star_lite
mode=simulated
```

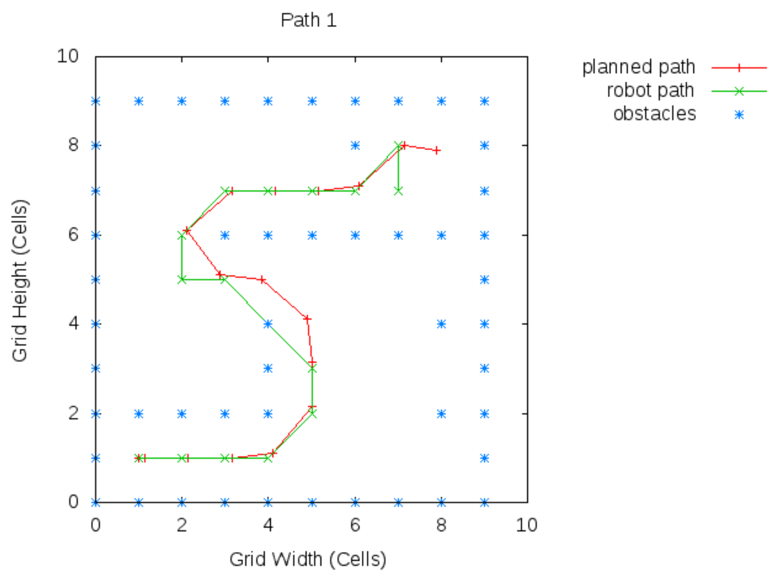
The most important parameter setting here is *mode* which is set to simulated. At run time this informs the *Tester* class that we want to run an experimental simulation across all traversable cells and that we do not need a communications channel via *Proxy*. In simulation mode all of the instructions are invoked on the "dummy" robot class *SimulatedRobot*, the method *go_to(self, x, y)* simply takes the x and y coordinates, introduces some random drift, and assigns them:

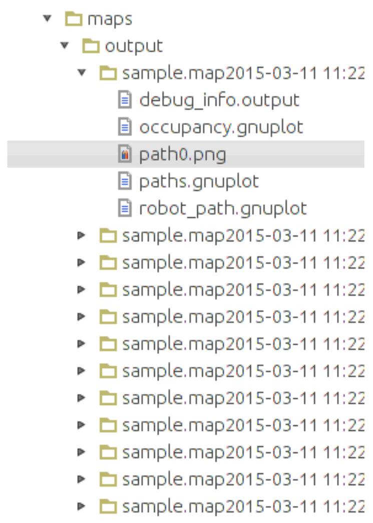
```
def go_to(self, x, y):
    # Introduce a little uncertainty.
    self.x = (x + random.uniform(-0.2, 0.2)) * self.cell_size
    self.y = (y + random.uniform(-0.2, 0.2)) * self.cell_size

    self.trail.append([self.get_cell_x(), self.get_cell_y()])

    self.state = "Travelled"
```

4.2.2 Sample Output





```
▾ maps
  ▾ output
    ▾ sample.map2015-03-11 11:22
        debug_info.output
        occupancy.gnuplot
        path0.png
        paths.gnuplot
        robot_path.gnuplot
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
    ▸ sample.map2015-03-11 11:22
```

Figure 4.2: A plot file for the first path from the sample map generated by *gnuplot* (left), and the contents of the output folder after the run. Note that each folder is timestamped (right).

4.3 Using a Real Bot

The real test for any path planning algorithm is its practical effectiveness [6] and the only way to gauge this is using a physical robotic platform i.e."a Real Bot". The simulations that we have performed here are very limited in nature as they do not take into account any variability in the mechanics of the robot. While it is perfectly possible to model variations such as drift, odometry error, friction, and battery drain in a simulation it is far more practical to simply use a physical platform.

There are a number of factors that we must consider when using a physical robot, to begin with we will need some form of communications mechanism be it wired (USB, Ethernet, Serial) or wireless (ZigBee, WiFi, Bluetooth). It is through this medium that the *Proxy* class will process the data to and from the robot. Then there is drift, occurrences such as wheel slippage can lead to the robot veering off course, we will assume that this issue is dealt with at a lower level than the path planning system.

4.3.1 Configuration

The contents of the configuration file for a hardware robot depends on the communications medium being employed. At present the system supports three forms of communication Bluetooth, IP, and Serial. Before any of these can be used the run mode must set to *physical*. Below is an example configuration using Bluetooth which requires the MAC address of the device and a port number:

```
map=maps/sample.map
planner=grid_nav
mode=physical
connection=bluetooth
mac=00:00:12:06:56:83
port=0x1001
```

For IP based configurations an *ip*, *mac*, and *port* setting will be required. Serial based connections need only a *baud*, and *port*, examples of both are available in the comments of the default configuration file:

```
# if connection == bluetooth
#     mac=
#     port=
# elif connection == ip
#     ip=
#     mac=
#     port=
# elif connection == serial
#     baud=
#     port=
```

4.4 How the Planner Works

As we have already seen our *Planner* is implemented on its own separate thread which is designed to handle the bulk of the processing required to get our mobile robot from point a to b. All of this work is carried out within the *run* method which is invoked when the thread is started. Below is a snippet of the code from the start of this method:

```
'''
Step 1: Plan.
'''
self.algorithm.plan()

# Write the state after first planning step.
self.write_state()
```

```
# Append a copy of the path to our paths record.
paths.append(self.algorithm.path[:])

# Stick the starting position of the robot into
# the first path.
paths[0].insert(0, [self.robot.get_cell_x(),
   self.robot.get_cell_y()])

# Calculate our initial distance from the goal.
[output omitted]

# Write some debugging info.
self.write_debug_info()
```

During the planning process the *Planner* goes through a number of steps, part of the first step is outlined in the above code snippet. After the initial planning step that is used to evaluate if a path to the goal exists the planner immediately writes the state to our output files. Then some additions to the path collection are made including the robot's start position, the goal distance is calculated and checked, and lastly debugging information is recorded. This is just the first of five steps involved during the planning process this particular step only runs once per test. In the next section we will discuss all five steps together in 4.4.1.

4.4.1 Five Simple Steps

The planning process presented in this work was inspired by [4]. In [4] T. Balch established a simple methodology that enabled a mobile robot equipped with grid based planner to successfully navigate towards its goal in an iterative manner using seven steps. Those steps can be seen on page 7 of his paper. Our planning process was originally derived from these steps and later simplified by reducing it from seven steps to five:

1. Plan.
2. Scan the immediate area for obstacles and free space.
3. Update the map if necessary.
    (a) Recompute the plan if necessary.
4. Pop the next point from the current path and travel.
5. Go to 2.

In the original version the initialisation of variables and the reading of the map file were included as a step, this is not included in the *Planner* directly so it was removed. Also step 5 from [4] became the optional step 3(a) as this step will only ever be taken if the condition for step 3 becomes *true*. This process will iteratively execute from step 2 through 5 until either the goal is reached *or* when no path can be found. Essentially all of the work in this entire project evolved from this methodology. By reading over [4] we can gain a fundamental understanding of everything that grid based path planning involves from start to finish.

4.4.2 Abstracting Away from the Algorithm

Art the heart of the object oriented patterns used in this project is *abstraction*, a technique that enables a developer to generalise the functionality of a class through abstract inheritance. Through this process the complexity of an object can be reduced and its efficiency greatly increased [**?**]. It has been used extensively in the implementation of the path planning algorithms which all inherit from the base class *AbstractAlgorithm*. From this class each algorithm inherits a series of default behaviours some of which are listed here:

- plan(self) -
- pop_next_point(self) -
- print_debug(self, stream) -

The real advantage to using abstraction here is the *pluggable* nature it provides. Every algorithm that inherits from the *AbstractAlgorithm* base class implements a common set of behaviours and properties. Regardless of which algorithm is currently being used for planning we simply access the same behaviour or property (see Figure 4.3). The implementation details of each algorithm is hidden from the *Planner* whom only cares about the results.

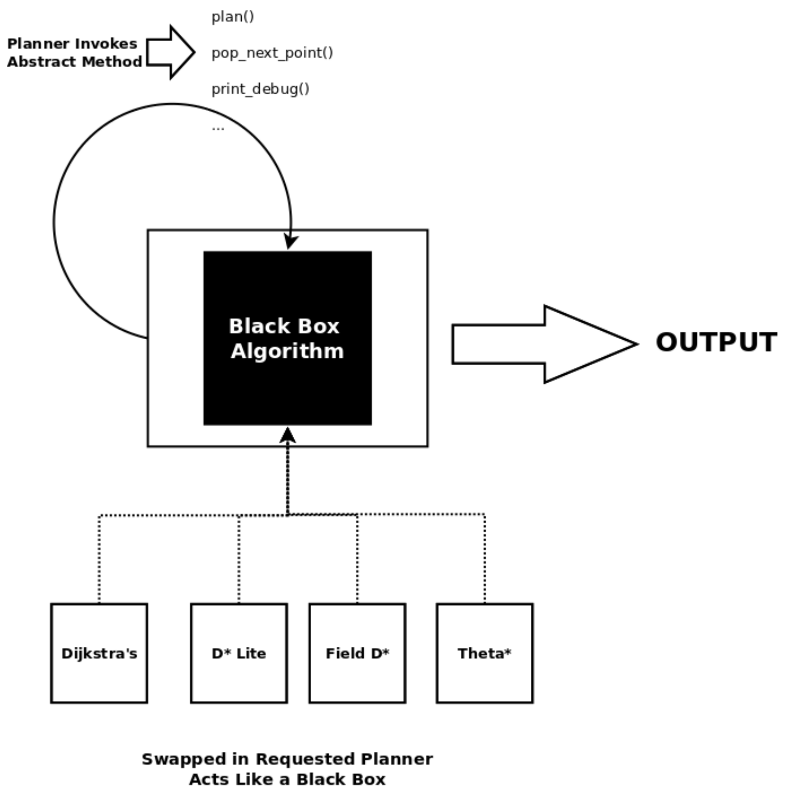
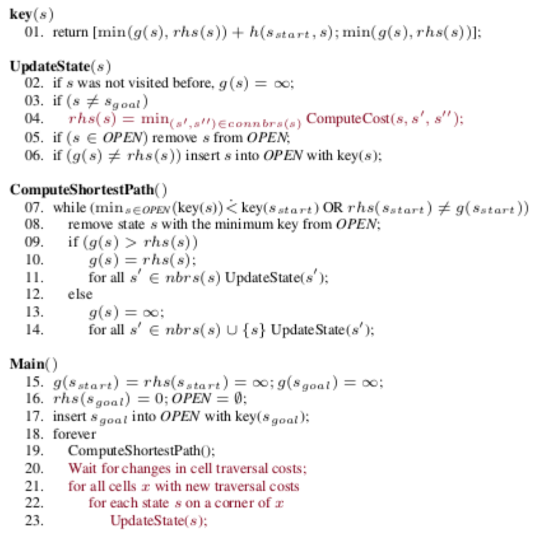


Figure 4.3: The current algorithm acts as a black box to the *Planner* whom calls a set of public abstract methods. Any planning algorithm can be swapped in for testing without having to account for its implementation specifics.

4.5 Open Field D*

For implementation purposes we modified sample implementations of Dijkstra's Algorithm, D* Lite, Theta*, and then plugged them into the path planning system. This made perfect sense as we saw no point in reinventing the wheel and coding all of these algorithms from scratch. However there was one algorithm for which no *public* implementation existed. Field D* has provided advance path planning capabilities to some of the most sophisticated robots ever made including NASA's Curiosity rover and the GDRS XUV [6]. Its authors claim that Field D* is an efficient planning and replanning algorithm that can produce smooth paths costing on average 4% less than traditional planners [6].

Verifying the claim made by the authors of Field D* requires a working implementation, the problem is that all existing work on Field D* has been carried out behind closed doors mostly at NASA. To the best of our knowledge the work presented here contains the first openly available version of Field D* nicknamed appropriately *Open Field D**. In order to derive our version of this novel path planner we based it on the basic D* Lite version presented on the 8th page of [6] which can be seen below:



```
key($s$)
01. return [$\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))$];

UpdateState($s$)
02. if $s$ was not visited before, $g(s) = \infty$;
03. if ($s \neq s_{goal}$)
04.     $rhs(s) = \min_{(s', s'') \in connbrs(s)}$ ComputeCost($s, s', s''$);
05. if ($s \in OPEN$) remove $s$ from $OPEN$;
06. if ($g(s) \neq rhs(s)$) insert $s$ into $OPEN$ with key($s$);

ComputeShortestPath()
07. while ($\min_{s \in OPEN}$(key($s$)) $\dot{<}$ key($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
08.     remove state $s$ with the minimum key from $OPEN$;
09.     if ($g(s) > rhs(s)$)
10.         $g(s) = rhs(s)$;
11.         for all $s' \in nbrs(s)$ UpdateState($s'$);
12.     else
13.         $g(s) = \infty$;
14.         for all $s' \in nbrs(s) \cup \{s\}$ UpdateState($s'$);

Main()
15. $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty$;
16. $rhs(s_{goal}) = 0; OPEN = \emptyset$;
17. insert $s_{goal}$ into $OPEN$ with key($s_{goal}$);
18. forever
19.     ComputeShortestPath();
20.     Wait for changes in cell traversal costs;
21.     for all cells $x$ with new traversal costs
22.         for each state $s$ on a corner of $x$
23.             UpdateState($s$);
```

Figure 4.4:

4.5.1 Basic Implementation

The basic version of Field D* differs little from the original specification of D* Lite except when it comes to updates. In the literature review we mentioned how Field D* shifts nodes from cell centres to cell edges which can be seen in Figure 2.7. Equipped with this modification Field D* can produce smooth paths using linear interpolation techniques that enable a path to intersect any point on the edge of a cell. Calculating the cost of a node requires all its consecutive neighbours (node pairs) that represent the neighbouring edges. This operation can be seen on line 4 in Figure 4.4. After a detailed analysis of the *pseudo code* we came up with the following solution in Python:

```
def get_consecutive_neighbours(self, s):
      consecutive_neighbours = []

      for i in range(len(self.NEIGHBOUR_DIRECTIONS)):
         try:
            if i == len(self.NEIGHBOUR_DIRECTIONS) - 1: # Edge s8
               -> s1
               x1 = s.x + self.NEIGHBOUR_DIRECTIONS[i][0]
               y1 = s.y + self.NEIGHBOUR_DIRECTIONS[i][1]
               x2 = s.x + self.NEIGHBOUR_DIRECTIONS[0][0]
               y2 = s.y + self.NEIGHBOUR_DIRECTIONS[0][1]

               if x1 > -1 and y1 > -1 and x2 > -1 and y2 > -1:
                  consecutive_neighbours.append(
                     (self.nodes[s.x +
                        self.NEIGHBOUR_DIRECTIONS[i][0]][s.y +
                        self.NEIGHBOUR_DIRECTIONS[i][1]],
                      self.nodes[s.x +
                         self.NEIGHBOUR_DIRECTIONS[0][0]][s.y +
                         self.NEIGHBOUR_DIRECTIONS[0][1]]))
            else: # All other edges.
```

```
                    x1 = s.x + self.NEIGHBOUR_DIRECTIONS[i][0]
                    y1 = s.y + self.NEIGHBOUR_DIRECTIONS[i][1]
                    x2 = s.x + self.NEIGHBOUR_DIRECTIONS[i + 1][0]
                    y2 = s.y + self.NEIGHBOUR_DIRECTIONS[i + 1][1]

                    if x1 > -1 and y1 > -1 and x2 > -1 and y2 > -1:
                        consecutive_neighbours.append(
                            (self.nodes[s.x +
                                self.NEIGHBOUR_DIRECTIONS[i][0]][s.y +
                                self.NEIGHBOUR_DIRECTIONS[i][1]],
                             self.nodes[s.x + self.NEIGHBOUR_DIRECTIONS[i
                                + 1][0]][
                                s.y + self.NEIGHBOUR_DIRECTIONS[i +
                                    1][1]]))

            except IndexError:
                pass
            except AttributeError:
                pass

        return consecutive_neighbours
```

Essentially what we are doing in the above code sample is iteratively processing all the neighbouring node pairs in a clockwise direction to get the sequence $\{\overrightarrow{s_1s_2}, \overrightarrow{s_2s_3}, \overrightarrow{s_3s_4}, \overrightarrow{s_4s_5}, \overrightarrow{s_5s_6}, \overrightarrow{s_6s_7}, \overrightarrow{s_7s_8}, \overrightarrow{s_8s_1}\}$. We can then compute the cost of node *s* by selecting the lowest costing neighbour pair and calling the following function:

```
def compute_cost(self, s, sa, sb):
        self.vertex_accesses += 1
        s.evaluations += 1

        [output omitted]
```

```
    # Map mid_x and mid_y to a cell cost, if the x and y is out
      of
    # bounds c == LARGE.
    c = self.get_cell_cost(math.floor(mid_x), math.floor(mid_y))

  [output omitted]

    if min(c, b) == self.BIG_COST:
        vs = self.BIG_COST
    elif s1.g <= s2.g:
        vs = min(c, b) + s1.g
    else:
        f = s1.g - s2.g

        if f <= b:
            if c <= f:
                vs = c * SQRT_2 + s2.g
            else:
                y = min(f / (math.sqrt(c ** 2 - f ** 2)), 1)
                vs = c * math.sqrt(1 + y ** 2) + f * (1 - y) + s2.g
        else:
            if c <= b:
                vs = c * SQRT_2 + s2.g
            else:
                x = 1 - min(b / (math.sqrt(c ** 2 - b ** 2)), 1)
                vs = c * math.sqrt(1 + ((1 - x) ** 2)) + (b * x) +
                    s2.g

    if vs > self.BIG_COST:
        vs = self.BIG_COST

    return round(vs, 3)
```

Cost calculation in Field D* is based on *g-values* and also the estimated traversal cost of the selected neighbouring cell which can vary. The above Python code is a direct implementation of the mathematical theory presented on page 7 of [6]. It is not possible to cover the algorithm in-depth here due to its length, further reference should be should be sought from [6, 18] and this project's source code.

4.5.2 Issues with Field D*

The most challenging part of this project was by far the implementation of Field D*. Little information on the algorithm exists beyond what is included in [6, 18] despite the fact that it dates back to 2007 [6]. Thankfully Field D* was originally derived from D* Lite [10] and its basic version simply extend its. With that said we encountered a large number of ambiguities that required us to mathematically reverse engineer some of its specification. One aspect that remains unclear is path extraction which is not covered in any level of detail by the authors.

Every node in the Field D* cost grid can be seen as a sample point in a continuous space, when extracting a path we select the lowest costing edge. The cost of an edge is the sum of both connecting nodes, our optimal path intersects this edge at some point (x, y). Exactly how this point is obtained is never clearly defined. In our implementation we extract our new point based on the *rhs-values* of the nodes s_1 and s_2, which we then shift based on the direction of travel:

```
f = s1.rhs - s2.rhs

if f > 1:
  f = 1
elif f < -1:
  f = -1

x_difference = s1.x - s2.x
```

```
y_difference = s1.y - s2.y
x_shift = 0
y_shift = 0

if x_difference != 0:
  if x_difference < 0:
    x_shift = f
  else:
    x_shift = -f
else:
  if y_difference < 0:
    y_shift = f
  else:
    y_shift = -f

if x_shift != 0:
  self.s_start.x += x_shift
  self.s_start.y = s2.y
elif y_shift != 0:
  self.s_start.x = s2.x
```

This method works well for most of the cases that we tested it against, however it breaks down in complex environments that are obstacle dense. While we have not been able to solve this part of Field D*, the rest of the algorithm functions as expected. It is possible to produce smooth and low costing paths using our implementation for 80% of the cases that we tried. Hopefully overtime the work can be improved and a fully functioning version of Field D* will exist outside of NASA.

Chapter 5

Testing

5.1 Test Cases

In this section we will establish our testing methodology that shall be used to answer our main research question. The purpose of these tests is to discover what is more important in path planning for mobile robots, computation speed, or shorter path lengths. To achieve this we have defined two test cases that look at:

- Traversal time based on the path length.
- Estimated computation time for vertex accesses.

Examining what proportion the computation time takes when compared to the travel time will help us answer our main research question. The most efficient planner will be selected based on the combined traversal and computation time. Before we carry out these tests we must be aware of potential biased scenarios, for example in a maze like environment where movement is completely restricted to headings of $\frac{\pi}{4}$ all planners shall have similar performances. Another case is where our path travels through wide open spaces containing large numbers of free cells. Planners that focus their search using *heuristics* [**?**] have a distinct advantage here (D* Lite, Field D*, Theta*).

5.1.1 Test Case 1: Path Length

Each of the planners produce one key output and that is a path from our start position to the goal. A property that all of these paths share in common is their physical length in metres. The goal of all of these planners is to compute the shortest path from the robot's current position to the goal. In order to determine on average which of the planners produces the best result for a given scenario we will examine the time it takes to traverse.

Based on a large number of evaluations from every possible starting position to the goal for a given map it will be possible to say which planning algorithm on average produces the shortest path. To make this test fair it shall be based on a theoretical mobile robot with the following properties:

- Average Speed: $0.6m/s$
- Turning Speed: $1rad/s$

These figures were gathered from the performance of the DFRobot Cherokey 4WD platform during path traversals. It is important to note here that we are assigning a time penalty for turning which requires deceleration. Calculating the cost using this theoretical model reduces the error rate in our results as we do not have to deal with drift or other physical properties that affect real robot platforms. It also reduces the time it takes to test as we can now perform thousands of simulated runs at once.

Expected Result: based on the claims of the authors of Field D* and Theta* we expect that the planners capable of dealing with headings other than $\frac{\pi}{4}$ will perform better.

Cost Calculation

The time it takes to traverse a path is then based on two inputs:

- Path length in metres.
- Total heading change in radians.

We can extract these two inputs from our raw path of points. The path length is simply the sum of all the distances between every point using the coordinate geometry distance formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Which can be implemented in *psuedocode* as:

```
length = 0

for each point in path:
  next_point = point.next()

  if next_point is not null:
    length += math.sqrt((next_point.x - point.x) ^ 2 +
       (next_point.x - point.x) ^ 2)
```

The total number of headings changes is just the sum of the difference between our current heading at a point and the change in direction required to face the new point. Our equation for calculating the cost of traversing a path is then:

$$time = path\_length \times 0.6 + total\_rotations \times 1.0$$

Example

Given: path of $3m$ containing a total heading change of $1rad$.

Based on the equation we just defined we can get the path cost by substitution:

$$\Rightarrow time = path\_length \times 0.6 + total\_rotations \times 1.0$$
$$\Rightarrow time = (3) \times 0.6 + (1) \times 1.0$$
$$\Rightarrow time = 5 + 1$$
$$\Rightarrow time = 6s$$

We can use the result from this simple computation during our comparative analysis to determine which planner is producing the lowest costing path. The result can also help in calculating the resources required to reach the goal *i.e.* battery drain, as time spent moving costs a significant amount of energy.

5.1.2 Test Case 2: Vertex Accesses

In order to properly address our main research question we must look at one other function that all the path planners perform during the planning process. When it comes to path planning algorithms the highest costing operation is anything that involves accessing a *vertex*. Depending on the planners representation a vertex can either be a cell or an edge, vertices are accessed during cost calculation, updates, and traversals. It makes sense then to take into account the time spent accessing vertices while the path is being generated.

While raw execution time may seem like a more obvious measure, the problem with it is that it depends on the underlying computer architecture. If we were to base our tests on the time the process spends in the processor they would be *machine dependent* and difficult to reproduce [18]. Instead we will adopt the same approach as in Test Case 1.

Given the number of vertices accessed during the planning we can assign a constant time cost for every access independently of the processor. This works in practice as all of the planners use simple mathematical calculations when working with vertices. This allows us to safely establish a theoretical execution time. The value of this constant is not important as long as it is realistic and consistent across all tests:

- Cost per access: 100μs

Expected: it is expected the planners that focus their search will access less vertices and therefore use less computation time.

Cost Calculation

Calculating the time cost for accessing vertices requires the following formula:

$$time = total\_accesses \times cost\_per\_access$$

Example

Given: the planner evaluated 400 of 2500 vertices for Test Case 1.

$$\Rightarrow time = total\_accesses \times cost\_per\_access$$
$$\Rightarrow time = (400) \times 100$$
$$\Rightarrow time = 0.04s$$

The total time spent for both Test Case 1 and 2 is 6.04s, as a percentage the time spent planning is 1.51%. What we shall be looking to establish during the testing phase is which is more proportionally important fast planning or a shorter path.

Bibliography

[1] W. Burgard, "Introduction to mobile robotics robot motion planning." Online Lecture Notes.

[2] E. Dijkstra, "A note on two problems in connexion with graphs," tech. rep., Numerische Mathematik, 1959.

[3] A. Hensman, "Data structures and algorithms bsc. in computing semester 5 lecture 8." Graph Applications.

[4] T. Balch, "Grid-based navigation for mobile robots," tech. rep., Georgia Tech, 1995.

[5] D. Ferguson and et al., "A guide to heuristic-based path planning," tech. rep., Carnegie Mellon University, 2005.

[6] D. Ferguson and A. Stentz, "The field d* algorithm for improved path planning and replanning in uniform and non-uniform cost environments," tech. rep., Carnegie Mellon University, 2007.

[7] Lego, "Mindstorms ev3." Online, 2015. http://www.lego.com/en-us/mindstorms/products/31313-mindstorms-ev3.

[8] Aldebaran, "Who is nao?." Online, 2015. https://www.aldebaran.com/en/humanoid-robot/nao-robot.

[9] W. Knight, "Driveless cars are further away than you think," tech. rep., MIT, October 2013.

[10] S. Koenig and M. Likhachev, "D* lite," in *AAAI-02 Proceedings*, pp. 476–483, AAAI, 2002.

[11] G. Mies, "Military robots of the present and the future," in *AARMS*, vol. 9, pp. 125–137, AARMS, May 2010.

[12] C. Ray, F. Mondada, and R. Siegwart, "What do people expect from robots?," tech. rep., IEEE, 2008.

[13] D. Fagnant and K. Kockelman, "Preparing a nation for autonomous vehicles," tech. rep., Eno, October 2013.

[14] C. Frey and M. Osborne, "The future of employment: How susceptible are jobs to computerisation?," tech. rep., University of Oxford, September 2013.

[15] U. Nehmzow, *Mobile Robotics A Practical Introduction*, ch. 5, pp. 95–108. Springer, 2nd ed., 2003.

[16] A. Stentz, "A complete navigation system for goal acquisition in unknown environments.," tech. rep., Autonomous Robots 2, 1995.

[17] P. Hart, N. Nilsson, and B. Rafael, "A formal basis for the heuristic determination of mimimum cost paths.," in *IEEE trans. Sys. Sci. and Cyb.*, vol. 4, pp. 100–104, IEEE, 1968.

[18] D. Ferguson and A. Stentz, "Field d*: An interpolation-based path planner and replanner," tech. rep., Carnegie Mellon University, 2007.

[19] J. Letzing, "Amazon adds that robotic touch." Online, March 2012. http://online.wsj.com/news/articles/SB10001424052702304724404577291903244796214.

[20] T. Stentz, "Cmus path planning software lets curiosity find its way." Online, 2014.

[21] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," tech. rep., University of Southern California, 2009.

[22] J. Daly, P. Butterly, and L. Morrish, “Implementing odometry and slam algorithms on a raspberry pi to drive a rover,” tech. rep., Institute of Technology Blanchardstown, 2014.

[23] O. S. R. Foundation, “About ros.” Online, 2015. http://http://www.ros.org/about-ros/.

[24] S. Thrun, “Programming a robotic car.” Online, 2015. https://www.udacity.com/course/cs373.

[25] S. Chacon, *Pro Git*. Scott Chacon, 2010.