DOCUMENT STRUCTURE

AN ANALYSIS OF THE FIELD D* ALGORITHM 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
February 2, 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: February 2, 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 [7].

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* [8] 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 [9]. 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* [10]. 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 [7]. 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 [11].

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 [12].

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 [13]:

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* [13] 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].

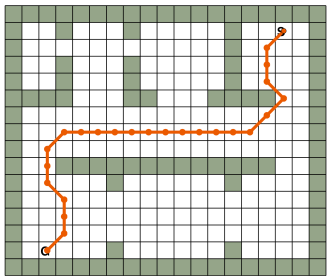

Figure 2.1: An example of planning a path through a grid, every cell is conveniently mapped to an *x* and *y* position. [1]

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].

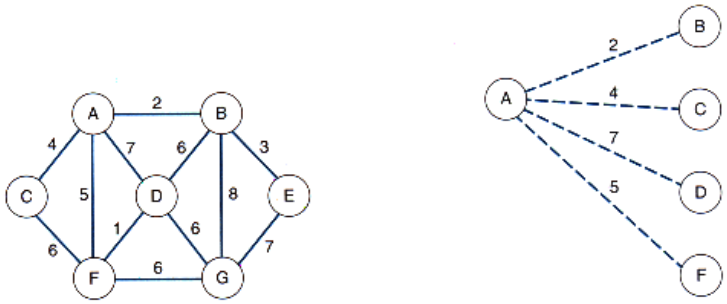

Figure 2.2: (left) An example of a graph that has been traversed by *Dijkstra's Algorithm* [2]. (right) All of the possible transitions from node A, traversing the edge connected to node B yields the lowest cost. [3]

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 [8]. 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** [14] and *D* Lite* [8] (both derived from *A** [15]) 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 [15] 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][8] 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 [8]. 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].

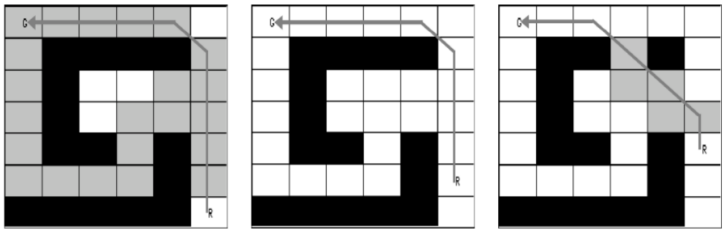

Figure 2.5: (left) The initial planning step evaluates the state of *22* cells. (center) Path produced from the initial plan. (right) When a shorter route is discovered only *5* cells need be evaluated. [5]

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 $\overrightarrow{e_0}$.

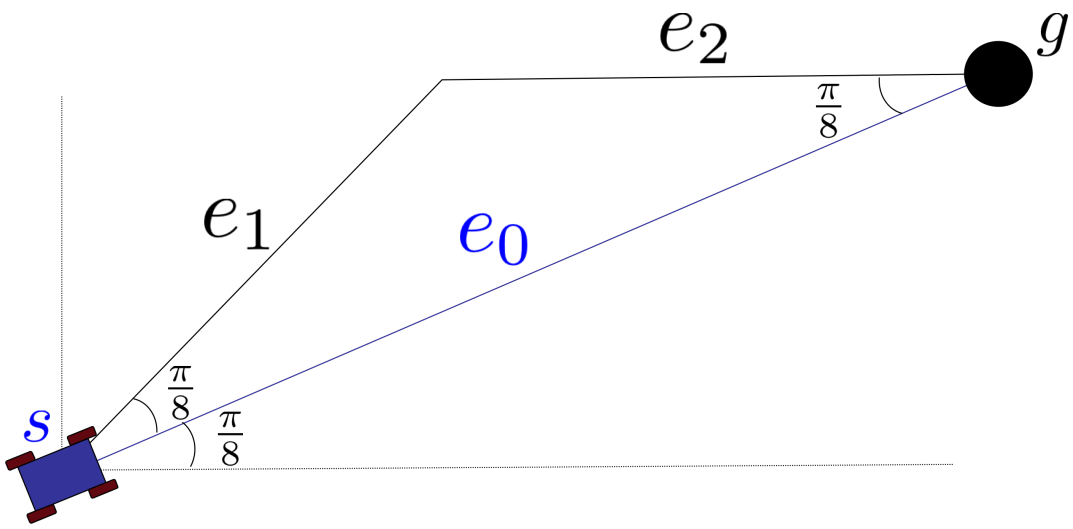

Figure 2.6: The shortest path here is along the vertex e_0, but with planners such as *D* Lite* the robot must correct its heading and pass through e_1e_2. [6]

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** [16] 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 $\{\overrightarrow{s_1s_2}, \overrightarrow{s_2s_3}, \overrightarrow{s_3s_4}, \overrightarrow{s_4s_5}, \overrightarrow{s_5s_6}, \overrightarrow{s_6s_7}, \overrightarrow{s_7s_8}, \overrightarrow{s_8s_1}\}$ [6], and so the 'optimal' path must intersect one of these edges (Figure 2.7(c)).

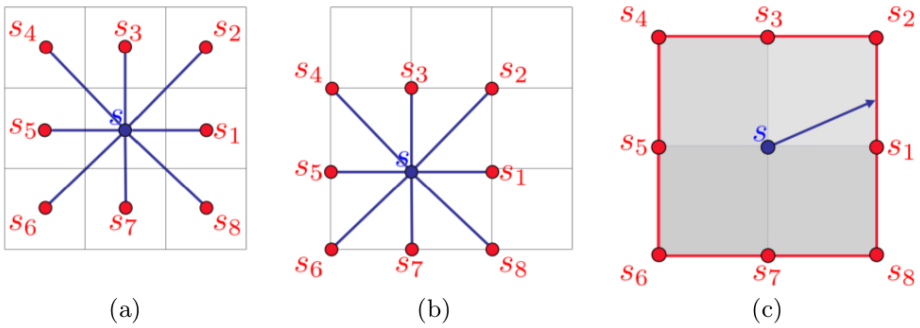

Figure 2.7: (a) The grid as a standard path planner represents it with the nodes appearing in the center. (b) How *Field D** treats cells with the nodes shifted to the corners. (c) The optimal path will intersect one of the defined edges at any heading. [6]

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' [16]. 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 [16]. 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.

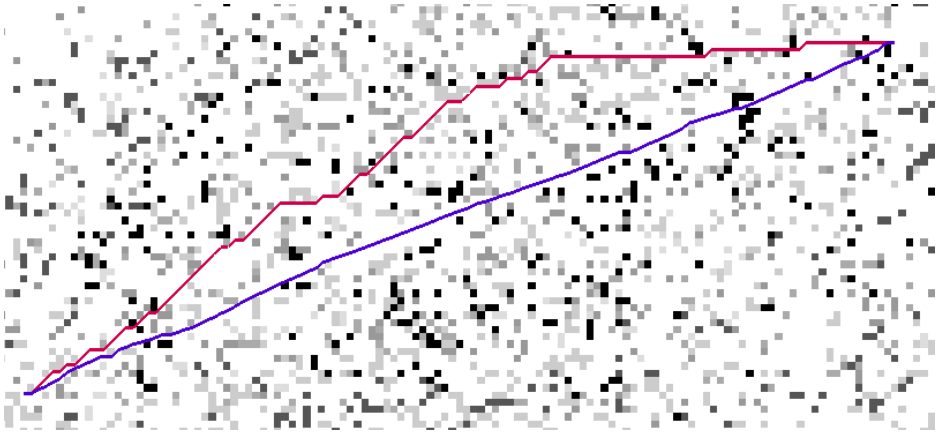

Figure 2.8: A comparison of the paths produce by *Field D** (blue) and D* Lite (red). [6]

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 [17], 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** [18]. 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 [19], 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

This section will contain the requirements for this system listed as statements or bullet points, use unambiguous English and avoid technical terms or specifications such as definite language or architecture choices.

3.1.1 Functional Requirements

When computing a shortest path the number of vertex accesses should be no more than the total number of vertices.

- Input: Map state space containing a goal and a start position.
- Output: Total number of vertices that are accessed during the computation.

The total time taken for the autonomous agent to traverse the physical path can be at most 50% more expensive than a tele-operated agent.

- Input: A path of points in Cartesian format.
- Output: Time taken in seconds to reach the goal.

Given an environment containing a start and goal state the path planner must compute a shortest path if one exists.

- Input: Map state space containing a goal and a start position.
- Output: A shortest path from the start to the goal or an error if none exists.

Where there is no traversable path to a goal state the system must immediately abort the planning procedure.

- Input: Map state space containing an unreachable goal.
- Output: None.

3.1.2 Non-Functional Requirements

Paths produced by the system should minimise the physical distance to be traversed with the aim of conserving energy resources.

The autonomous agents motion model must be based on a skid steering system that allows for on the spot rotations.

At least one physical courier robot must be capable of carrying out the instructions given to it from the path planning system.

3.2 System Modelling

3.2.1 Overview

One big diagram of the entire system should feature here, provide the viewer with a complete overview of the system and how everything relates to each other. Explain it in a general way.

3.2.2 Interaction Models

Include various UML based diagrams that show the interactions between the major components in the system the Planner, Algorithm, and Proxy in particular. Keep it high level do not go down into scrupulous detail, focus on communication mechanisms.

3.2.3 Core Class Diagrams

Class diagrams showing relationships between entities such as inheritance, each class diagram should be accompanied with an explanation. Core classes to model include:

- Robot
- Proxy
- Planner
- Algorithm (abstract)
- DStarLite (inherits Algorithm)
- FieldDStar (inherits Algorithm)

3.2.4 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.

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

...

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 [20] 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 A.

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.

3.3.3 Initiating a Scan

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 [**?**], Baxter [**?**], Nao [**?**], EV3 Lego Mindstorms [**?**], and the horde of robots that use ROS [**?**]. Linux is unique as it is mature, stable, customisable, embeddable, and freely available [**?**], making it a suitable candidate for powering the robot revolution.

INSERT PICTURE OF MENTIONED ROBOTS HERE WITH CITATIONS!

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 [**?**].

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♯ 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 [**?**]. An open repository was set-up and maintained in the cloud at the GitHub BotNav.

3.6 Hardware Agent 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 have all ready discussed this mechanism under 3.3 from the software side now it is time to 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.

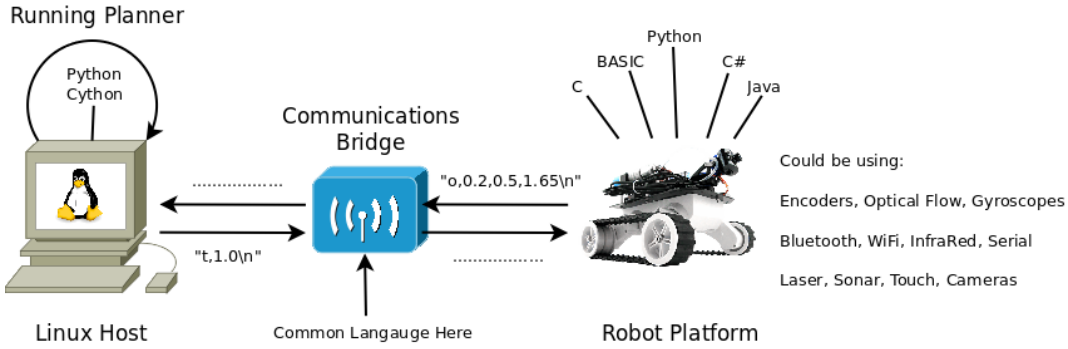

Figure 3.1: The planner on the Linux host and robot platform communicate via a common communications bridge that they both understand regardless of implementation details.

Talk about a specific case for a robot probably the microbot or something here...

3.6.1 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* [**?**], on the spot rotations are achieved by running the drive systems on either 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.

Chapter 4

Core Implementation

4.1 Building the Project

Section discusses the steps required to get the project up and running from raw source code, including getting the source, compiling C code, and running it from Python3. Basically a how to guide.

4.1.1 Obtaining the Source

State the three ways to obtain the source:

- Check it out using git.
- Download it as a Zip.
- Or use the CD at the back of this thesis.

4.1.2 Compiling the Cython Modules

State that since C code is target dependent it is necessary that the compilation be explained since it can be built for x86, x64, ARM, SPARC, PowerPC, etc. Build tool that will be used is Make. Talk about the Makefile contents.

4.1.3 Running it in Python3

Explain how you go about running the project once it is compiled, by default it will run a simulation example. Give the command line arguments for running it from a terminal.

4.2 Running Simulations

Talk about how simulations provide an easy means to test the system against predefined use cases, speeds up the testing process, and provides a means for reproducing results. Discuss these points here.

4.2.1 Configuration

Outline the changes that need to be made to the configuration file, basically the simulation flag will need to be set.

4.3 Using a Real Bot

Mention how using a real robot differs by:

- The need for communications.
- Introducing drift wheel slippage etc.
- By proving the practical application of the planner.

4.3.1 Configuration

Outline the changes that need to be made to the configuration file, highlight the fact that communications medium is required USB, Bluetooth, or Wifi. Explain the different variations.

4.4 How the Planner Works

Explain how the planner was implemented in code, the control logic, looping structure, reacting to change, and the conditions for terminating the planner.

4.4.1 Five Simple Steps

Every planner in robotics splits the problem into five simple steps, include them under this section as numbered items. State any recursive operations that take place. Also include a flowchart or diagram in some form.

4.4.2 Abstracting Away from the Algorithm

This section will cover the abstract model that the Algorithm class enforces, every planning algorithm has a common interface which allows them to be interchanged. Explain how this is achieved using abstraction and talk about the advantages.

4.5 Open Field D*

Core of the project very important, state that every implementation of Field D* to date is closed source NASA's code is not available, nor is Carnegie Mellon's. Open Field D* is significant because it bucks this trend making it open to ITB students and others.

4.5.1 Modifying D* Lite

Point out the key differences between D* Lite and Field D* from a coding perspective, nodes to cell corners how this is represented, linear interpolation. Using Georgia Institute of Technologies D* Lite code state the modifications required to get Field D*.

4.5.2 Basic Implementation

Cover basic implementation of Field D*, most importantly state any problems encountered, or variations/optimisations made during the coding stage.

Chapter 5

Results

The purpose of this section is to define the testing methodology used to compare D* Lite and Field D* to each other. It is split into two sections one covering the tests carried for simulations and the other for actual physical testing and their results.

5.1 Simulated Runs

Simulated runs are those performed using a software robot, basically a dummy. The real differences is that unlike hardware dependent tests they should be straight forward to reproduce, all except Test 3.

5.1.1 Sample Environments

Provide all of the necessary details surrounding the sample environments used, file names, physical and cell size, and an example of one possibly a picture or a scaled text version. These environments should be based on real ones however they will remain unchanged during the planning process.

5.1.2 Test 1: Path Length

Test 1: is concerned with the actual length of the path that both planners produce for the same environment. The length will be some absolute value which will be accompanied by a percentage that indicates how much shorter one is from the other.

Expected: based on Field D*'s authors it is expected that Field D*'s result will be shorted for any paths involve headings other than $\frac{\pi}{4}$.

Output: a plot file containing the full path for use with gnuplot.

5.1.3 Test 2: Vertex Accesses

Test 2: is concerned with how many vertices (cells) need to be accessed to build a shortest path. This is different from expansion which looks at all the vertices at once.

Expected: since Field D* can deal with more headings it should produce a shorter path and therefore pass through less vertices than D* Lite.

Output: absolute number of vertex accesses wrote to the debug file.

5.1.4 Test 3: Execution Time

Test 2: is concerned with how the time it took to execute each planning step, this value will be the total time taken for all planning steps.

Expected: Field D*'s authors claim that on average it takes 1.7 times as long to plan than D* Lite something close to this figure is expected but it is subject to the hardware.

Output: absolute total planning time wrote to the debug file.

5.2 Courier Robot Runs

Covers the tests conducted on physical robots these are designed to cover specific cases that are difficult to reproduce in simulations such as collisions and the real delivery time. They are completely implementation dependent and will vary from robot to robot making them difficult to reproduce.

5.2.1 Real Environments

Give a quick summary of the real environments that the courier robot(s) had to navigate through include some floor plans and pictures. Highlight any obstacles that were not included in the initial map, these environments will not containing moving entities.

5.2.2 Test 1: Navigating Unknown Terrain

Test 1: is concerned with the robots ability to navigate a completely unknown terrain when given only its dimensions. Obstacles shall be placed along the robots initial path to force replanning steps, the robot will have to rely on the accuracy of its sensors.

Expected: the robots sensors should enable it to detect any obstructions and plan around these. The resulting map should closely reassemble the floor plan.

Output: a map file of the discovered terrain.

5.2.3 Test 2: Collision Frequency

Test 2: is concerned with how frequent the proposed path results in the robot colliding with an obstacle. The likely cause for this will be that the planner produces a path that brings the robot very close to an obstacle.

Expected: collisions should be kept to a minimum.

Output: the number of disabling collisions a robot makes when following a path.

5.2.4 Test 3: Delivery Time

Test 2: is concerned with the time it takes the robot to travel from the start location to the goal. This includes all of the operations: planning, driving, turning, and scanning.

Expected: the robot should be able to complete the task with in a justifiable time frame.

Output: time it took to reach the goal wrote to the debug file.

5.3 Discussion

Discuss the findings, highlight any particularly interesting trends or patterns, also mention any difficulties encountered during testing. State the significance of the results

Chapter 6

Conclusion

Wrap up the entire project with a 1 page conclusion that covers:

- The outcome of the project success or failure.
- Significance of the findings.
- Any big challenges that were overcome.
- Outstanding issues or bugs that remain.
- Anything that could have been done differently.
- Learning outcome of the project.

Chapter 7

Further Work

It is hard to predict what any further work maybe at this stage but some possible suggestions will include:

- Modelling uncertainty in the robot's movements.
- Developing an advanced graphical user interface.
- Improving the robot's pose estimation using SLAM.
- Accommodating more sensor models than just LIDAR.

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

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

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

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

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

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

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

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

[15] 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.

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

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

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

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

[20] 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.

Appendices

Appendix A

Communications Protocol Specification

Full listing for the communications protocol used here explaining:

- Connection Types.
- All possible Commands.
- All possible Responses.
- Example usage.

Appendix B

ITB DFRobot Pirate Platform

Detailed explanation of the ITB's robot platform that was used with this project. Include all the pin configurations to sensors, circuit diagrams. Mention anything else of particular importance.

Appendix C

Source Code Listings

Appendix C.1

Planner

Dump the full source code for the Planner class here when it is ready, include the whole thing.

Appendix C.2

Field D* Algorithm

Dump the most important parts of the Field D* Algorithm here when it is ready. Do not include all of it because it is likely to be 1000+ lines of code.

Appendix D

Project Plan

Modify the original project plan to suit the final outcome/work flow of the project and place it here once done.