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 19, 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 19, 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

System Design

1.1 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.

1.1.1 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.

1.1.2 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.

1.2 System Modelling

1.2.1 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.

1.2.2 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.

1.2.3 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)

1.2.4 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 1.3 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.

INSERT FLOW CHART OR INTERACTION DIAGRAM HERE!

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
```

1.3 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 [**?**] 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 **??**.

1.3.1 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.

1.3.2 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.

1.3.3 Initiating a Scan

1.4 Programming Platforms

1.4.1 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)

1.4.2 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.

1.4.3 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.

1.5 Source Control

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

1.5.1 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.

1.6 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 already discussed this mechanism under 1.3 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 1.1). 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.

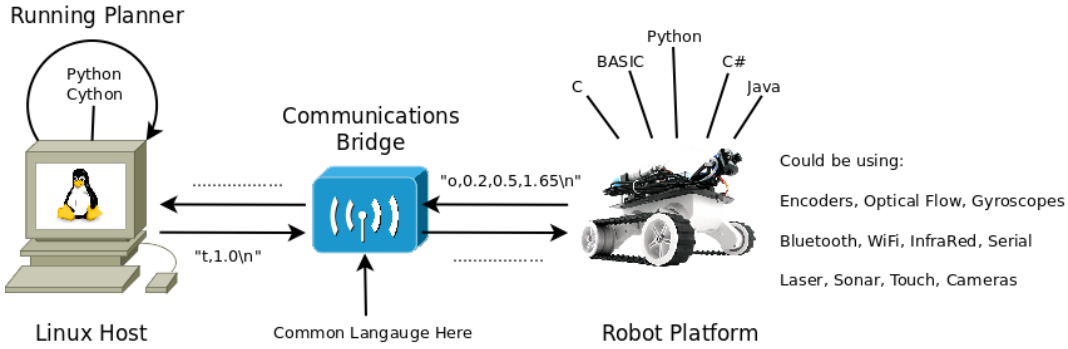

Figure 1.1: The planner on the Linux host and robot platform communicate via a common communications bridge that they both understand regardless of implementation details.

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

1.6.1 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.