

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  
March 6, 2015

## **Declaration**

I hereby 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: March 6, 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

<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Table of Contents</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>List of Figures</b>	<b>vi</b>
<b>Abbreviations</b>	<b>vii</b>
<b>1 Core Implementation</b>	<b>1</b>
1.1 Building the Project . . . . .	1
1.1.1 Obtaining the Source . . . . .	1
1.1.2 Compiling the D* Lite Cython Module . . . . .	1
1.1.3 Running it in Python3 . . . . .	2
1.2 Running Simulations . . . . .	3
1.2.1 Configuration . . . . .	3
1.3 Using a Real Bot . . . . .	4
1.3.1 Configuration . . . . .	5
1.4 How the Planner Works . . . . .	5
1.4.1 Five Simple Steps . . . . .	5
1.4.2 Abstracting Away from the Algorithm . . . . .	5
1.5 Open Field D* . . . . .	6
1.5.1 Modifying D* Lite . . . . .	6
1.5.2 Basic Implementation . . . . .	6

# List of Tables

# List of Figures

1.1	An example of the type of output generated after running GridNav over <i>sample.map</i> in simulation mode. . . . .	3
-----	---	---

# Abbreviations

DARPA	Defence Advanced Research Project Agency
SLAM	Simultaneous Localisation and Mapping



# Chapter 1

## Core Implementation

### 1.1 Building the Project

#### 1.1.1 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.

#### 1.1.2 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.

### 1.1.3 Running it in Python3

By default the project is set-up to run a sample simulation with *sample.map* using the GridNav 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 1.1.

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

## 1.2 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.

### 1.2.1 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"
```

---

## 1.3 Using a Real Bot

The real test for any path planning algorithm is its practical effectiveness 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 platform, 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 data to and from the robot. Then there is drift, occurrences such as wheel slippage can lead to the robot veering of course, we will assume that this issue is dealt with at a lower level than the path planning system.

### **1.3.1 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.

## **1.4 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.

### **1.4.1 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.

### **1.4.2 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.

## **1.5 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.

### 1.5.1 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\*.

### 1.5.2 Basic Implementation

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