**Oliver Freeman**

# A Comparison of Any-Angle Pathfinding Algorithms for Virtual Agents

Computer Science: Part II

Clare College

January 22, 2014

# Proforma

| | |
|---|---|
| Name: | **Oliver Freeman** |
| College: | **Clare College** |
| Project Title: | **A Comparison of Any-Angle Pathfinding Algorithms for Virtual Agents** |
| Examination: | **Computer Science Tripos: Part II, May 2014** |
| Word Count: | **????**[1] |
| Project Originator: | **Oliver Freeman** |
| Supervisor: | **Dr R. J. Gibbens** |

## Original Aims of the Project

To do.

## Work Completed

To do.

## Special Difficulties

None

# Declaration

I, Oliver Freeman of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

# Contents

# List of Figures

# Acknowledgements

To do.

# Chapter 1

# Introduction

## 1.1 Motivation

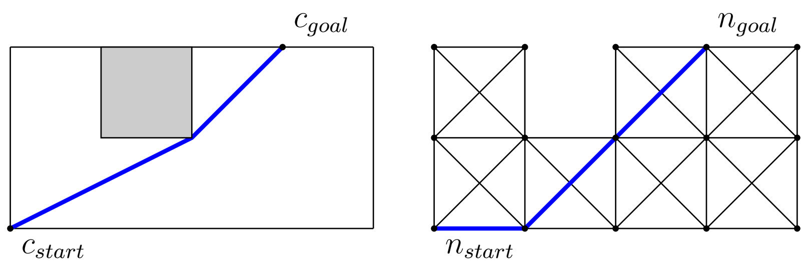
Finding short and realistic-looking paths through environments with arbitrarily placed obstacles is one of the central problems in artificial intelligence for games and robotics.

Pathfinding algorithms operate on a discrete mathematical construct called a graph. Therefore, to find a path through a continuous environment, the robot must obtain a graph that is a simplified representation of the environment - it is a simplification because it represents a continuous space environment with a discrete, finite object. Studies comparing algorithms for finding paths through graphs have shown that A* finds the optimal path in consistently less time than other pathfinding algorithms [Dagys 2013]. However, although the paths are optimal with respect to the graph, they are rarely optimal with respect to the environment (see Figure 1.1).

Over the last few years there have been multiple new algorithms proposed to efficiently find optimal or near-optimal paths through continuous environments. This dissertation aims to investigate their relative merits.

## 1.2 Related work

To do.



(a) Environment (b) Graph

Figure 1.1: Shortest path through environment vs. shortest path through graph

## 1.3 Project goals

To do.

# Chapter 2

# Preparation

## 2.1 Introduction to any-angle pathfinding

In this section I will introduce some key terms, define the any-angle pathfinding problem, and describe some decisions and methods to solve the problem.

### 2.1.1 Definitions

**Agent**

An entity that can move from one location in the environment to another. The agent cannot move to or through a location that is blocked by an obstacle.

**Map**

A representation of an environment as a two dimensional plane $\mathbb{R}^2$. For simplicity, this dissertation will focus on grid-based maps, where a map is divided into $N^2$ cells of equal size.

**Cell**

The logical unit of a map. Each cell may either be free or blocked. A blocked cell represents an obstacle that completely fills the cell.

**Line of Sight**

Exists between two coordinates on a map if a straight line between those two coordinates does not intersect any blocked cells.

**Path**

An ordered list of coordinates ($c_{start}$, $c_1$, ..., $c_{goal}$). For the path to be valid, a line of sight must exist between $c_{start}$ and $c_1$, $c_1$ and $c_2$, ... $c_{n-1}$ and $c_{goal}$.

**Graph**

A finite mathematical construct, consisting of nodes and undirected edges. A graph is a discrete representation of the $\mathbb{R}^2$ space of the map.

**Node**

Represents a single *coordinate* on the map.

**Edge**

An unordered pair of nodes. The existence of edge ($n_1$,$n_2$) denotes that an agent can travel in a straight line between the coordinates represented by $n_1$ and $n_2$. Each edge has an associated *weight*, which represents the Euclidean distance between the coordinates represented by the two nodes.

**Neighbour**

A node $n'$ is the neighbour of a node $n$ if there exists an edge ($n$,$n'$).

### 2.1.2 The any-angle pathfinding problem

The problem is to compute optimal or near-optimal paths, if they exist, through maps. The optimality is based on the total Euclidean distance and cumulative angle turned by an agent following the path.

The problem is split into two sub-problems:

- *Discretization* - a graph is created that represents that map in some way
- *Pathfinding* - an algorithm is run on the graph to produce a path

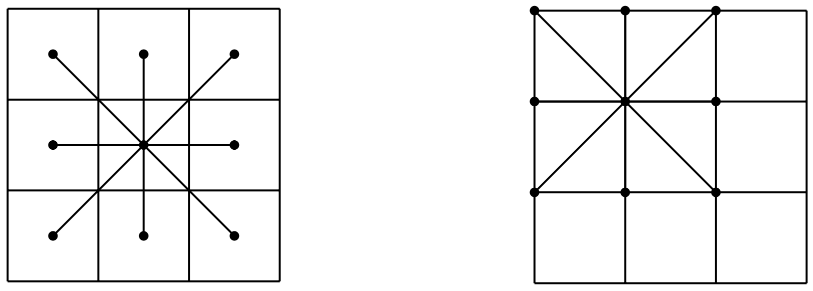
These two sub-problems will now be introduced:

### 2.1.3 Discretization

We must choose how to represent a continuous space, grid-based map with a graph consisting of a finite number of nodes and edges.

#### Grid-based graph

A common representation is to use a grid-based graph: there is one node per cell, and there are edges between nodes that represent locations that can be reached in a 1 unit step at $0^o$, $90^o$, $180^o$ or $270^o$ bearing, or a $\sqrt{2}$ step at $45^o$, $135^o$, $225^o$ or $315^o$ bearing. If the location represented by one node cannot be reached from the location represented by another node (because there is a blocked cell in between), then there will be no edge between the two nodes. We are now presented with a choice as to where to place the nodes:



(a) Nodes in the centres of cells (b) Nodes on the corners of cells

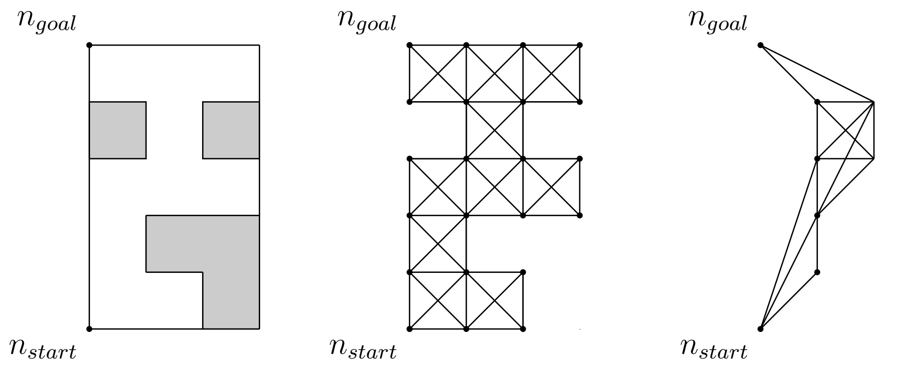
Figure 2.1: A node and its eight neighbours

The choice is essentially a stylistic one for most of the algorithms, but placing nodes in the middle of cells places stringent restrictions on the paths produced by Block A*, so I will place my nodes on the corners of cells.

#### Visibility graph

This is an alternative graph representation. Nodes exist at the start and end coordinates, and otherwise only on the corner of cells for which exactly three of the four surrounding cells are free. An edge exists between any two nodes that have a line of sight.

The optimal path through a visibility graph will correspond to the optimal



(a) Map (b) Grid-based graph (c) Visibility graph

Figure 2.2: Different graph representations for a given map

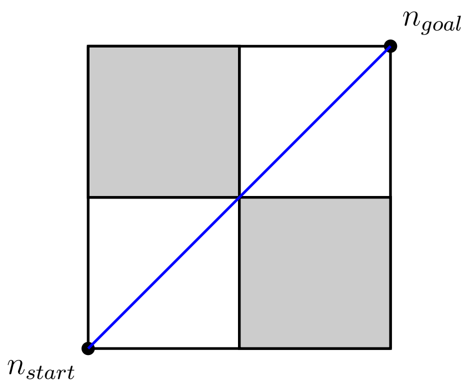


Figure 2.3: A valid path for an agent modelled as a point

path through a map, whereas the optimal path through a grid-based graph may not. However, grid-based graphs are generally accepted as preferable since, for a map of $N^2$ cells, a grid-based graph will have $O(N^2)$ edges, whereas a visibility graph has $O(N^4)$ edges, which can be unfeasible for large or high resolution maps. Therefore, this investigation will focus predominantly on grid-based graphs.

### Agent size

I will adopt the standard practice in any-angle pathfinding research by modelling the agent as a dimensionless point. Therefore, the path shown in Figure

2.3, which features a 'diagonal blockage', is a valid path. It is possible to ensure that such paths are never taken by adding extra checks in the graph creation and line of sight algorithms, but this would distract from the core investigation and be an unnecessary deviation from the accepted standard.

## 2.2 Any-angle pathfinding algorithms

In this section, I will introduce the four any-angle pathfinding algorithms that I will be investigating: A* with post smoothing, Theta*, Lazy Theta* and Block A*. Firstly, I will introduce an additional two comparatively basic algorithms to aid in the explanation of the more complex algorithms.

### 2.2.1 Dijkstra's shortest paths

Most of the algorithms that I will study will be derivatives of the well known A* graph traversal algorithm, which itself is a derivative of Dijkstra's famous shortest-path algorithm.

Dijkstra finds the shortest path from a given node $n_{start}$ to all other nodes in a graph - though it can be prematurely terminated when the goal is reached. Each node $n$ has a:

- *g-value* - represents the length of the shortest path found so far from $n_{start}$ to $n$. Initialised to $\infty$.
- *parent* - the previous node in the shortest path found so far from $n_{start}$ to $n$

The algorithm processes each node once only, selecting it from *openSet*, a priority queue that prioritises low g-values. When a node is selected, it's g-value is the length of the shortest path to it from $n_{start}$. A closed set is used to ensure that no node is expanded twice, as this would incur unnecessary work. When $n_{goal}$ is processed, the algorithm terminates. Dijkstra's shortest-path algorithm is optimal and complete.

If a path exists, the algorithm returns $n_{goal}$. The shortest path from $n_{start}$ to $n_{goal}$ can be then be traced:
$(n_{goal}, n_{goal}.parent, n_{goal}.parent.parent, ..., n_{start})$

**Algorithm 1:** DIJKSTRA

```
def Dijkstra($G$, $n_{start}$, $n_{goal}$)
    $openSet \leftarrow \emptyset$
    $closedSet \leftarrow \emptyset$
    $n_{start}.g \leftarrow 0$
    $openSet.add(n_{start})$
    while $openSet \neq \emptyset$ do
        $n_{curr} \leftarrow openSet.pop()$
        $closedSet.add(n_{curr})$
        if $n_{curr} = n_{goal}$ then
            return $n_{goal}$
        foreach $n_{neigh}$ of $n_{curr}$ do
            if $closedSet.contains(n_{neigh}) = false$ then
                if Update$(n_{neigh}) = true$ then
                    if $openSet.contains(n_{neigh}) = false$ then
                        $openSet.add(n_{neigh})$
    return $\emptyset$
def Update($n_{neigh}$)
    if $n_{curr}.g + euclidean(n_{curr}, n_{neigh}) < n_{neigh}.g$ then
        $n_{neigh}.g = n_{curr}.g + euclidean(n_{curr}, n_{neigh})$
        $n_{neigh}.parent = n_{curr}$
        return $true$
    else
        return $false$
```

### 2.2.2 A*

A* is based on Dijkstra's shortest-path algorithm, but uses a heuristic $h$ to reduce the number of nodes expanded, and hence often finds the optimal route to $n_{goal}$ in less time. Where Dijkstra preferentially expands nodes with low g-values, A* preferentially expands nodes with low f-values, where $f(n) = g(n) + h(n)$:

- $g(n)$ - calculated in the same way as in Dijkstra's algorithm
- $h(n)$ - Euclidean distance between $n$ and $n_{goal}$ - a cheaply computable monotonic estimate of the actual distance that will need to be traversed.

The monotonicity of Euclidean distance as a heuristic ensures that A* is optimal and complete.

The pseudocode for A* differs only from Dijkstra in the `update` subroutine, where the h-score must also be initialised.

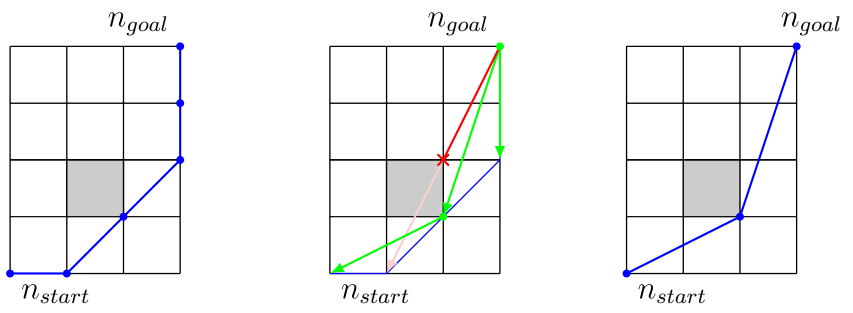
**Algorithm 2:** `Update` from A*

```
def Update($n_{neigh}$)
    if $n_{curr}.g + euclidean(n_{curr}, n_{neigh}) < n_{neigh}.g$ then
        $n_{neigh}.g \leftarrow n_{curr}.g + euclidean(n_{curr}, n_{neigh})$
        $n_{neigh}.f \leftarrow euclidean(n_{neigh}, n_{goal})$
        $n_{neigh}.parent = n_{curr}$
        return $true$
    else
        return $false$
```

### 2.2.3 A* with post-smoothing

A post processing step is run on the path returned by A*, where any unnecessary deviations are cut out by changing the parents of nodes of the path:
The algorithm starts with the node returned by A*: $n = n_{goal}$. The algorithm performs a line of sight test from $n$ to $n$'s parent's parent. If the test passes, $n$ will have its parent set to be its parent's parent. This test and

parent resetting is repeated until the line of sight test fails. Once the test has failed, we choose a new $n$, which will be the old $n$'s parent's parent - i.e. the node on which the line of sight test failed. We repeat this process until we reach the start node, at which point the algorithm terminates.



(a) Original path (b) Line of sight tests (c) Improved path

Figure 2.4: A* with post-smoothing

### 2.2.4 Basic $\theta^*$

Where A* with post-smoothing does an explicit smoothing step after finding the basic path, Basic $\theta^*$ interleaves smoothing with exploration by attempting to re-parent each node that is expanded with the parent of its parent. This reparenting occurs if there exists a line of sight between the node being expanded and the parent of its parent.

### 2.2.5 Lazy $\theta^*$

Lazy $\theta^*$ reduces the number of line of sight tests performed by Basic $\theta^*$. Basic $\theta^*$ will do a line of sight on every node that is visited (i.e. every neighbour of every expanded node) - however, if the node is visited but not expanded then this line of sight test will have been unnecessary.

Instead, when visiting a node $n$, Lazy $\theta^*$ assumes that there is a line of sight between each neighbour $n'$ and $n.parent$, and updates the g-value and parent of each $n'$ accordingly. The algorithm only checks to see if that line of sight

**Algorithm 3:** `PostSmoothing` from A* WITH POST-SMOOTHING

```
def PostSmoothing($n_{start}, n_{goal}$)
    $n_{curr} \leftarrow n_{goal}$
    $n_{next} \leftarrow n_{goal}.parent.parent$
    if $n_{next} = \emptyset$ then
        return
    while $true$ do
        while LineOfSight($n_{curr}, n_{next}$) do
            $n_{curr}.parent \leftarrow n_{next}$
            $n_{next} \leftarrow n_{next}.parent$
            if $n_{next} = n_{start}$ then
                return
        $n_{curr} \leftarrow n_{next}$
        if $n_{curr}.parent = n_{start}$ then
            return
        $n_{next} \leftarrow n_{next}.parent.parent$
```

**Algorithm 4:** `Update` from $\theta^*$

```
def Update($n_{neigh}$)
    if LineOfSight($n_{neigh}, n_{curr}.parent$) $= true$ then
        if $n_{curr}.parent.g + euclidean(n_{curr}.parent, n_{neigh}) < n_{neigh}.g$
        then
            $n_{neigh}.g \leftarrow n_{neigh}.parent.g + euclidean(n_{curr}.parent, n_{neigh})$
            $n_{neigh}.f \leftarrow euclidean(n_{neigh}, n_{goal})$
            $n_{neigh}.parent \leftarrow n_{curr}.parent$
            return $true$
        else
            return $false$
    else
        if $n_{curr}.g + euclidean(n_{curr}, n_{neigh}) < n_{neigh}.g$ then
            $n_{neigh}.g \leftarrow n_{curr}.g + euclidean(n_{curr}, n_{neigh})$
            $n_{neigh}.f \leftarrow euclidean(n_{neigh}, n_{goal})$
            $n_{neigh}.parent \leftarrow n_{curr}$
            return $true$
        else
            return $false$
```

actually exists if the $n'$ is ever expanded, by calling `Initialise` on $n'$ when it is popped off *openSet*. If the line of sight doesn't pass, `Initialise` alters the g-value of $n'$ to reflect this by taking as its new parent the expanded neighbour $n''$ or $n'$ which minimises $g(n') + distance(n', n'')$.

**Algorithm 5:** `Initialise` and `Update` from Lazy $\theta^*$

```
    def Initialise($n_{curr}$)
1       if LineOfSight($n_{curr}, n_{curr}.parent$) = $false$ then
2           $newParent \leftarrow \underset{n' \in expandedNeigh(n_{curr})}{\text{argmin}} (n'.g + distance(n', n_{curr}))$
3           $n_{curr}.parent \leftarrow n'$
4           $n_{curr}.g \leftarrow n'.g + distance(n', n_{curr})$
    def Update($n_{neigh}$)
        // assume line of sight test passes
5       if $n_{curr}.parent.g + euclidean(n_{curr}.parent, n_{neigh}) < n_{neigh}.g$ then
6           $n_{neigh}.g \leftarrow n_{neigh}.parent.g + euclidean(n_{curr}.parent, n_{neigh})$
7           $n_{neigh}.f \leftarrow euclidean(n_{neigh}, n_{goal})$
8           $n_{neigh}.parent \leftarrow n_{curr}.parent$
9           return $true$
10      else
11          return $false$
```

### 2.2.6 Block A*

Block A* is by far the most complicated algorithm that I will investigate. It was published in 2011, and is at the very cutting edge of any-angle pathfinding algorithmic research.

Block A* is a variant of A* that uses the concept of an LDDB ('Local Distance Database') - a database that holds the distances and inflection points of the optimum routes through small $n$ by $n$ submaps, known as blocks. The size of the blocks in the LDDB is optional - the size chosen will affect the size of the database, the lookup time of the database, the speed of the algorithm and the optimality of the algorithm.

Block A* deals with blocks in a similar way to that in which A* deals with nodes: blocks have neighbours and priority-values (called 'heap-values', like A*'s f-values), and can be expanded, put in a priority queue and popped off the priority queue. However, note that blocks can be re-expanded if a better route is found - this does not occur in A*.[1] The majority of the work of Block A* is performed by only considering nodes on the boundaries of blocks - since the LDDB can be used to quickly give costs for traversing the inside of the nodes. Therefore, Block A* can quickly skip across the map, block by block. The blocks that contain $n_{start}$ and $n_{goal}$ are special cases, as there is no guarantee that these nodes lie on the boundary of their respective blocks - and they may even lie in the same block. Therefore, the LDDB cannot be used for these blocks.

A block's heap-value is the lowest f-value of a boundary node that has been updated since the block was last expanded, and is reset to $\infty$ when the block is removed from the *openSet*. In a similar way to A*, the basic cycle to Block A* is that the block with the lowest heap-value will be removed from the *openSet* and expanded. To expand the block:

- a list Y is made of all boundary nodes of the block that have been updated since the block was last expanded.

[1]This subtlety caused me such confusion that I eventually emailed Peter Yapp, the author of the Block A* paper, for clarification. He explained the difference, and told me that he had included an explanation of this specific point when he gave a presentation [citation] on Block A* as the confusion is not uncommon.

- For each side of the block that isn't on the edge of the map (see Figure 2.4):
  - a list $listX$ is made of each node on that side
  - a corresponding list $listX'$ is made which contains, for each $x_i$ in $listX$, an $x_i'$ which is the corresponding node to $x_i$ in the neighbour block to that side.
  - the g-value of $x_i$ is updated if there is a $g(y) + LDDB(y, x_i)$ that is smaller that $g(x_i)$. We call $y$ the $ingress$ coordinate and $x_i$ the $egress$ coordinate, as $y$ is the point of entry to the block and $x_i$ is the proposed point of exit from the block.
  - if $x_i$ is updated, so is $x_i'$ - to $g(x_i') + distance(x_i, x_i')$ - which is 0 if we use nodes on corners or 1 if we use nodes in centre of cells.
  - if the smallest f-value of an updated $x'$ is smaller than the neighbour block's current heap-value, we update it and insert the neighbour block into the *openSet* if it is not already there.

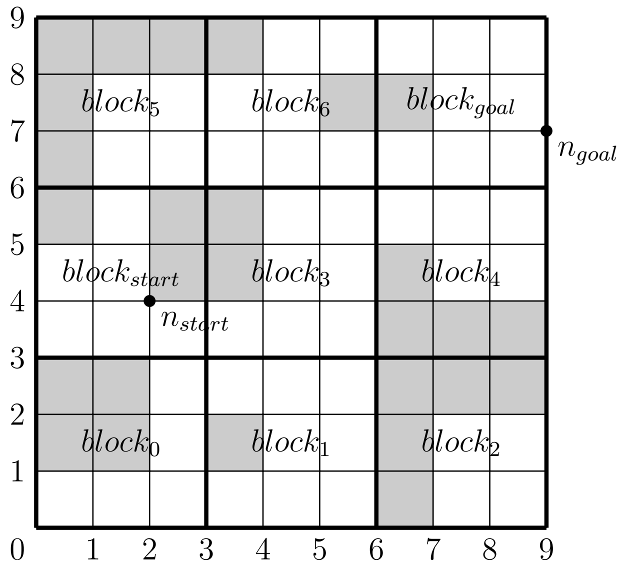
Special cases - where, in the general case, the LDDB is not sufficient:

- $block_{start}$: the call to `init(start)` creates and initialises $block_{start}$. The optimum path from $n_{start}$ to each boundary node $x$ is computed, and the length of this path is used to set $x$'s g-value.
- $block_{goal}$: the call to `init(goal)` creates and initialises $block_{goal}$. The optimum path from $n_{goal}$ to each boundary node $x$ is computed, and the length of this path is used to set $x$'s h-value.[2]
- $block_{start} = block_{goal}$: the optimum path from $n_{start}$ to $n_{goal}$ is calculated directly, and returned.

When $block_{goal}$ is evaluated, its h-value have already been calculated by finding the actual shortest path from each boundary node to $n_{goal}$ (see Special cases above), as opposed to using an estimate. Therefore, if the heap-value of $block_{goal}$ is less that any other blocks in *openSet* then there are no possible shorter paths to $goal_{block}$ so the algorithm terminates.

The pseudocode for Block A* can be found in Appendix ?

[2]h-value, not g-value. The difference between the treatment of $block_{start}$ and $block_{start}$ is important



When expanding $block_4$:

$listX$ =
(6,3),(7,3),(8,3),(9,3),(6,4),(6,5),(6,6),(7,6),(8,6),(9,6) from $block_4$

$listX'$ =
(6,3),(7,3),(8,3),(9,3) from $block_2$,
(6,3) from $block_1$,
(6,3),(6,4),(6,5),(6,6) from $block_3$,
(6,6) from $block_6$,
(6,6),(7,6),(8,6),(9,6) from $block_{goal}$

Figure 2.5: Block A*: map of $9 \times 9$ cells is split into 9 blocks of $3 \times 3$ cells

## 2.3 Requirements analysis

As specified in the 'Work to be done' section of my Proposal (see Appendix E), my project was divided into four sections. This section outlines the functional and non-functional requirements for the system, and their relative priorities using the MoSCoW system.

**M** - Must; **S** - Should; **C** - Could; **W** - Won't

### 2.3.1 Testing simulator

| ID | Functional requirement | Priority |
|---|---|---|
| 1 | The system shall load one of a collection of maps from the generator | M |
| 2 | The system shall load one of a collection of maps from a saved file | S |
| 3 | The system shall create a grid-graph from a given map | M |
| 4 | The system shall create a visibility graph from a given map | C |
| 5 | The system shall run one of a collection of any-angle path-finding algorithms on a graph and collect data such as the path-length and the length of computation | M |
| 6 | The system shall display a visual representation of the current map and the paths found by any algorithms that have been run on it | M |
| 7 | The system shall display the numeric statistics for each path for the current map | M |

| ID | Non-functional requirement | Priority |
|---|---|---|
| 1 | The system shall be designed in a modular way to allow easy extension for new algorithms | S |

### 2.3.2 Map generation

| ID | Functional requirement | Priority |
|---|---|---|
| 1 | The system shall generate pseudo-random maps of a given resolution, coverage percentage and clustering | M |
| 2 | The system shall allow maps to be saved so that multiple tests can be run on the same map suite | M |
| 3 | The system shall allow maps to be created with an interactive map editor | C |
| **ID** | **Non-functional requirement** | **Priority** |
| 1 | The system shall generate maps of the highest resolution in under 2 seconds | S |

### 2.3.3 Algorithm implementation

| ID | Functional requirement | Priority |
|---|---|---|
| 1 | The system shall correctly implement each of the chosen algorithms. If a path exists, the path and numerical statistics will be returned. If no path exists, this will be returned | M |
| 2 | The system shall be allow arbitrary start and end coordinates for any map | S |
| **ID** | **Non-functional requirement** | **Priority** |
| 1 | The system shall be designed in a modular way to allow easy extension for new algorithms | S |

### 2.3.4 Data gathering

| ID | Functional requirement | Priority |
|---|---|---|
| 1 | The system shall write statistics for an arbitrary set of specified algorithms on an arbitrary set of specified maps and write the results to a CSV file | M |
| **ID** | **Non-functional requirement** | **Priority** |
| 1 | The system shall be designed with a clear API that enables quick and easy data gathering. | M |

To do - use case diagrams etc.

## 2.4 Testing

To do.

## 2.5 Design model

Having completed the preparation phase of the project, I refined plan for the implementation phase from that presented in the Project Proposal (see Appendix ?). An incremental model of implementation was adopted, with new modules being developed and tested separately before being integrated into the work program. The milestones of the project were:

*Milestone 1* - Maps of arbitrary size, coverage and clustering can be created and printed to system output.

*Milestone 2* - Arbitrary maps can be converted to grape, and A* can be run on these maps. A visual representation of the path can be printed to system output.

*Milestone 3* - Basic UI is built, including all functionality from *Milestone 2*. Basic path statistics are displayed.

*Milestone 4* - Map saving, map loading and map creation functionality are present. This will facilitate debugging edge cases for more complex algorithms.

*Milestone 5* - Line of sight, A* with post-smoothing, $\theta^*$ and Lazy $\theta^*$ are implemented.

*Milestone 6* - Block A* is implemented.

*Milestone 7* - Data extraction scripts are implemented.

## 2.6 Languages and tools

**Programming language**

*Java* - provides abstraction and class hierarchy to enable development of modular, extensible code.

**Libraries**

*Swing* - for graphical user interface design and *CSVWriter* for data export.

**Integrated development environment**

*Eclipse* - allows rapid development through integrated testing, refactoring and version control tools.

**Statistical analysis and visualisation**

*R* - an open-source statistical package, due to its flexibility and extensibility.

**Backup**

*DropBox* and *Google Drive* - both of which maintain multiple shadow copies of my work in the cloud.

**Version Control**

*GitHub* - which facilitated exploring different implementation strategies by forking my core code repository.

# Chapter 3

# Implementation

In this chapter I will give an overview of the implementation of the project, as well as investigating some of the more interesting features.

This section will be split into six parts:

**Map generation**
: an explanation of the algorithm used to generate maps.

**Simulation**
: an overview of the simulator, including graph generation and algorithm data.

**Algorithms**
: each of the algorithms will be covered, with specific attention given to Block A*.

**User Interface**
: an overview of the graphical user interface.

**Testing**
: an overview of the methods used to test the code for correctness

**Data extraction**
: an explanation of how large volumes of data were extracted for statistical analysis

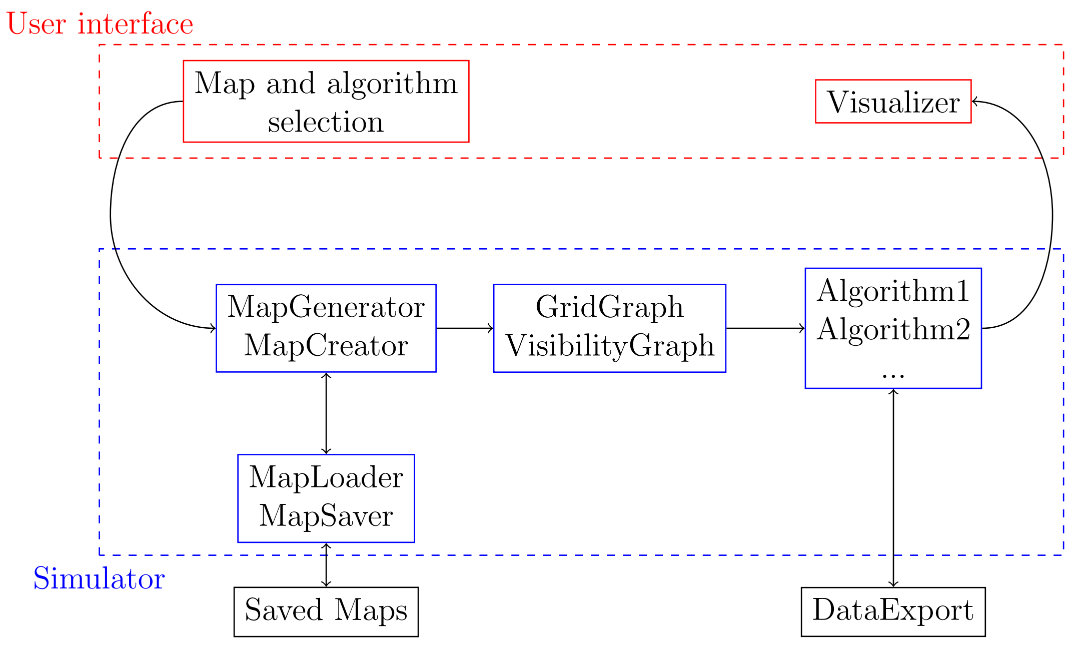


Figure 3.1: Flow of the user interface and simulator

## 3.1 Map generation

In order to make reliable conclusions about the performance of the algorithms, I needed a way to generate large volumes of maps of a given resolution $N$, coverage percentage $C$ and clustering $D$. I devised an algorithm to pseudo-randomly such maps. The algorithm creates maps of varying clustering by loosely approximating the idea of potential fields.

The input to the algorithm includes an integer matrix $m_{i,j}$ of size $N^2$ - where each element has been initialised to value 1. At any point in the algorithm, $m_{i,j}$'s value represents its 'potential' - i.e. the chance that $m_{i,j}$ will be the next element set to 0 (i.e. blocked). The algorithm performs $C \times N^2$ iterations. In each iteration, it chooses a random number $r$ between 0 and $\sum_{i,j} m_{i,j}$. It then traverses the matrix row-by-row until the sum of the elements it has seen is at least $r$. The potential of the element that has been reached is set to 0, and the potential of the surrounding elements is increased in a crude approximation of a potential field. The output is an integer matrix $m_{i,j}$ of size $N^2$ - where $C\%$ of the elements have value 0, which denotes a blocked cell, and the rest of the elements have value >0, which denotes a free cell. The array can then be parsed into a `Map` by the `Map` constructor.

**Algorithm 6:** GenerateMap

```
def GenerateMap($m, C, D$)
1   repeat
2       $r \leftarrow random(0, \sum_{i,j} m_{i,j})$
3       $i, j \leftarrow 0$
4       while $r \geq 0$ do
5           $r \leftarrow r - m_{i,j}$
6           if $i < R - 1$ then
7               $i \leftarrow i + 1$
8           else
9               $i \leftarrow 0$
10              $j \leftarrow j + 1$
11      SetAsBlocked($i, j$);
    until $(C \times N^2) times$
def SetAsBlocked($m_{i,j}$)
12  $m_{i,j} \leftarrow 0$
13  foreach $m_{k,l}$ in $horizontalOrVerticalNeighbour(m_{i,j})$ do
14      if $m_{k,l} \neq 0$ then
15          $m_{k,l} \leftarrow m_{k,l} + D$
16  foreach $m_{k,l}$ in $diagonalNeighbour(m_{i,j})$ do
17      if $m_{k,l} \neq 0$ then
            $m_{k,l} \leftarrow a_{k,l} + 2 \times D$
```

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

(a) Initialisation

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 3 | 5 | 3 | 1 |
| 5 | 0 | 5 | 1 |
| 3 | 5 | 3 | 1 |

(b) Iteration 1: r = 5

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 3 | 5 | 3 | 1 |
| 5 | 0 | 9 | 3 |
| 3 | 9 | 0 | 5 |

(c) Iteration 2: r = 2

Figure 3.2: Two iterations of `GenerateMap` with $R$=4 and $D$=2

## 3.2 Simulation

### 3.2.1 Graph Generation

To create a graph from a map, the map was iterated over twice. The first iteration created nodes for all the coordinates that were valid node locations. The second iteration set the neighbours of each node. Checks for both of the iterations were performed by checking whether neighbouring blocks were blocked. See Figure 3.3.

As discussed in the Preparation chapter, along with grid-based graphs, I am also going to investigate the performance benefits of running path-finding algorithms on visibility graphs. The creation of visibility graphs depends on having a `LineOfSight` function.

#### Line of Sight

The line of sight algorithm is based on the pseudocode in [reference Theta* paper], which itself is a derivative of Bresenham's line drawing algorithm - though instead of choosing pixels (i.e. cells) to draw, it chooses cells to check whether they are blocked. Bresenham's algorithm is a useful framework as it avoids any floating point calculations when the start and end points are integers - this has dual benefits:

- the algorithm is fast
- the algorithm doesn't suffer from rounding errors inherent in floating-point calculations.

```
...
Coordinate[] diagonalRelativeCellCoordinates =
  {new Coordinate(-1,-1), new Coordinate(0,-1),
  new Coordinate(0,0), new Coordinate(-1,0)};

Node[][] graphArray2D =
 new Node[map.getWidth()+1][map.getHeight()+1];
for(int j=0; j < map.getHeight()+1; j++) {
 for(int i=0; i < map.getWidth()+1; i++) {
  boolean isUnblockedAdjacentCell = false;
  for(Coordinate c: diagonalRelativeCellCoordinates) {
   try {
    if(!map.getCell(i+c.getX(),j+c.getY()).isBlocked()) {
     isUnblockedAdjacentCell=true;
    }
   } catch (ArrayIndexOutOfBoundsException e) {}
  }
  if(!isUnblockedAdjacentCell) {
   graphArray2D[i][j] = null;
  } else {
   graphArray2D[i][j] = new Node(new Coordinate(i,j));
  }
 }
}
...
```

Figure 3.3: Code snippet showing `Node` creation for grid-based graphs

A notable alteration to the basic Bresenham algorithm is that instead of checking one cell per column (or one per row), the line of sight algorithm will check any cell that the real line passes through. This only requires a minor alteration.

For the purposes of clarity, the pseudocode presented in LINEOFSIGHT assumes the line of sight is in octant 1 - i.e. whose angle with the x-axis is between $0^o$ and $45^o$. The full pseudocode can be found in Appendix ?. On this assumption, the key variables are:

$x$ and $y$ - integers that represent the coordinate of the cell being considered, which is always a cell that the line passes through.

$f$ - a value that represents at what point the line intersects $x + 1$ with respect to the current $y$ value. See Figure 3.4.

The algorithm starts at $n_{start}$. The current f-value is increased by $dy$ every time $dx$ is increased, but decreased by $dx$ if $y$ is increased. If at any point $f = 0$ then then the line intersects the bottom right-corner of the cell currently being considered, or if $y = dy$ then it intersects at the top left-corner. Therefore, if $f > 0$ then we need to check the $cell_{x,y}$ to see if it's blocked and if $f > dy$ then we additionally need to check $cell_{x,y+1}$.

*Note:* to disallow a line of sight through a 'diagonal blockage' (as introduced in section 2.1.3, a new **if** clause would be added after line 18 of LINEOFSIGHT:
**if** $f = 0 \wedge cell_{x,y}.isBlocked() \wedge cell_{x+1,y-1}.isBlocked()$ **then return** $false$.
To avoid the possibility of the final clause in the condition throwing an `ArrayIndexOutOfBoundsException`, an extra $x$-coordinate check or a `try-catch` block would also be required.

### 3.2.2 Algorithm Data

Each `MapInstance` has a `Map`, a `Graph` and an `AlgorithmData` object for each of the algorithms that have been run on that map.

`AlgorithmData` is an abstract class to facilitate adding new algorithms in a modular way. It has a public method `go()`, as well as getter methods for all statistical data about each algorithm. `go()` calls the $getPath(n_{start}, n_{goal})$

**Algorithm 7:** LineOfSight

```
def LineOfSight ($n_{source}$,$n_{goal}$)
    $x \leftarrow n_{source}.x$
    $y \leftarrow n_{source}.y$
    $x_{goal} \leftarrow n_{goal}.x$
    $y_{goal} \leftarrow n_{goal}.y$
    $dx \leftarrow x_{goal} - x$
    $dy \leftarrow y_{goal} - y$
    $f \leftarrow 0$
    while $x \neq x_{goal}$ do
        $f \leftarrow f + dy$
        if $x \geq dx$ then
            if $cell_{x,y}.isBlocked()$ then
                return $false$
            $y \leftarrow y + 1$
            $f \leftarrow f - 1$
        if $f \neq 0 \wedge cell_{x,y}.isBlocked()$ then
            return $false$
        if $dy = 0 \wedge cell_{x,y}.isBlocked() \wedge cell_{x,y-1}.isBlocked()$ then
            return $false$
        $x \leftarrow x + 1$
    return $true$
```

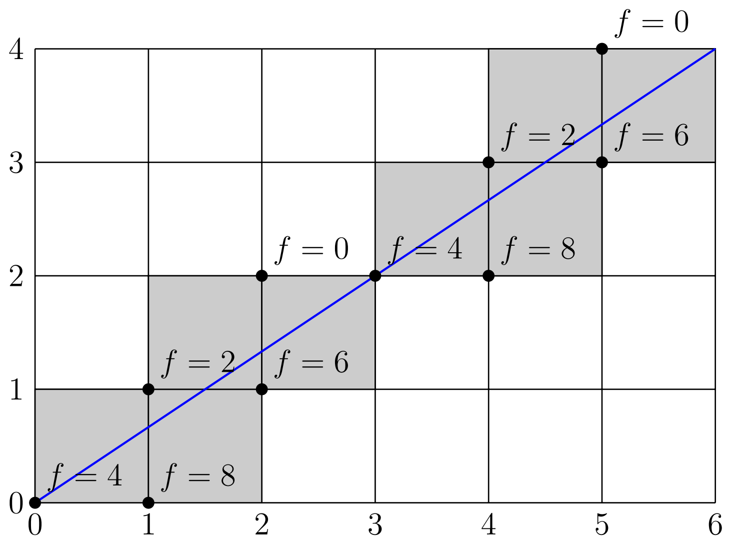


Figure 3.4: Line of sight algorithm

method, which returns $n_{goal}$ if a path is found, or *null* otherwise. If *null* is returned, this is recorded in the statistics. If $n_{goal}$ is returned, `go()` calculates some of the statistics as follows:

**Path length**
the sum of the Euclidean distances between each pair of consecutive nodes in the path.

**Cumulative path angle**
the sum of the scalar product between each pair of adjacent path segments in the path.

**Graph calculation time**
`System.nanoTime()` is used to calculate the duration between the call to `generateGraph()` and it returning.

**Path calculation time**
`System.nanoTime()` is used to calculate the duration between the call to `getPath()` and it returning.

**Number of nodes expanded**
a counter was incremented each time a node was expanded. It should

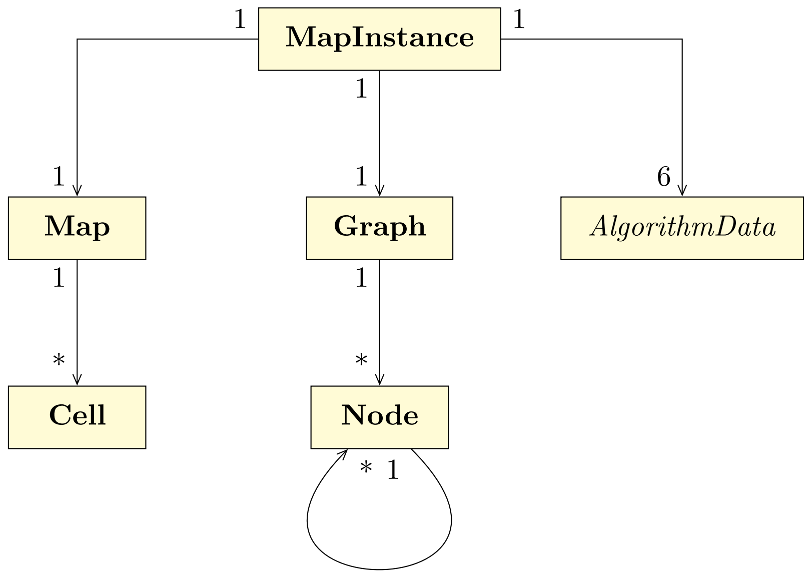


Figure 3.5: Composition of `MapInstance`

be noted that in Block A*, node expansion is not the same as block expansion.

## 3.3 Algorithms

To emphasise the close relationships between the algorithms and allow for maximal code re-use, I organised the concrete instances of `AlgorithmData` in a hierarchy. This also ensured that any performance differences between algorithms was due to the different nature of each algorithm and not different implementation of similar concepts. See Figure 3.5.

### 3.3.1 Dijkstra's shortest paths

The implementation was based on the pseudo-code seen in section 2.2.1. Details of note include:

**Open Set**
the standard `java.util.PriorityQueue` would occasionally not return

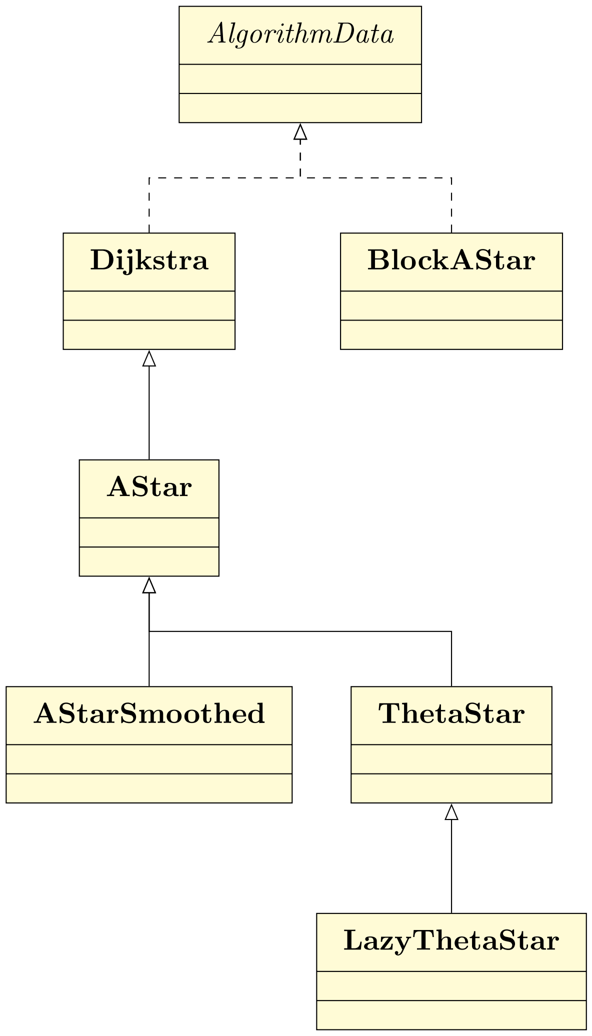


Figure 3.6: Inheritance structure of algorithms

the node with the smallest g-value when *openSet.pop*() was called. This is a documented bug for large queues that occurs when a queue item has its priority[1] altered while residing in the *openSet*, so l had to manually *pop*(), *update*() and re-*add*() any node that needed to be updated when already in the *openSet*.

**Closed set**
the only two operations on the `closedSet` are adding and checking for membership. Therefore, a `HashSet` was used for its average case O(1) insertion and search speed.

**Extensibility**
to allow a clear algorithm hierarchy, I needed to add a call to $initialise(n)$ whenever a node $n$ was popped from the *openSet*, and a $postProcessing(n)$ step before the node is returned. In `Dijkstra` these have empty method bodies, but some of the algorithms that inherit from Dijkstra will override these methods.

### 3.3.2 A*

The implementation was based on the pseudo-code seen in section 2.2.2: A* inherits from Dijkstra, and overrides the $updateCost()$ function.

### 3.3.3 A* with post-smoothing

The implementation was based on the pseudo-code seen in section 2.2.3: A* inherits from A* and implements its post-smoothing by overriding the $postProcessing()$ function.

### 3.3.4 Basic $\theta^*$ and Lazy $\theta^*$

The implementation was based on the pseudo-code seen in section 2.2.4 and 2.2.5:

**Basic $\theta^*$**
inherits from A*, and overrides the $updateCost()$ function

[1]In this case, the `Node`'s g-value

**Lazy $\theta^*$**
inherits from Basic $\theta^*$, and overrides the $initialiseNode()$ and $updateCost()$ functions

### 3.3.5 Block A*

#### Local Distance Database

The first challenge was to obtain an LDDB. Since there was no publicly available library containing the LDDBs, I had to create my own. Although it would have been possible to manually create the entries for the an LDDB for block sizes of $2 \times 2$ cells, this would certainly not have been feasible for block sizes that were any larger, since for blocks of size $n \times n$, there will be $2^{n^2}$ possible blocks, each with $4.n.4.(n-1)$ pairs of ingress and egress coordinates, which gives over 12 million calculations for a block size of $4 \times 4$.

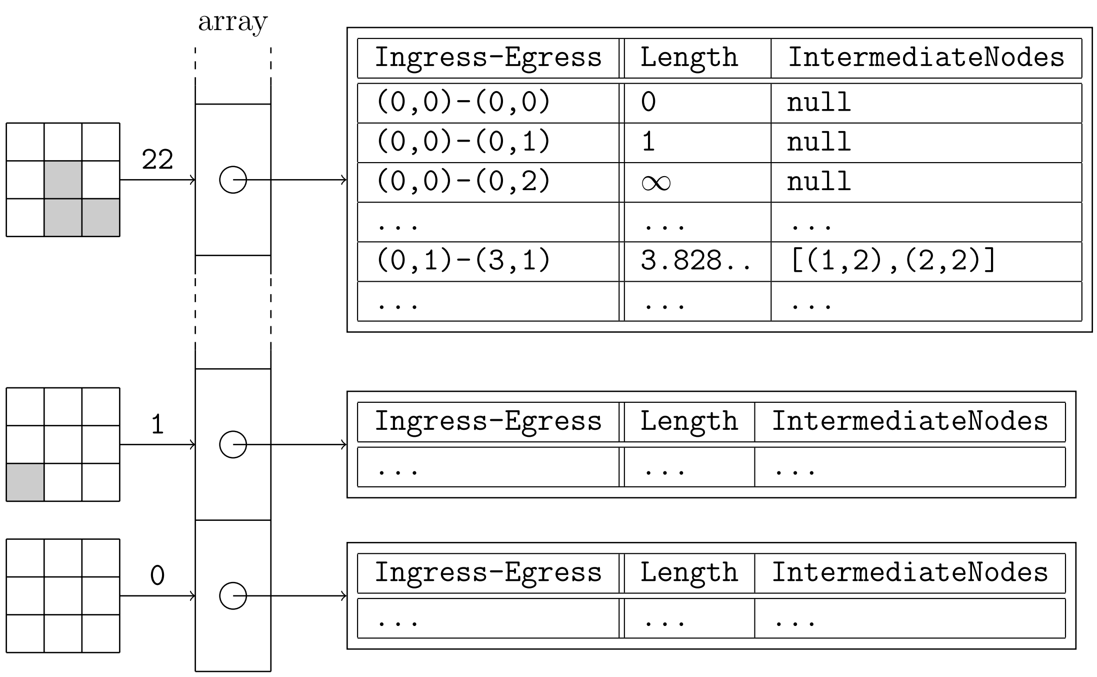
I calculated the shortest paths using A* over visibility graphs, which gives provably optimal paths.

For a given block size , I required:

```
for every possible block of size $n \times n$
    for every possible ingress-egress coordinate pair
        (a) the shortest path between that pair
        (b) a list containing every intermediate coordinate on that path.
```

It was necessary to put these entries into a database structure that would be compact in memory and fast to load and query. Since these constraints were fundamental to the operation of the algorithm, any library or $3^{rd}$ party database implementation could not be guaranteed to be specialised enough for the task, so I implemented my own database using arrays and hash tables to ensure optimal performance and minimum space wastage.

All queries to the database would need to identify a block and pair of coordinates as an input, and would receive either the length of shortest length between those nodes, or a list of the intermediate nodes on the shortest path between those two nodes. I devised a simple bitwise encoding scheme to represent block as integers that would be much more space and time efficient that storing Map objects in the database to use as comparison: each cell



| Ingress-Egress | Length | IntermediateNodes |
|---|---|---|
| (0,0)-(0,0) | 0 | null |
| (0,0)-(0,1) | 1 | null |
| (0,0)-(0,2) | $\infty$ | null |
| ... | ... | ... |
| (0,1)-(3,1) | 3.828.. | [(1,2),(2,2)] |
| ... | ... | ... |

| Ingress-Egress | Length | IntermediateNodes |
|---|---|---|
| ... | ... | ... |

| Ingress-Egress | Length | IntermediateNodes |
|---|---|---|
| ... | ... | ... |

Figure 3.7: Extract of the LDDB for block size of $3 \times 3$ - array of `HashTable<PairOfCoords,Pair<double,List<Coord>>`s

in the block is represented by a bit in the integer, and that bit is set if the corresponding cell is blocked. The 32 bits of the integer are sufficient for all the block sizes that are worth considering: $2 \times 2$ to $4 \times 4$.[1]

Using this scheme, the underlying implementation of my database was an array of `HashMaps` - one `HashMap` per block, with the array indexed by the code of the block. The `HashMap` mapped a key: a `Pair` of ingress-egress `Coordinate`s, to a value: a `Pair` consisting of the length (as a `double`) of the shortest path and an `ArrayList` of the `Coordinate`s on the path.

This implementation was sufficient but unsatisfactory, as the database sizes were unnecessarily large. This would cause slow loading to memory, less of the LDDB stored in cache and more likelihood of thrashing. Therefore, I compressed the database using bitwise encoding schemes:

- each `Pair` of ingress-egress `Coordinate`s can be represented with a unique integer representable in a `byte`'s worth of space - so a cod-

```
PairOfCoords(Coordinate c1, Coordinate c2, int blockSize) {
  this.blockSize = blockSize;
  this.coordCode = (byte)
   (c1.getX() +
   c1.getY() * (blockSize+1) +
   c2.getX() * (blockSize+1)*(blockSize+1) +
   c2.getY() * (blockSize+1)*(blockSize+1)*(blockSize+1));
 }
```

Figure 3.8: Encoding scheme for `PairOfCoords` in the compressed LDDB

```
int getListCode(List<Coordinate> intermediateNodes) {
  int listCode = 0;
  for(Coordinate c : intermediateNodes) {
   listCode = listCode | c.getX();
   listCode = listCode << 3;
   listCode = listCode | c.getY();
   listCode = listCode << 3;
  }
  listCode = listCode >>> 3; //undoes line 7 on final loop
  return listCode;
 }
```

Figure 3.9: Encoding scheme for `List<Coordinate>` in the compressed LDDB

ing scheme was devised to create an efficient hash function for the `PairOfCoordinate` class.

- the `List` of intermediate `Coordinates` can be represented by a code that fits into the 32 bits of an integer: since the maximum number of intermediate nodes on a shortest path in a sub map of size up to $4 \times 4$ is $4^2$, and the range of $x$ and $y$ in the `Coordinate` is 0-4, therefore 6 bits can be used for each `Coordinate` - 3 for $x$, 3 for $y$, which is a maximum total of 24 bits.

These compression techniques allowed me to reduce the size of the databases by about 50%:

| Submap size | Size of uncompressed LDDB | Size of compressed LDDB[3,4] |
|---|---|---|
| $2 \times 2$ | 33KB | 17KB |
| $3 \times 3$ | 2.2MB | 1.1MB |
| $4 \times 4$ | 485MB | 192MB |

I will investigate the performance of the uncompressed and compressed LDDBs in the Evaluation section.

### Special case blocks: $block_{start}$ and $block_{goal}$

$block_{start}$ and $block_{goal}$ are treated as special cases by Block A*. On initialisation:

- the *g-values* of the boundary nodes of $block_{start}$ are set by manually computing shortest paths from $n_{start}$ to each boundary node, instead of looking them up in the LDDB as with other blocks. This is because $n_{start}$ is not guaranteed to be on the boundary of a block, and the LDDB only holds details for routes between boundary nodes

- the *h-values* of the boundary nodes of $block_{goal}$ are set manually for similar reasons.

- a check is made as to whether $n_{start}$ and $n_{goal}$ are in the same block - i.e. $block_{start}$ equals $block_{goal}$. If so, the shortest path between the two is computed manually.

To compute these values, I created a visibility graph for the relevant block and computed the shortest path to each boundary node. Having run some preliminary tests, it became clear that on small maps these initialisations

[1]The exponential increase in size of LDDB would mean that the LDDB for block sizes of $5 \times 5$ or larger take so long to search that performance benefit diminishes

[2]Obtained by experimental results

[3]These savings could be improved further for sub maps of size $2 \times 2$ and $3 \times 3$ by using specific data-types depending on the submap size, but this was unnecessary, as the LDDB size was very small compared to the total available memory for sub maps of size$2 \times 2$ and $3 \times 3$, and the LDDB only needs to be loaded into memory once per run of the program.

[4]These sizes were significantly larger than those found in Yapp's paper, but his agents were only allowed to travel horizontally and vertically, so only 4 bits were needed to store the path length value.

took up to 50% of the run-time of the algorithm [insert some statistics of test runs]. Therefore, I decided to implement two alternative, extended forms of the LDDB:

**Semi-extended** contains details of shortest paths from *any* node to *any* *boundary* node in a block.

**Fully-extended** contains details of shortest paths from *any* node to *any* node in a block.

The increase in the size of the LDDB in comparison to the original is:

| Submap size | Semi-extended | Fully-extended |
|---|---|---|
| $2 \times 2$ | 12.5% | 26.5% |
| $3 \times 3$ | 33.3% | 77.8% |
| $4 \times 4$ | 56.3% | 144.1% |

The use of the fully-extended LDDB allows the initialisation initialisation of $block_{start}$ and $block_{goal}$ and the $block_{start}$ equals $block_{goal}$ case to use the LDDB, whereas using semi-extended LDDB requires that the $block_{start}$ equals $block_{goal}$ case finds the shortest route manually.[1] The performance benefits of these different forms of the LDDB will be investigated in the Evaluation chapter.

### Block A* - algorithm

The core of the algorithm was a straightforward implementation once the details of it were understood. The most challenging part of the implementation was the traceback stage, which was unspecified in the paper. The traceback stage extracts the coordinates of the path from the Graph structure. This requires finding and inserting into a list:

- the optimal path from $n_{goal}$ to the optimal ingress node of $block_{goal}$
- the nodes on the optimal path that share block boundaries
- the intermediate nodes where the optimal path traverses a block
- the optimal route from the optimal egress node of $block_{start}$ to $n_{start}$

[1]Note that when using the semi-extended LDDB, $block_{goal}$ needs to reverse the ingress-egress coordinates before querying the LDDB.

whilst omitting from the list:

- all but one of any nodes in this path that lie in the same physical location but belong to different blocks: this occurs when the path crosses block boundaries

## 3.4 User Interface

The user interface was built using Java's Swing toolkit. The UI allows the user to:

**Generate a map** of a given resolution, coverage percentage and clustering.

**Create a map** of a given resolution using an interactive editor that has brushes and erasers of differing sizes. If the UI is in map creation mode then UI uses `MouseListener`s to detect when the mouse is being dragged over the map. The array that represents the map is updated according to the size of the brush and the resolution of the map, and then the `repaint()` method of the `JPanel` is called. Once the creation is complete, the array is passed as a parameter to a `Map` constructor, and the `Map` object is passed to the Simulator.

**Save a map** with or without paths and statistics.

**Load a map** with or without paths and statistics.

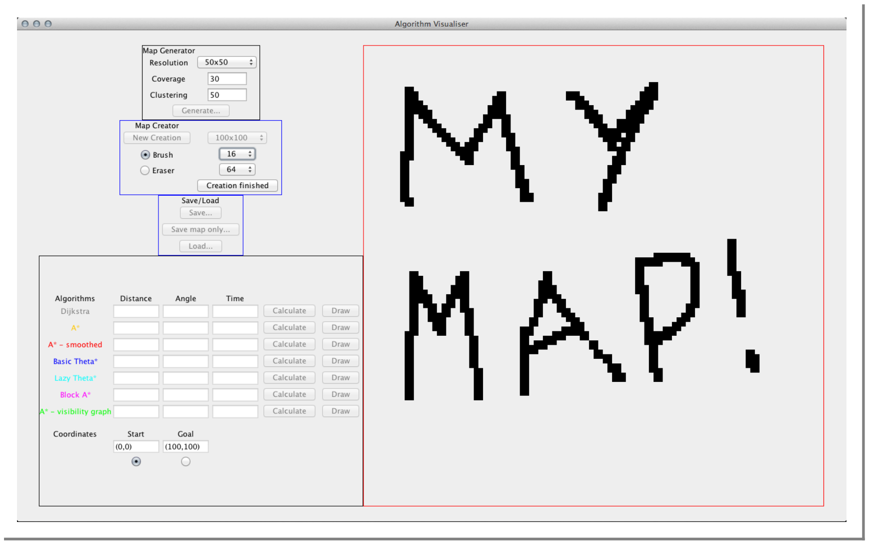
**Choose start and end points** for the paths on the current map. If the UI is not in map creation mode then `MouseListener`s are used to detect where on the screen the mouse is clicked. The start and end points can be set anywhere on the map.

**Calculate paths** for each of the 6 algorithms on the current map. If there is no path then `NoPath` will be displayed.

**Display paths** once the path has been calculated

## 3.5 Testing

To do.



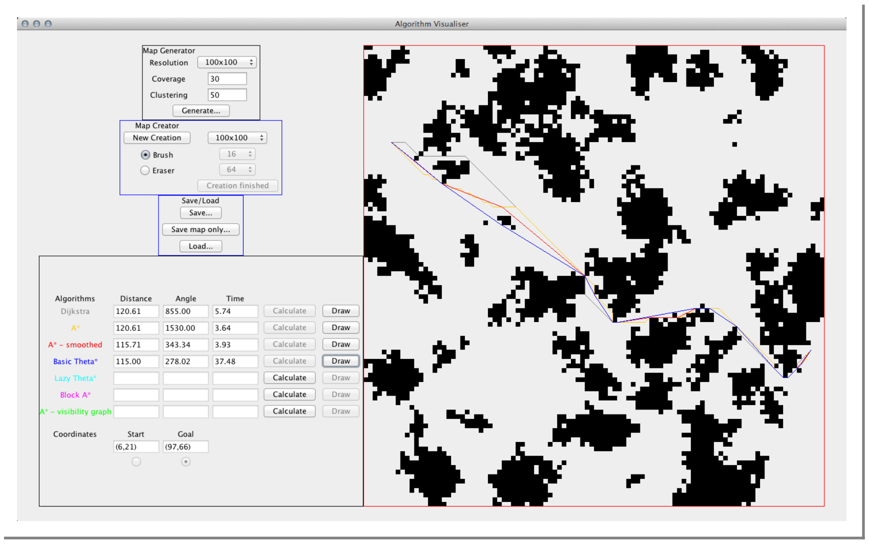
Map Generator
Resolution 50x50
Coverage 30
Clustering 50

Map Creator
100x100
[✓ Brush] 16
Eraser 64

| Algorithms | Distance | Angle | Time |
|---|---|---|---|
| Dijkstra | | | |
| A* | | | |
| A* – smoothed | | | |
| Basic Theta* | | | |
| Lazy Theta* | | | |
| Block A* | | | |
| A* – visibility graph | | | |

| Coordinates | Start | Goal |
|---|---|---|
| | (0,0) | (100,100) |
| | ● | ○ |

Figure 3.10: User interface in map creation mode



Map Generator
Resolution 100x100
Coverage 30
Clustering 50

Map Creator
100x100
Brush 16
Eraser 64

| Algorithms | Distance | Angle | Time |
|---|---|---|---|
| Dijkstra | 120.61 | 855.00 | 5.74 |
| A* | 120.61 | 1530.00 | 3.64 |
| A* – smoothed | 115.71 | 343.34 | 3.93 |
| Basic Theta* | 115.00 | 278.02 | 37.48 |
| Lazy Theta* | | | |
| Block A* | | | |
| A* – visibility graph | | | |

| Coordinates | Start | Goal |
|---|---|---|
| | (6,21) | (97,66) |
| | ○ | ● |

Figure 3.11: User interface displaying paths for a generated map

| Map | Algorithm | PathTime | TotalLength | TotalAngle |
|---|---|---|---|---|
| Map 1 | Dijkstra | 852.926976 | 160.752308679 | 495.0000736525 |
| Map 1 | AStar | 169.831936 | 160.752308679 | 855.0000688228 |
| Map 1 | AStarVisibility | 5.334016 | 151.6768359881 | 102.262577962 |
| Map 1 | AStarSmoothed | 85.908992 | 154.9127142045 | 165.3889464848 |
| Map 1 | ThetaStar | 30.230016 | 151.7637935619 | 122.1222152092 |
| Map 1 | LazyThetaStar | 32.45824 | 151.7637935619 | 122.1222152092 |
| Map 1 | BlockAStar | 10.85792 | 152.8713149055 | 566.7296353554 |
| Map 2 | Dijkstra | 615.220992 | 161.3380951166 | 810.0000676154 |

Figure 3.12: Extract from a CSV file exported by `DataExtract`

```java
void generateMaps
 (int size, int coverage, int clustering, int numberOfMaps) {

 for(int i=0;i<numberOfMaps;i++) {
  saveMapOnly(size+"/"+coverage+"/"+clustering+"/"+i+".ser",
   new MapInstance(
    MapGenerator.generateMap(size,size,coverage,clustering)));
 }
}
```

Figure 3.13: Code snippet from `DataExtraction` showing automation of `Graph` creation

## 3.6 Data extraction

To harvest enough data to do meaningful statistical analysis of the performance of the algorithms, the simulator was designed to have a simple API that allowed both a UI to be bolted on and scripts that could bypass the UI altogether. I used the open source package `CSVWriter` write the data obtained to CSV files, as these are the standard input for *R* based statistical analysis.

# Chapter 4

# Evaluation

# Chapter 5

# Conclusion