Oliver Freeman

A comparison of Any-Angle Pathfinding Algorithms for Virtual Agents

Computer Science: Part II

Clare College

March 6, 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

Original Aims of the Project ...

Work Completed

Work completed ...

Special Difficulties

None

[1] This word count was computed by `detex diss.tex | wc -w`.

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

List of Algorithms

Acknowledgements

Acknowl
includ-
ing
cita-
tions?
. . .

Chapter 1

Introduction

1.1 Motivation

Finding short and realistic-looking paths through maps with arbitrarily placed obstacles is one of the central problems in artificial intelligence for games and robotics. Figure 1.1 (a) shows a grid-based map, consisting of free cells and blocked cells. Figure 1.1 (b) shows the shortest path through this map between the *start* and the *goal*.

However, path-finding algorithms operate on graphs. A graph representation of this map is shown in figure 1.1 (c) — each node in the graph represents a coordinate on the map, and an agent can travel directly between any two nodes connected by an edge. Figure 1.1 (d) shows the optimum path through the graph — however, this is clearly longer than the optimum path through the map.

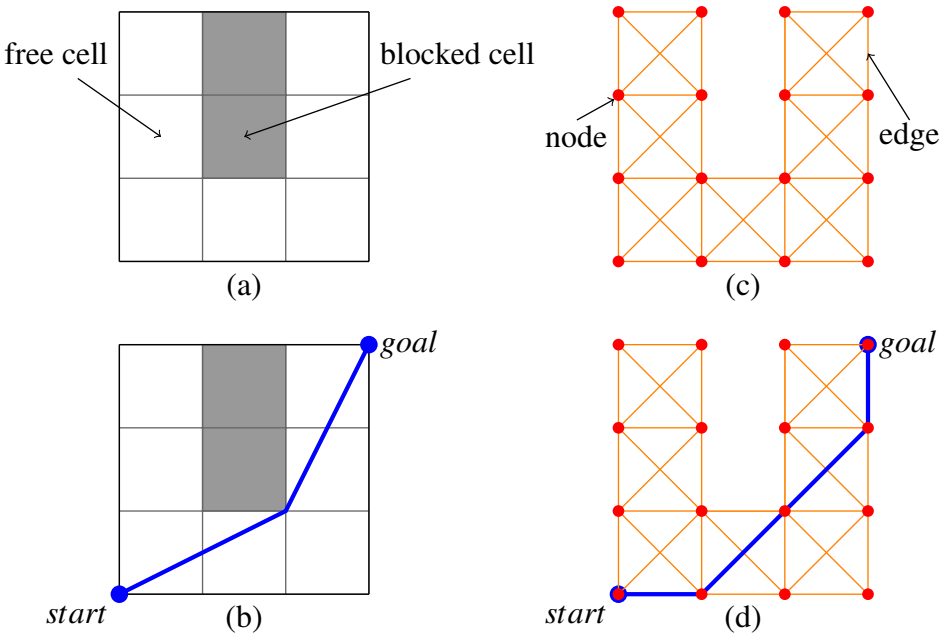


Figure 1.1: Shortest path through a map vs. shortest path through the graph representing the map

The structure of this graph is based on the grid structure of the map. If a different graph representation was chosen then a shorter path could have been achieved, but in general such a graph would require many more nodes and edges which would cause exponentially increasing memory requirements and search space size. This dissertation will investigate various algorithms that aim to find a near-optimal path through a map while using space-efficient grid-based graph like that shown in figure 1.1(c).

1.2 Related work

Applying a post-processing step to smoothe paths returned by *A** has been a technique that has been used since the earliest video games [1], but most of the research into more advanced any-angle pathfinding algorithms have taken place in the last half decade.

Ferguson and Stentz's paper on *Field D** [2] in 2006 was followed by a significant contribution to the field by Nash and Koenig et al., who published papers on *Theta** [3] and *Lazy Theta** [4] in 2010. In 2011, Yap's paper on *Block A** [5] introduced the concept of pre-calculating solutions to sub-maps to speed up execution.

Studies such as Nash and Koneig's article in *Artificial Intelligence Magazine* [6] have been conducted that compare a selection of these any-angle pathfinding algorithms, though the only established authority that utilises empirical data on *Block A** is Yap's own work [5], which compares only *A**, *Theta** and *Block A**.

1.3 Project goals

The goals of this project are to:

- create an algorithm simulation environment that enables intuitive and informative comparison of the performance of the most prominent any-angle path-finding algorithms;
- use statistical analysis to present conclusions on the suitability of these algorithms to different path-finding situations.

Therefor in this project I am aiming to reconcile the gaps in the literature by...?

Chapter 2

Preparation

2.1 Introduction to any-angle pathfinding

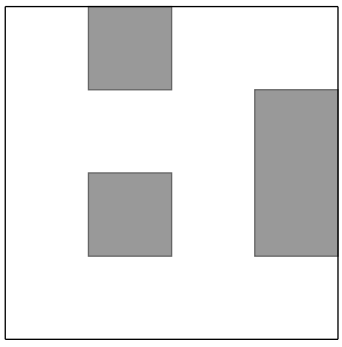
This section formally introduces the concept of maps and describes how graphs are created from maps. It then defines the any-angle path-finding problem.

2.1.1 Map

A map M of size N^2 is a square region in two-dimensional space $[0,N]^2 \subseteq \mathbb{R}^2$, where $N \in \mathbb{Z}$ and $N > 0$. A location on the map can be specified with a coordinate $(x,y) \in \mathbb{R}^2$, where $0 \leq x,y \leq N$.

The map is logically divided into a grid of N^2 cells of size 1×1, where the cell identified by the coordinate $(i,j) \in M$ includes all locations $(x,y) \in \mathbb{R}^2$ where $i \leq x \leq i+1$ and $j \leq y \leq j+1$, and each cell is either 'free' or 'blocked'.

A map M can be represented by a square boolean array A_M, where element $A_M[i][j]$ represents the cell in M with coordinate (i,j) — $A_M[i][j] = 0$ denotes that (i,j) is a free cell, whereas $A_M[i][j] = 1$ denotes that (i,j) is a blocked cell



(a) Map M

0	1	0	0
0	0	0	1
0	1	0	1
0	0	0	0

(b) Array A_M of map M

Figure 2.1: A map of size 4^2

Valid location

The agent is modelled as a dimensionless point, as is the convention [3] in any-angle pathfinding research.[1] Therefore, a valid location is defined as any location $(x,y) \in M$ that lies in or on the boundary of a free cell.

is this the correct way to reference it?

[1] 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.

Line of sight

A line of sight exists between two locations (x_0, y_0) and (x_1, y_1) on a map if all locations that lie on the straight line drawn between (x_0, y_0) and (x_1, y_1) are valid — that is to say, for all $t \in \mathbb{R}$ where $0 \leq t \leq 1$: $(x_0 + t(x_1 - x_0), y_0 + t(y_1 - y_0))$ is a valid location.

Path through a map

A path $P_M = ((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n))$ through map M is an ordered list of coordinates $(x, y) \in \mathbb{R}^2$ where $0 \leq x, y \leq N$, $x_0 = (x_{start}, y_{start})$ and $x_n = (x_{goal}, y_{goal})$. A path is valid if there exists a line of sight between every pair of coordinates (x_i, y_i) and (x_{i+1}, y_{i+1}).

It should be noted that since the agent is modelled as a dimensionless point, the path through the map shown in Figure 2.2, which features a 'diagonal blockage', is a valid path.

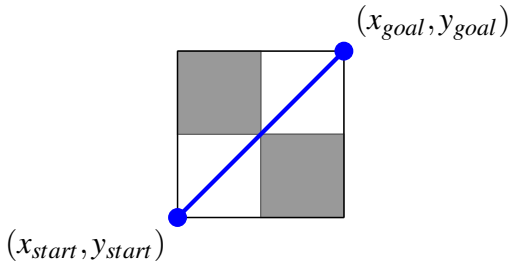


Figure 2.2: A valid path for an agent modelled as a point

2.1.2 Lattice

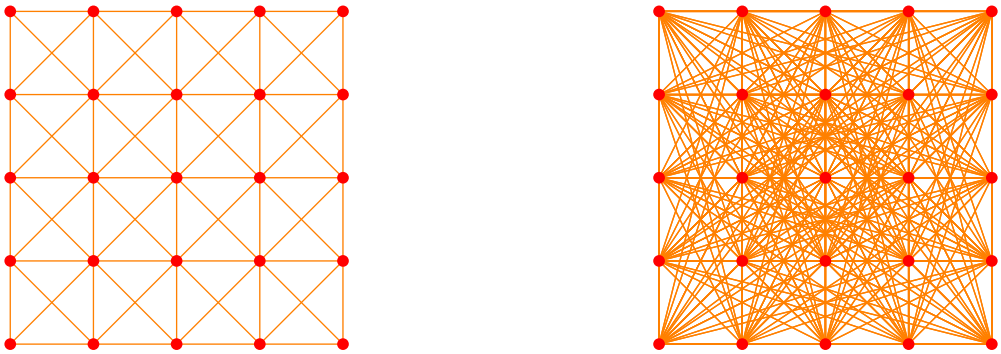
A lattice L_N is a graph of N^2 nodes, where $N \in \mathbb{Z}$ and $N > 0$.

Octile lattice $L_{N,O}$

The nodes are arranged in a square grid, and each node is connected with an edge to the closest node, if any exists, at a bearing of any integer multiple of $\frac{\pi}{4}$ radians.

Full lattice $L_{N,F}$

The nodes are arranged in a square grid, and each node is connected with an edge to every other node in the lattice.



(a) Octile lattice $L_{N,O}$ (b) Full lattice $L_{N,F}$

Figure 2.3: Octile lattice and full lattice of size 4^2

2.1.3 Graph

A graph $G_M = (V,E)$ is a discrete representation of a map M. Each node $n \in V$ represents a valid location $(a,b) \in M$, where $0 \leq a,b \leq N$. An agent can travel directly between the locations represented by two nodes n and n' if they are connected by an edge $e = (n,n') \in E$, and are thus called 'neighbours'. The weight $(n,n').weight$ is the Euclidean distance an agent would traverse by travelling between the locations represented by the nodes n and n'.

Path through a graph

A path $P_G = (n_0, n_1, \ldots, n_{n-1}, n_n)$ through graph $G_M = (V,E)$ is a list of nodes $n \in V$, where $n_0 = n_{start}$ and $n_n = n_{goal}$. A path is valid if for each pair of nodes n_i and n_{i+1} there exists an edge $(n_i, n_{i+1}) \in E$.

Discretisation

Discretisation is the process of creating a graph $G_M = (V,E)$ that represents a map M. The processes can be visualised as refining a lattice by removing edges and nodes from the lattice until the desired graph remains — where a node n is removed if the map indicates that n represents an invalid location, and an edge (n,n') is removed if there is no line of sight between the locations represented by n and n'. If a grid-based graph is desired, an octile lattice is used, whereas if a visibility graph is desired, a full lattice is used.

A more formal description of this process is now provided:

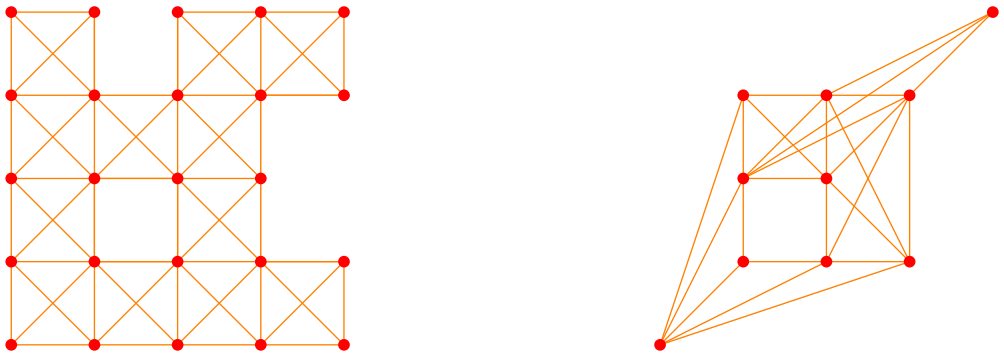
Grid-based graph: $f_G(L_O, M) \rightarrow G_{M,G}$

An octile lattice $L_{N,O}$ is laid over M such that each node in $L_{N,O}$ lies directly on top of a coordinate (x,y) in M, where $(x,y) \in \mathbb{Z}^2$. If none of the (up to) four cells surrounding a node[2] are free then that node is removed, along with all edges that are connected to it. Additionally, if a cell is blocked, diagonal edges that cross that cell are removed, along with any horizontal or vertical edges that lie beneath the blocked cell and are also on the boundary of the lattice.

Visibility graph: $f_V(L_O, M) \rightarrow G_{M,V}$

A full lattice $L_{N,O}$ is laid over M such that each node in $L_{N,O}$ lies directly on top of a coordinate (x,y) in M, where $x,y \in \mathbb{Z}^2$. Unless *either* exactly three of the (up to) four cells surrounding a node are free *or* the node is n_{start} or n_{goal} then that node is removed, along with all edges that are connected to it. In addition each edge $(n,n') \in E$ is removed if there is no line of sight between the locations that n and n' represent.

[2] In a map M of size N^2, the four cells surrounding a node that represents coordinate $(a,b) \in M$ are the cells (a,b), $(a,b-1)$, $(a-1,b-1)$ and $(a-1,b)$ if they exist, where $0 \leq a,b \leq N$.



(a) Grid-based graph $G_{M,G}$ (b) Visibility graph $G_{M,V}$

Figure 2.4: Graph representations of map M, where $n_{start} = (0,0)$ and $n_{goal} = (4,4)$

2.1.4 The any-angle pathfinding problem

The problem is to compute optimal or near-optimal paths, if they exist, through maps.

An optimal path P^*_M through map M is a path P_M has a path length that is no longer than the path length of any other paths through M. If there exists more than one path with this shortest path length, the optimal paths are the paths with the shortest path length and the smallest path angle-sum, where:

Need to sort out wording here

Path length
The length of path P_M is the sum of the Euclidean distances between each pair of coordinates (x_i, y_i) and (x_{i+1}, y_{i+1}) in P_M.

Path angle-sum
The angle-sum of path P_M is the sum of the smallest angle between each pair of path segments (x_i, y_i) to (x_{i+1}, y_{i+1}) and (x_{i+1}, y_{i+1}) to (x_{i+2}, y_{i+2}) in P_M.

2.1.5 Solving the any-angle pathfinding problem

An optimal or near-optimal path through map M is obtained by applying a pathfinding algorithm[3] to a graph G_M. The path $P_G = (n_0, n_1, ..., n_n)$ returned by the pathfinding algorithm corresponds to the path $P_M = (x_0, x_1, ..., x_n)$, where x_i is the location represented by n_i.

Classic pathfinding algorithm
A classic pathfinding algorithm will return the optimal[4] path P^*_G through G_M.

Any-angle pathfinding algorithm
An any-angle pathfinding algorithm will return the optimal path $P^*_{G'}$ through G'_M, where the augmented graph G'_M may have extra edges to G_M.

The optimal path P^*_G through a visibility graph $G_{M,V}$ directly corresponds to the optimal path P^*_M through map M, whereas the optimal path through a grid-based graph $G_{M,G}$ or an augmented graph $G_{M,G'}$ may not. However, grid-based graphs are generally accepted as the preferable form of map representation for pathfinding since, for a map M of size N^2, $G_{M,G}$ has $O(N^2)$ edges, whereas $G_{M,V}$ has $O(N^4)$ edges, which may be unfeasible for large or high resolution maps. Therefore, this investigation will focus predominantly on pathfinding algorithms applied to grid-based graphs.

[3] *Block A** operates in a different way, and will be dealt with separately.
[4] The optimal path P^*_G through graph G is defined analogously to the optimal path P^*_M through map M.

2.2 Any-angle pathfinding algorithms

This section starts with an introduction of pathfinding over graphs. It then describes two classic pathfinding algorithms: *Dijktra's shortest paths* [7] and *A** [8], followed by the four any-angle pathfinding algorithms that form the core investigation in this dissertation: *A* with post smoothing*, *Theta**, *Lazy Theta** and *Block A**.

2.2.1 Pathfinding over graphs

Section 2.1.3 introduced the concept of a graph G_M that is a representation of a map M. In addition to an associated coordinate, each node n in G_M has a set of parameters of which the values will change during the execution of the pathfinding algorithm. These parameters include:

- *g-value* — the path length of the shortest path found so far from n_{start} to n;
- *parent* — a pointer to the previous node in the shortest path found so far from n_{start} to n.

When the algorithm is initialised, $n.g = \infty$ and $n.parent = \perp$[5] for all nodes.

When the algorithm terminates, $n_{goal}.g$ is the path length of the shortest path, if one exists, found by the algorithm from n_{start} to n_{goal}, and the shortest path, if one exists, can be found by recursively following the *parent* pointers from n_{goal} to n_{start}[6].

2.2.2 Dijkstra's shortest paths

Most of the algorithms in this dissertation are derivatives of the *A** graph traversal algorithm, which itself is a derivative of *Dijkstra*'s famous shortest-path algorithm.

Explanation
Starting with n_{start}, *Dijkstra* processes (or 'expands') nodes from G_M until it processes n_{goal}, at which point it terminates. Each node is expanded at most once, since it is a provable [9] invariant of the algorithm that when a node n is selected to be processed, $n.g$ is the length of the shortest path in the graph to n from n_{start} — a set $closedSet$[7] ensures that no node is expanded twice, as this would incur unnecessary work.

Iteration
On each iteration, *Dijkstra* selects a node n to expand from $openSet$, a priority queue that stores nodes in increasing order of their *g-values*. For each n_{neigh} of the neighbours of n, *Dijkstra* attempts to 'relax' n_{neigh} — that is to say: *Dijkstra* tests whether the shortest path to n_{neigh} that has been found so far is longer than the path to n_{neigh} that goes via n:

$$n.g + (n, n_{neigh}).weight < n_{neigh}.g \tag{2.1}$$

and if (2.1) is true, *Dijkstra* updates n_{neigh} so that the shortest path to it is via n, by:

- updating the *g-value* of n_{neigh} to $n.g + (n, n_{neigh}).weight$
- updating the *parent* of n_{neigh} to n;
- adding n_{neigh} to $openSet$ if it is not already in it.

[5] Where $\perp$ denotes that the parameter value is undefined.
[6] The pathfinding algorithm guarantees $n_{start}.parent = \perp$.
[7] The set and priority queue methods $add()$, $pop()$ and $contains()$ are defined in the usual way.

Termination
This process continues until *openSet* is empty or n_{goal} has been processed, at which point the algorithm terminates. If a valid path exists, *Dijkstra* returns P^*_G. Otherwise it returns $\perp$.

Algorithm 1: DIJKSTRA

```
def Dijkstra(G, n_start, n_goal)
1   openSet ← ⊥
2   closedSet ← ⊥
3   n_start.g ← 0
4   openSet.add(n_start)
5   while openSet ≠ ⊥ do
6       n_curr ← openSet.pop()
7       closedSet.add(n_curr)
8       if n_curr = n_goal then
9           return n_goal
10      foreach n_neigh of n_curr do
11          if closedSet.contains(n_neigh) = false then
12              if Update(n_neigh) = true then
13                  if openSet.contains(n_neigh) = false then
14                      openSet.add(n_neigh)
15  return ⊥
def Update(n_neigh)
1   if n_curr.g + (n_curr, n_neigh).weight < n_neigh.g then
2       n_neigh.g = n_curr.g + (n_curr, n_neigh).weight
3       n_neigh.parent = n_curr
4       return true
5   else
6       return false
```

2.2.3 A*

*A** is based on *Dijkstra's shortest-paths* algorithm, but uses a heuristic h to reduce the number of nodes expanded.

In addition to a *g-value* and a *parent*, each node also has a:

- *h-value* — Euclidean distance between n and n_{goal}: a cheaply computable monotonic estimate[8] of the actual shortest path length between n and n_{goal}.

Arbitrary choice between node having h-value or f-value?

While *Dijkstra* preferentially expands nodes with low *g-value*s, *A** preferentially expands nodes with low *f-score*s, where a node n's *f-score* is the algorithms current estimate of the shortest path from n_{start} to n_{goal} that goes via n — that is to say:

$$n.f = n.g + n.h \tag{2.2}$$

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

[8] The monotonicity of Euclidean distance as a heuristic ensures that *A**, like *Dijkstra*, is complete (if a path exists, it finds it) and optimal (if a path is found, it is a shortest distance path)

Algorithm 2: `Update` from A*

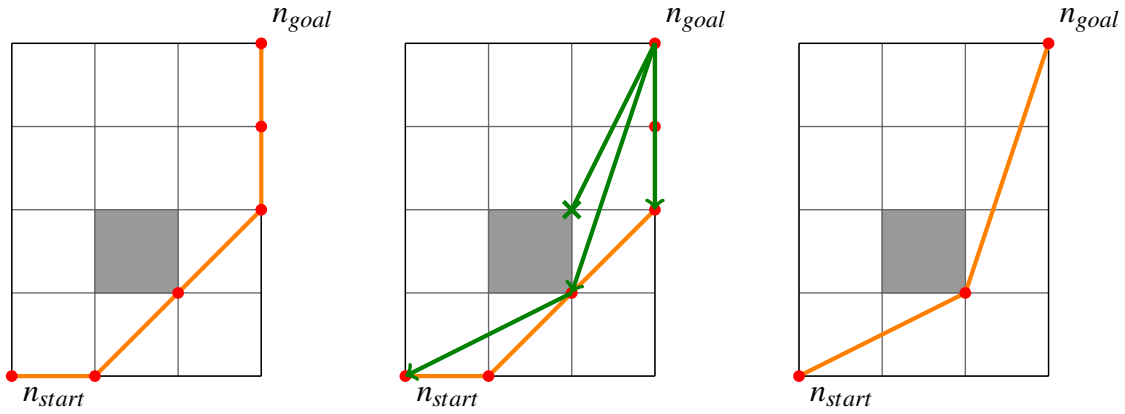
```
def Update(n_neigh)
1   if n_curr.g + (n_curr, n_neigh).weight < n_neigh.g then
2       n_neigh.g ← n_curr.g + euclidean(n_curr, n_neigh)
3       n_neigh.h ← euclidean(n_neigh, n_goal)
4       n_neigh.parent = n_curr
5       return true
6   else
7       return false
```

2.2.4 A* with post-smoothing

A post-processing step is run on the path $P_G = (n_{start}, n_1, \ldots, n_{n-1}, n_{goal})$ returned by $A*$, which attempts to reduce the path distance by changing the *parent* pointer of some of the elements of P_G — this has the effect of adding new edges to the graph G to create a new graph G'.

Iteration
On each iteration, the algorithm considers a node n_{curr}. Starting with $n_{curr} = n_{goal}$ a line of sight test is performed from the coordinate associated with n_{curr} to the coordinate associated with $n_{curr}.parent.parent$. If a line of sight exists, $n_{curr}.parent$ is set to $n_{curr}.parent.parent$ — this has the effect of adding an edge $(n_{curr}, n_{curr}.parent.parent)$ to G. This process is repeated on n_{curr} until a line of sight test fails, at which point n_{curr} is set to $n_{curr}.parent.parent$[9], and the next iteration commences.

Termination
When $n = n_{start}$, the algorithm terminates.



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

Figure 2.5: A* with post-smoothing

2.2.5 Theta*

Where *A* with post-smoothing* performs an explicit smoothing step after finding the basic path with *A**, *Theta** interleaves smoothing with exploration by attempting to re-parent each of n_{curr}'s neigh-

[9]Note: $n_{curr}.parent.parent$ is the node on which the line of sight test failed.

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

```
def PostSmoothing(n_start, n_goal)
1   n_curr ← n_goal
2   n_next ← n_goal.parent.parent
3   if n_next = ⊥ then
4       return
5   while true do
6       while LineOfSight(n_curr, n_next) do
7           n_curr.parent ← n_next
8           n_next ← n_next.parent
9           if n_next = n_start then
10              return
11      n_curr ← n_next
12      if n_curr.parent = n_start then
13          return
14      n_next ← n_next.parent.parent
```

bours n_{neigh} with $n_{curr}.parent$ at the time when n_{curr} is expanded[10].

The pseudocode for *Theta** differs only from *A** in the `update` subroutine. If a line of sight exists between the coordinates represented by n_{neigh} and $n_{curr}.parent$, *Theta** (notionally) creates an edge $e = (n_{neigh}, n_{curr}.parent) \in G$ and attempts to relax e. If the line of sight does not exist, e is removed from G and *Theta** mimics *A** by attempting to relax the edge (n_{neigh}, n_{curr})[11].

Algorithm 4: `Update` from THETA*

```
def Update(n_neigh)
1   if LineOfSight(n_neigh, n_curr.parent) = true then
2       if n_curr.parent.g + (n_curr.parent, n_neigh).weight < n_neigh.g then
3           n_neigh.g ← n_neigh.parent.g + (n_curr.parent, n_neigh).weight
4           n_neigh.f ← euclidean(n_neigh, n_goal)
5           n_neigh.parent ← n_curr.parent
6           return true
7       else
8           return false
9   else
10      if n_curr.g + (n_curr, n_neigh).weight < n_neigh.g then
11          n_neigh.g ← n_curr.g + (n_curr, n_neigh).weight
12          n_neigh.f ← euclidean(n_neigh, n_goal)
13          n_neigh.parent ← n_curr
14          return true
15      else
16          return false
```

[10] As with *A* with post-smoothing*, re-parenting in *Theta** occurs if a line of sight exists between the coordinates represented by the two nodes in question.

[11] As per the `update` subroutine of *A**.

2.2.6 Lazy Theta*

*Lazy Theta** attempts to refine *Theta** by finding similar paths despite performing fewer line of sight tests. Where *Theta** performs a line of sight for every neighbour n_{neigh} of every node n_{curr} that is expanded, *Lazy Theta** avoids unnecessary line of sight tests by only performing them for every node n_{curr} that is expanded.

Iteration
For each n_{curr} that is expanded, *Lazy Theta** assumes that a line of sight exists between $n_{curr}.parent$ and each neighbour n_{neigh} of n_{curr}, and updates the *g-value* and *parent* of each n_{neigh} accordingly[12]. *Lazy Theta** only actually performs that line of sight test if the n_{neigh} itself is ever expanded (henceforth referred to as n'_{curr}, for clarity), by calling `Initialise` when n'_{curr} is popped off $openSet$. If the line of sight doesn't in fact exist, `Initialise` alters n'_{curr} accordingly[13].

Algorithm 5: `Initialise` and `Update` from LAZY THETA*

```
   def Initialise(n_curr)
1  |  if LineOfSight(n_curr, n_curr.parent) = false then
2  |  |  newParent ← argmin_{n' ∈ expandedNeigh(n_curr)} (n'.g + (n', n_curr).weight)
3  |  |  n_curr.parent ← n'
4  |  |  n_curr.g ← n'.g + (n', n_curr).weight
   def Update(n_neigh)
   |  // assume line of sight test passes
5  |  if n_curr.parent.g + (n_curr.parent, n_neigh).weight < n_neigh.g then
6  |  |  n_neigh.g ← n_neigh.parent.g + (n_curr.parent, n_neigh).weight
7  |  |  n_neigh.f ← euclidean(n_neigh, n_goal)
8  |  |  n_neigh.parent ← n_curr.parent
9  |  |  return true
10 |  else
11 |  |  return false
```

It should be noted that when run on a given map, *Lazy Theta** does not necessarily return the same path as *Theta**.

2.2.7 Block A*

*Block A** is the most complex algorithm in this dissertation. It was published in 2011, and is at the very cutting edge of any-angle path-finding algorithmic research. This dissertation attempts a novel explanation of *Block A** that draws closer parallels with the graph-based explanations provided in the previous sections for *Dijkstra* and *A**.

Overview
*Block A** divides a map *M* into logical blocks and then finds an optimal or near-optimal path by proceeding through *M* in a block by block fashion, as opposed to node by node like the other algorithms studied in this dissertation. This gives *Block A** the potential for fast evaluation. *Block A** uses the concept of a *Local Distance Database* to provide information about shortest paths *through* blocks without having to explicitly calculate them.

[12] *Lazy Theta** performs the update part of relaxation without first performing the test in equation (2.1)
[13] See **Algorithm 5** lines 2-4

Blocks
Where all of the previously introduced algorithms operate on a simple graph structure, *Block A** operates on a structure based on blocks. A block $B_{i,j}$ of size N_B^2, identified by the coordinate $(i,j) \in \mathbb{Z}^2$, is a graph that represents the subsection of a map M containing all locations $(x,y) \in \mathbb{R}^2$, where $i \leq x \leq i+N_B$ and $j \leq y \leq j+N_B$.

For each block $B_{i,j} = (V_{i,j}, E_{i,j})$, $n \in V_{i,j}$ represents a location $(a,b) \in \mathbb{Z}^2$ in M where (a,b) lies on the boundary of the $B_{i,j}$ — that is to say, where $a = i$ or $a = i+N_B$, and $b = j$ or $b = j+N_B$. In addition to an associated coordinate, each node $n \in B_{i,j}$ has the parameters:

- *g-value* — the path length of the shortest path found so far from *start*[14] to n;
- *h-value* — Euclidean distance between n and *goal*[15].
- *parent* — the previous node in the shortest path found so far from *start* to n

Each *block* $B_{i,j}$ maintains an unordered $openSet_{i,j} \subseteq V_{i,j}$, and a $heapValue_{i,j}$. $heapValue_{i,j}$ is the smallest *f-score* of the nodes in $openSet_{i,j}$[16], and is used to order blocks in $openSet_{blocks}$, a priority queue that stores blocks in increasing order of their *heapValue*s.

If the locations represented by $B_{i,j}$ contain the location *start* or *goal* then $B_{i,j}$ is called B_{start} or B_{goal} respectively, and $B_{i,j}$ has an extra node n_{start} or n_{goal} with an edge from that node to every other node in the block. For all other blocks, called 'regular' blocks, there is an edge between every pair of nodes in $B_{i,j}$. See figure 2.7. For all edges $(n,n') \in B$, $(n,n').weight$ *is the length of the shortest path in map M between the two locations represented by n and n′, and* $0 \leq (n,n').weight \leq \infty$. This important distinction bears clarificiation: in grid-based graphs and visibility graphs, an edge denotes that an agent can travel in a straight line between the two nodes that the edge connects, whereas an edge in a block merely denotes that an agent can travel between those two nodes *on some path within that block*, where the length of that path is the edge weight which may range from 0 to ∞. See figure 2.7

The edge weights for regular blocks are discovered by querying the *Local Distance Database (LDDB)*, which is introduced in the next subsection. Since the *LDDB* only contains data for regular blocks, the weights of edges in B_{start} and B_{goal} are calculated separately — this is discussed in the Implementation chapter of this dissertation.

Each block $B_{i,j}$ contains nodes that are collocational with nodes from up to six neighbouring blocks. *Block A** maintains the invariant for two collocational nodes n and n' that $n.g = n'.g$ and $n.h = n'.h$.

Local Distance Database — *LDDB*
An $LDDB_N$ is a pre-computed database that holds the path length and inflection points of the optimum paths between all pairs of locations (a,b) and (c,d) for all map configurations M of size N^2, where $a,b,c,d \in \mathbb{Z}^2$ and (a,b) and (c,d) lie on the boundary of M.

Initialisation
For all $n \in B_{start}$, $n.g$ is set to $(n_{start},n).weight$. For all $n \in B_{goal}$, $n.h$ is set to $(n,n_{goal}).weight$. B_{start} and B_{goal} are treated as special cases as there is no guarantee that *start* and *goal* lie on the boundary of a block. The exceptional case that $B_{start} = B_{goal}$ must also be checked for.

[14] In *Block A** there is no explicit node n_{start} or n_{goal} — *start* and *goal* are coordinates $(x,y) \in \mathbb{R}^2$ in M.

[15] Except when the block is B_{goal}, when the *h-value* is the length of the shortest path between n and *goal* — this is discussed in the 'Initialisation' step.

[16] If $openSet_{i,j} = \perp$, $B_{i,j}.heapValue = \infty$.

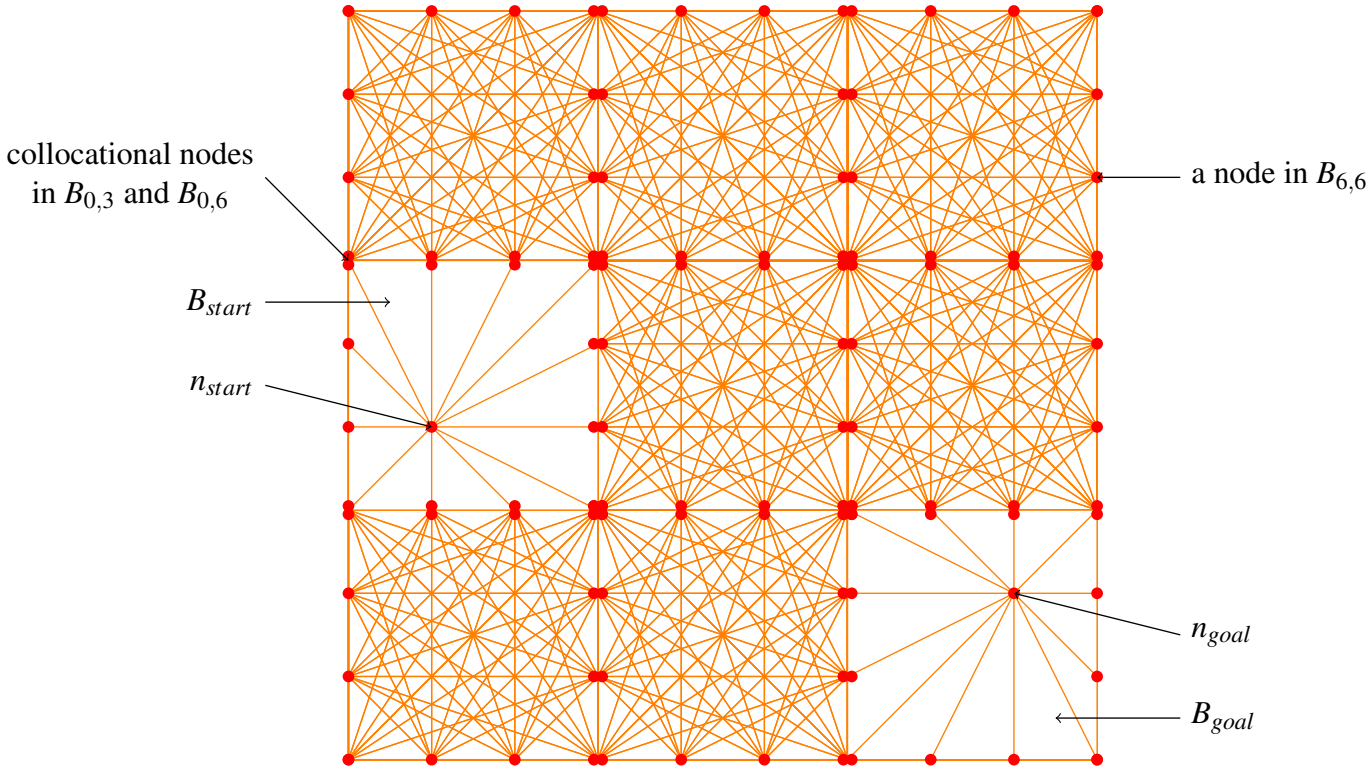


Figure 2.6: Logical view of a map of size 9^2 as a set of 9 blocks of size 3^2

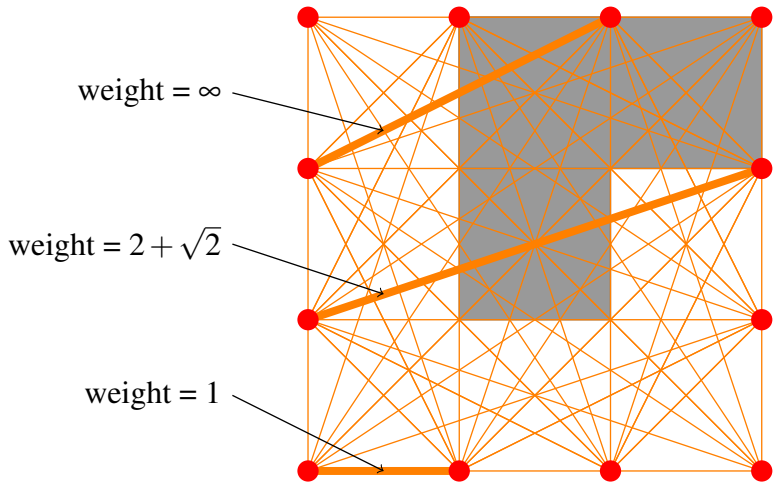


Figure 2.7: A block of size 3^2 laid over the area of map it represents, with certain highlighted edge weights labeled.

Having initialised B_{start} and B_{goal}, B_{start} is added to $openSet$.

Iteration
The iteration stage of *Block A** proceeds similarly to *A**, though *Block A** expands blocks chosen from $openSet_{blocks}$ whereas *A** expands nodes. However, note that a block can be expanded multiple times, whereas in *A** a node is only expanded once[17]. For this reason, a $length$ variable is maintained, which ensures that the algorithm terminates only when it is impossible to find a shorter path.

Expansion
On expansion of $B_{i,j}$, *Block A** attempts to relax every edge $e = (n_{ingress}, n_{egress}) \in E_{i,j}$, where $n_{ingress} \in openSet_{i,j}$ and $n_{egress} \in V_{i,j}$. If e is relaxed, all nodes that are collocational with n_{egress} are added to the $openSet_{k,l}$ of their respective block $B_{k,l}$, and $B_{k,l}$ is added to $openSet$ if it is not already a member.

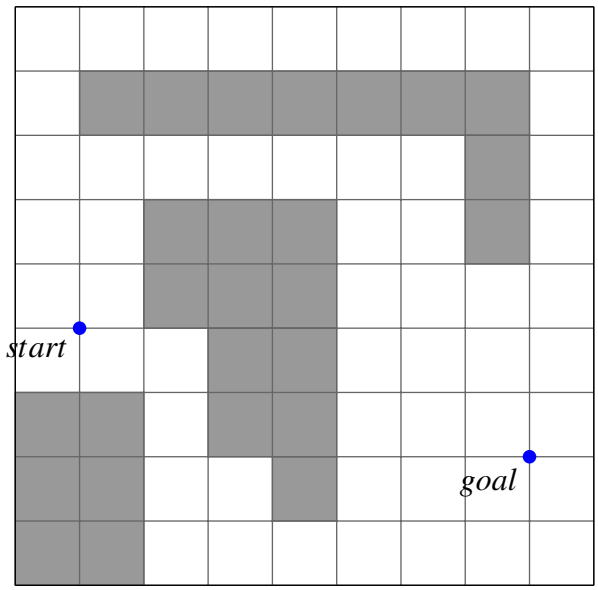
Termination
The *h-value* of B_{goal} is the actual shortest path from each node $n \in V_{goal}$, (see 'Initialisation' section) as opposed to a Euclidean estimate that is used for the *h-value* in every other block. Therefore, if $B_{goal}.heapValue$ is less than the *heapValue* of any other blocks in $openSet$ then there are no possible shorter paths to B_{goal}, so the algorithm terminates. This condition is ensured by the $length$ variable.

Traceback
When *Block A** terminates, the boundary nodes of the blocks through which P_G passes can be found by recursively following the *parent* pointers from n_{goal} to n_{start}. See figure 2.8 (b). However, to recover the nodes of P_G that do not lie on the boundaries of blocks, a traceback stage is required which involves multiple queries to the *LDDB*. See figure 2.8 (c). Although the details of this are not presented in Yap's paper, an explanation is presented in the Implementation chapter of this dissertation.

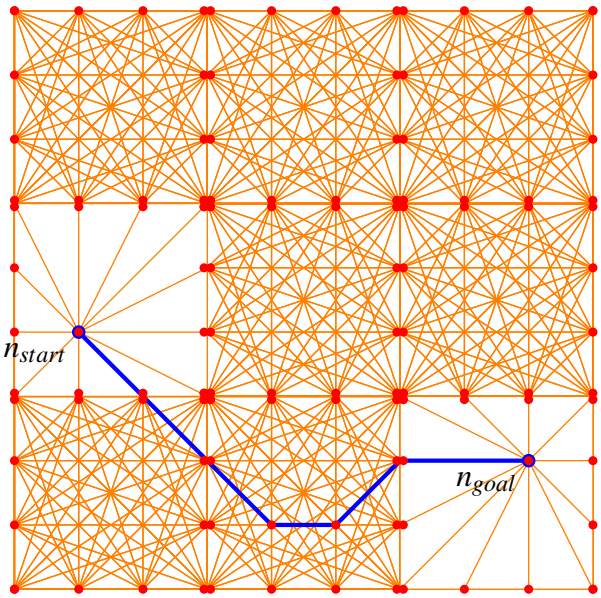
[17]This subtlety caused me such confusion that I eventually emailed Peter Yap, the author of the Block A* paper, for clarification. He explained the difference, and told me that he had included an explanation of this specific point when he gave a presentation [citation] on Block A* as the confusion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

Algorithm 6: BLOCK A*

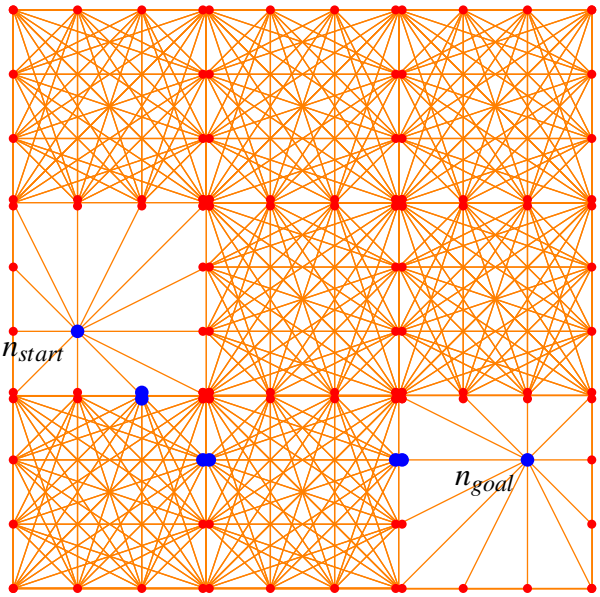
```
def BlockAStar(G, n_start, n_goal)
    B_start ← init(n_start)
    B_goal ← init(n_goal)
    length ← ∞
    openSet_blocks.add(B_start)
    while (openSet_blocks ≠ ⊥) ∧ ((openSet_blocks.peek()).heapValue < length) do
        B_curr ← openSet_blocks.pop()
        openSet_curr ← B_curr.openSet
        if B_curr = B_goal then
            length ← min_{n∈openSet_curr} (n.g + euclidean(n, n_goal), length)
        Expand(B_curr, openSet_curr)
    if length ≠ ∞ then
        TraceBack(n_goal)
    else
        return ⊥
def Expand(B_curr, openSet_curr)
    while openSet_curr ≠ ⊥ do
        n_ingress ← openSet_curr.pop()
        foreach n_egress ∈ B_curr do
            if n_ingress.g + (n_ingress, n_egress).weight < n_egress.g then
                n_egress.g ← n_ingress.g + (n_ingress, n_egress).weight
                foreach n' ∈ n_egress.collocational do
                    n' ← n_ingress.g + (n_ingress, n_egress).weight
                    openSet_n'.coord.add(n')
                    if openSet_blocks.contains(B_n'.coord) = false then
                        openSet_blocks.add(B_n'.coord)
```



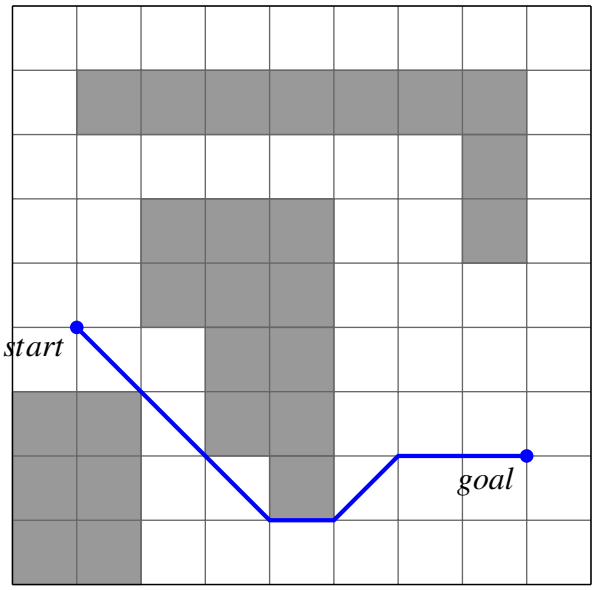
(a) Original map M



(c) Traceback stage complete



(b) Boundary nodes of path identified



(d) Path P_M through map M

Figure 2.8: Solution of *Block A**

2.3 Requirements analysis

As specified in the **Work to be done** section of the Proposal (see **Appendix E**), this project is logically composed of four parts. This section outlines the functional and non-functional requirements for each part, 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

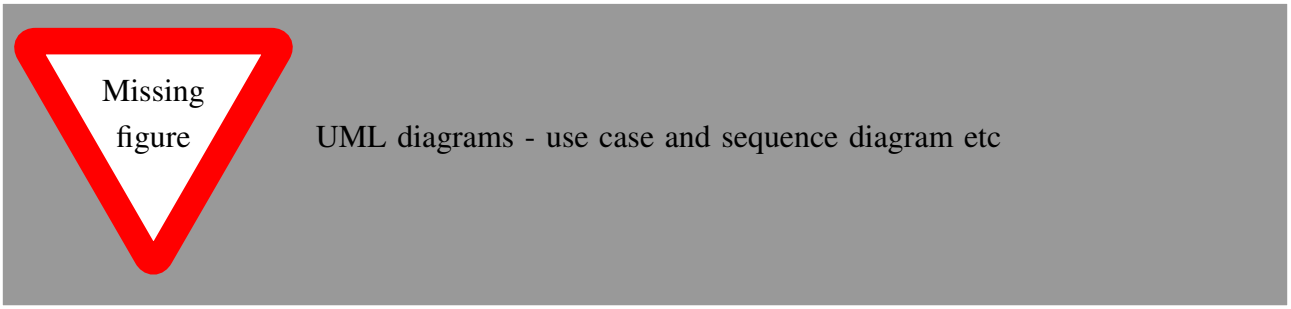
ID	Non-functional requirement	Priority
1	The system shall generate maps of the highest resolution in under 2 seconds	S

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 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



2.4 Testing

A thorough testing strategy was devised to reduce the chances of bugs being introduced into the code. Separate strategies were required for different modules of the system:

Simulator

Core structural classes — basic functionality to be unit tested;
Line of sight algorithm — to be unit tested thoroughly, as its correctness will be assumed during testing of pathfinding algorithm implementations;
Pathfinding algorithms — correctness of all paths returned by to be optionally verified.

User interface

Use case scenarios — to be manually tested by human user.

Data extraction scripts

Exported CSVs — contents of CSV files to be unit tested.

2.5 Design model

Once the preparation phase of the project was completed, the plan for the implementation phase was refined 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 graphs, and *Dijkstra* 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
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

This section justifies the languages, libraries and tools that were chosen for this project.

Programming language
Java — provides abstraction and class hierarchy to enable development of modular, extensible code.

Libraries
Swing — for graphical user interface design;
CSVWriter — for data export.
JUnit — for unit testing.

Integrated development environment
Eclipse — allows rapid development through integrated testing, refactoring and version control tools.

Statistical analysis and visualisation
R — open-source statistical package; chosen due to its flexibility and extensibility.

Document preparation

LaTeX — allows precise, integrated control over all aspects of document layout and style;
Tikz — drawing package for LaTeX that enables programmable diagram creation;
algorithm2e — pseudo-code package for LaTeX that enables integration with the parent document.

Backup

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

Version Control

GitHub — facilitated exploring different implementation strategies by forking my core code repository.

Chapter 3

Implementation

This chapter provides an overview of the implementation of the project, and investigates some of its more interesting features.

The chapter is 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
: details of the implementation of each of the pathfinding algorithms, with specific attention given to *Block A**.

User Interface
: an overview of the graphical user interface.

Data extraction
: an explanation of how large volumes of data are extracted for statistical analysis.

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

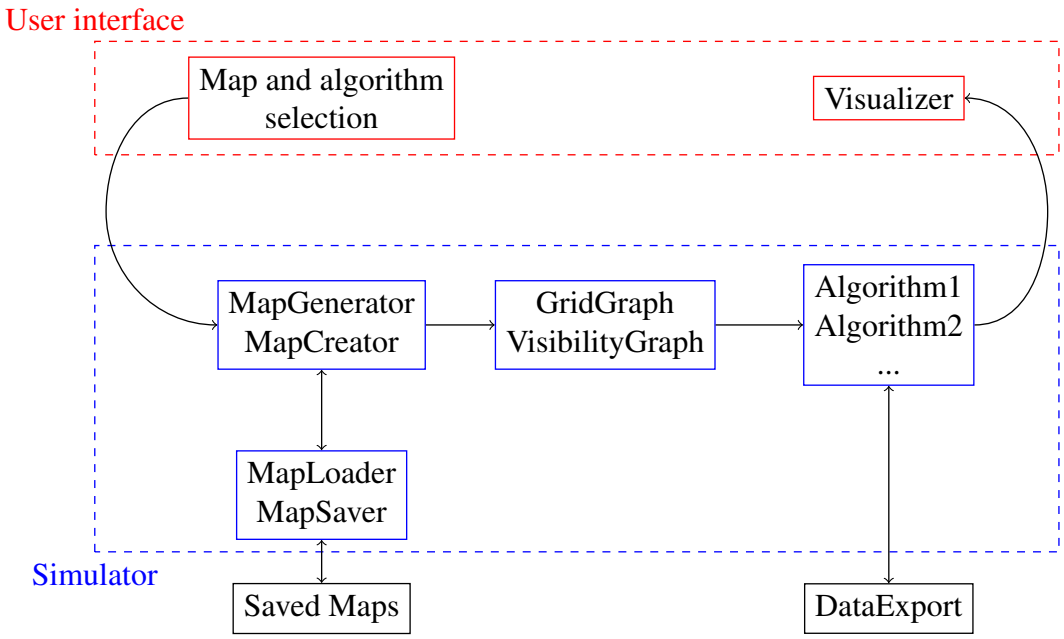


Figure 3.1: Flow of the user interface and simulator

3.1 Map generation

In order to reach reliable conclusions about the performance of the algorithms, a method was required to generate large volumes of maps of a given size N^2, coverage percentage C and clustering D was require. A bespoke was devised for this project to pseudo-randomly create such maps. The algorithm creates maps of a specified level of 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 [1], 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.

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

[1] Note that this is not the same as our notation in the Preparation chapter, where a 0 denoted a free cell. The notation of 0 denoting a blocked cell is *only* used during the execution of the map generation algorithm.

Algorithm 7: GENERATEMAP

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

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.

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

```
1  ...
2  Coordinate[] diagonalRelativeCellCoordinates =
3    {new Coordinate(-1,-1), new Coordinate(0,-1),
4    new Coordinate(0,0), new Coordinate(-1,0)};
5
6  Node[][] graphArray2D =
7   new Node[map.getWidth()+1][map.getHeight()+1];
8  for(int j=0; j < map.getHeight()+1; j++) {
9   for(int i=0; i < map.getWidth()+1; i++) {
10   boolean isUnblockedAdjacentCell = false;
11   for(Coordinate c: diagonalRelativeCellCoordinates) {
12    try {
13     if(!map.getCell(i+c.getX(),j+c.getY()).isBlocked()) {
14      isUnblockedAdjacentCell=true;
15     }
16    } catch (ArrayIndexOutOfBoundsException e) {}
17   }
18   if(!isUnblockedAdjacentCell) {
19    graphArray2D[i][j] = null;
20   } else {
21    graphArray2D[i][j] = new Node(new Coordinate(i,j));
22   }
23  }
24 }
25 ...
```

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

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 x is increased, but decreased by dx if y is increased. If at any point $f = 0$ 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.

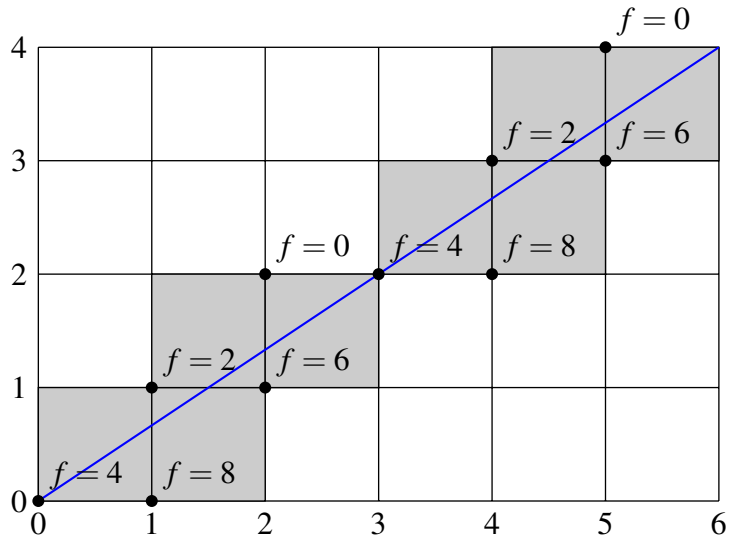


Figure 3.4: Line of sight algorithm

`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 $getPath(n_{start}, n_{goal})$ 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 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 maximum 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.

Algorithm 8: LINEOFSIGHT

```
def LineOfSight (n_source, n_goal)
1   x ← n_source.x
2   y ← n_source.y
3   x_goal ← n_goal.x
4   y_goal ← n_goal.y
5   dx ← x_goal − x
6   dy ← y_goal − y
7   f ← 0
8   while x ≠ x_goal do
9       f ← f + dy
10      if x ≥ dx then
11          if cell_{x,y}.isBlocked() then
12              return false
13          y ← y + 1
14          f ← f − 1
15      if f ≠ 0 ∧ cell_{x,y}.isBlocked() then
16          return false
17      if dy = 0 ∧ cell_{x,y}.isBlocked() ∧ cell_{x,y−1}.isBlocked() then
18          return false
19      x ← x + 1
20  return true
```

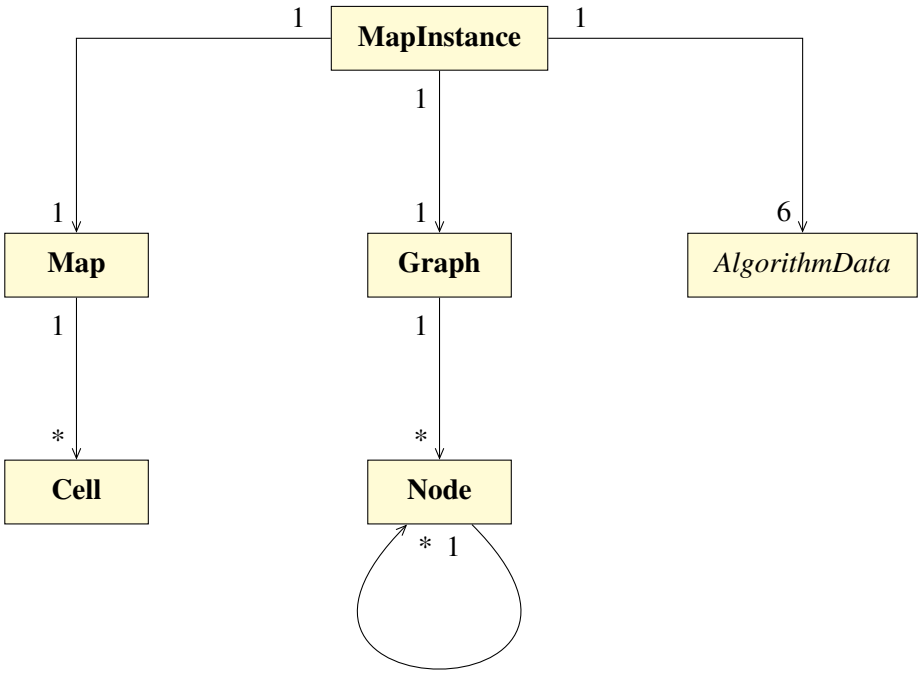


Figure 3.5: Composition of `MapInstance`

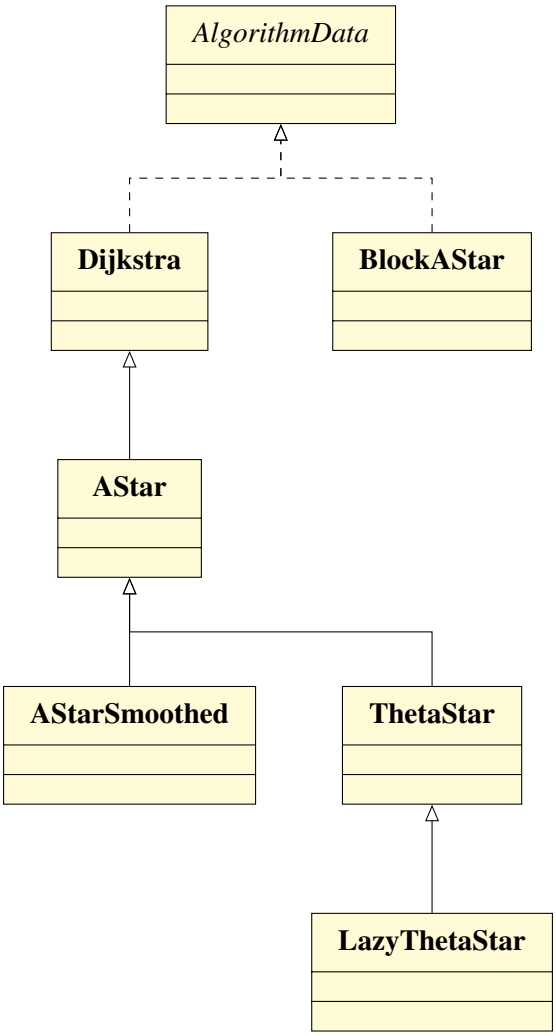


Figure 3.6: Inheritance structure of algorithms

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 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[2] altered while residing in the $openSet$, so I 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* with post-smoothing and implements its post-smoothing by overriding the $postProcessing()$ function.

3.3.4 Basic θ^* and Lazy θ^*

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

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

Lazy θ^*
inherits from Basic θ^*, 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 create the entries manually for the LDDB for block sizes of 2×2 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

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

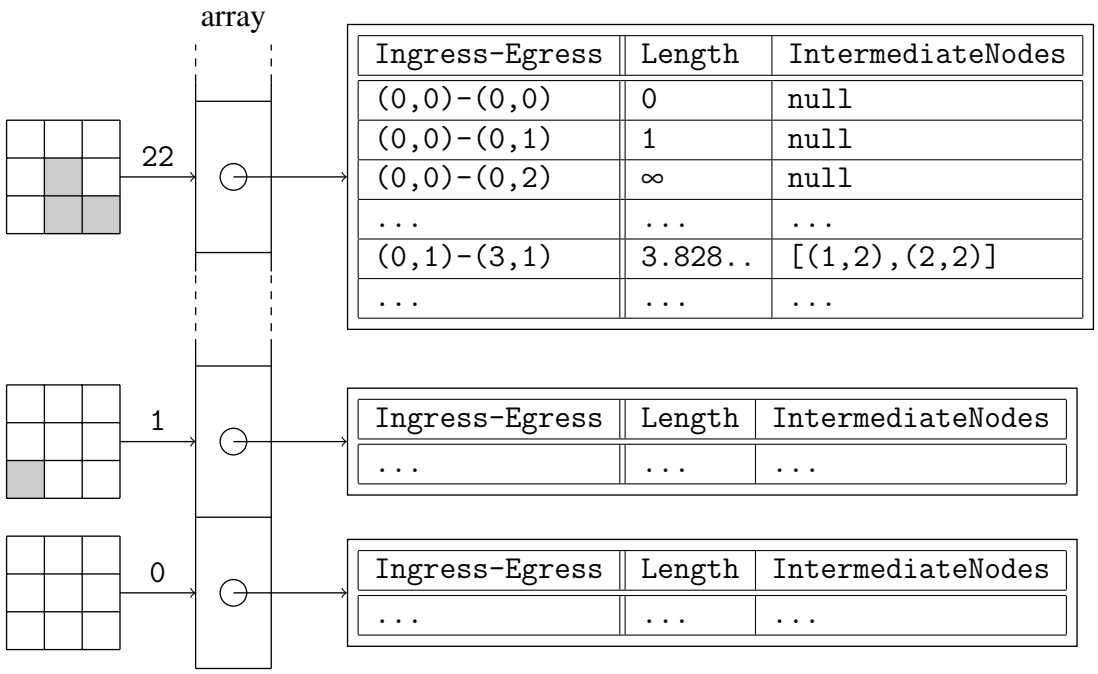


Figure 3.7: Extract of the LDDB for block size of 3×3 - array of `HashTable<PairOfCoords,Pair<double,List<Coord>>`s

block size of 4×4.

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 sufficiently specialised 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 the shortest path 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 blocks as integers that would be much more space and time efficient than storing `Map` objects in the database to use as comparison: each cell 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: 2×2 to 4×4.[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 `Coordinates`, to a value: a `Pair` consisting of the length (as a `double`) of the shortest path and an `ArrayList` of the `Coordinates` on the path.

```
1  PairOfCoords(Coordinate c1, Coordinate c2, int blockSize) {
2    this.blockSize = blockSize;
3    this.coordCode = (byte)
4     (c1.getX() +
5     c1.getY() * (blockSize+1) +
6     c2.getX() * (blockSize+1)*(blockSize+1) +
7     c2.getY() * (blockSize+1)*(blockSize+1)*(blockSize+1));
8   }
```

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

```
1  int getListCode(List<Coordinate> intermediateNodes) {
2    int listCode = 0;
3    for(Coordinate c : intermediateNodes) {
4     listCode = listCode | c.getX();
5     listCode = listCode << 3;
6     listCode = listCode | c.getY();
7     listCode = listCode << 3;
8    }
9    listCode = listCode >>> 3; //undoes line 7 on final loop
10   return listCode;
11  }
```

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

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 `Coordinates` can be represented with a unique integer representable in a `byte`'s worth of space - so a coding 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 4×4 is four[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]
2×2	33KB	17KB
3×3	2.2MB	1.1MB
4×4	485MB	192MB

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

[1]The exponential increase in size of LDDB would mean that the LDDB for block sizes of 5×5 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 2×2 and 3×3 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 size2×2 and 3×3, 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 Yap's paper, but his agents were only allowed to travel

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 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
2×2	12.5%	26.5%
3×3	33.3%	77.8%
4×4	56.3%	144.1%

The use of the fully-extended LDDB allows the 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}

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.

horizontally and vertically, so only 4 bits were needed to store the path length value.

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

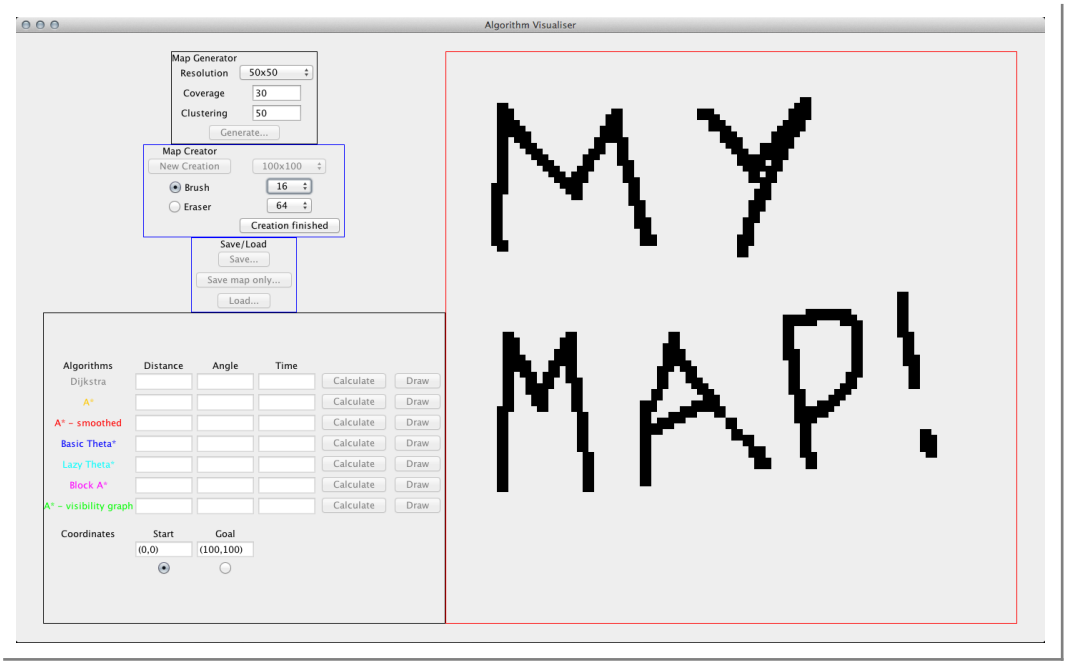


Figure 3.10: User interface in map creation mode

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 `MouseListeners` 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 `MouseListeners` 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

Impleme
testing
...

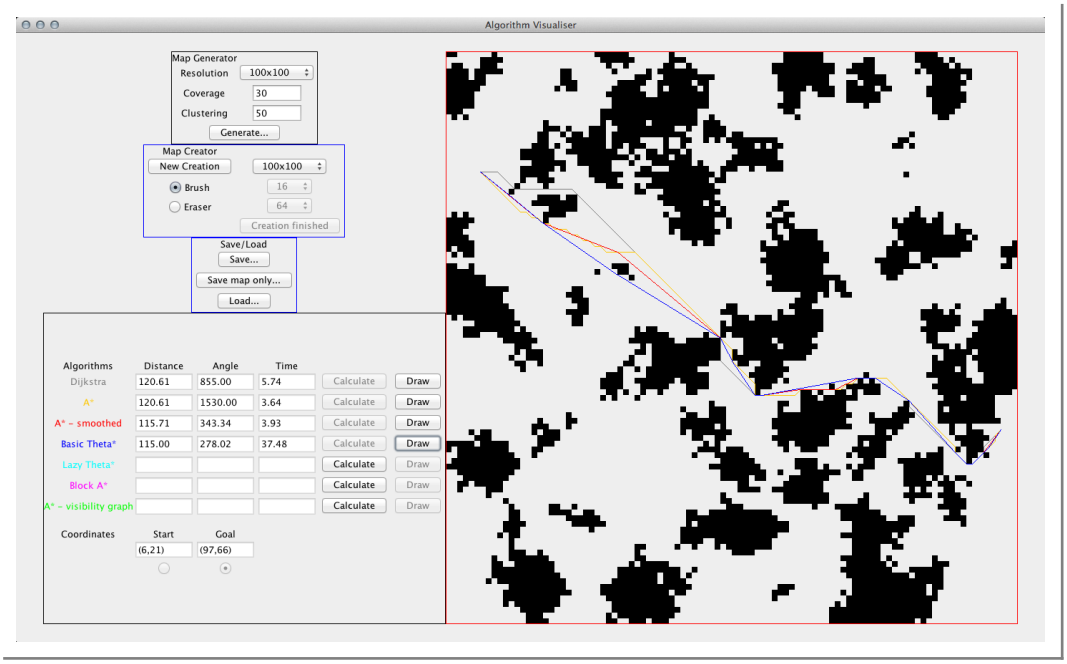


Figure 3.11: User interface displaying paths for a generated map

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`

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` to write the data obtained to CSV files, as these are the standard input for *R* based statistical analysis.

```
1  void generateMaps
2   (int size, int coverage, int clustering, int numberOfMaps) {
3
4   for(int i=0;i<numberOfMaps;i++) {
5    saveMapOnly(size+"/"+coverage+"/"+clustering+"/"+i+".ser",
6     new MapInstance(
7      MapGenerator.generateMap(size,size,coverage,clustering)));
8   }
9  }
```

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

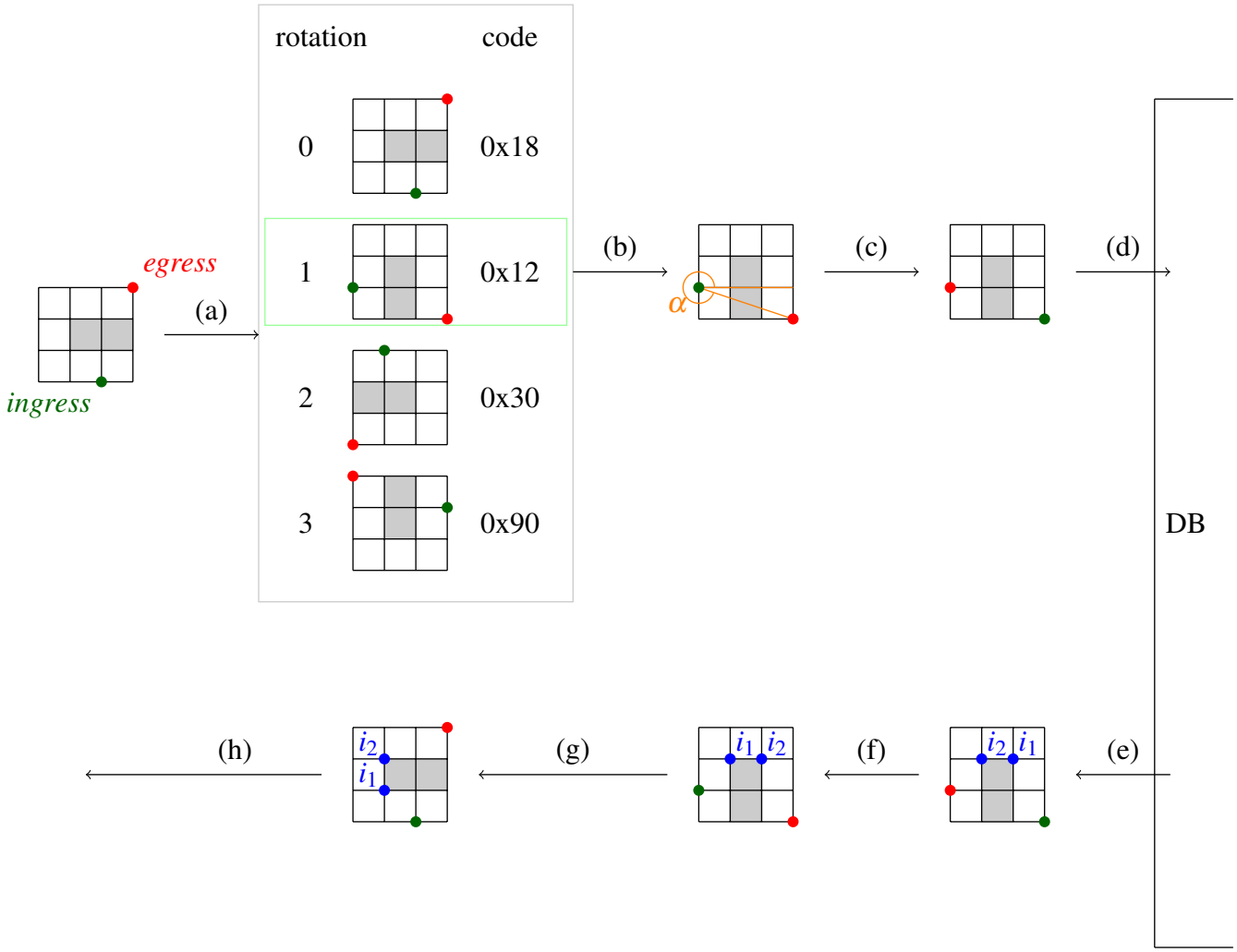


Figure 3.14: my caption

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

[1] C. Thorpe. Path relaxation: Path planning for a mobile robot and reference manual. *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 318–321, 1984.

[2] D. Ferguson and A. Stentz. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, 23(2):79–101, 2006.

[3] K. Daniel, A. Nash, S. Koenig, and A. Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[4] A. Nash, S. Koenig, , and C. Tovey. Lazy Theta*: Any-angle path planning and path length analysis in 3D. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

[5] P. Yap, N. Burch, R. Holte, and J. Schaeffer. Block A*: Database-driven search with applications in any-angle path-planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2011.

[6] A. Nash and S. Koenig. Any-angle path planning. *Artificial Intelligence Magazine*, 34(4):85–107, 2013.

[7] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[8] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *EEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[9] K. Cormen, C. Leiserson, R. Rivest, and S. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.