Oliver Freeman

A comparison of Any-Angle Pathfinding Algorithms for Virtual Agents

Computer Science: Part II

Clare College
University of Cambridge

May 10, 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:	**11,944**[1]
Project Originator:	**Oliver Freeman**
Supervisor:	**Dr. R. J. Gibbens**

Original Aims of the Project

The aims of this project were to compare and contrast the structure and performance of a selection of recently published any-angle pathfinding algorithms.

Work Completed

The pathfinding simulator allows a user to choose or create a map and specify any of the pathfinding algorithms to find a path through that map. The simulator displays the path found along with various statistics about that map. Statistics can be automatically generated for large volumes of maps using the simulator's API.

Evaluation of the statistics produced by the simulator shows results consistent with the literature, and alternative implementation strategies are also compared. Discussion of the algorithms identifies shortcomings in common comparison metrics, and suggests possible optimisations for the algorithms.

Special Difficulties

None.

[1]This word count was computed by `detex of209dissertation.tex | tr -cd '0-9-Za-z \n' | 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

Date

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

Much thanks is owed to my supervisor Dr. Richard Gibbens, for his patience and invaluable advice.

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 a shortest path through this map between the *start* and the *goal*.

However, pathfinding 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 (such as a robot or computer game character) can travel directly between any two nodes connected by an edge. Figure 1.1 (d) shows the shortest path through the graph — however, this is clearly longer than the optimum path through the map, shown in Figure 1.1 (b).

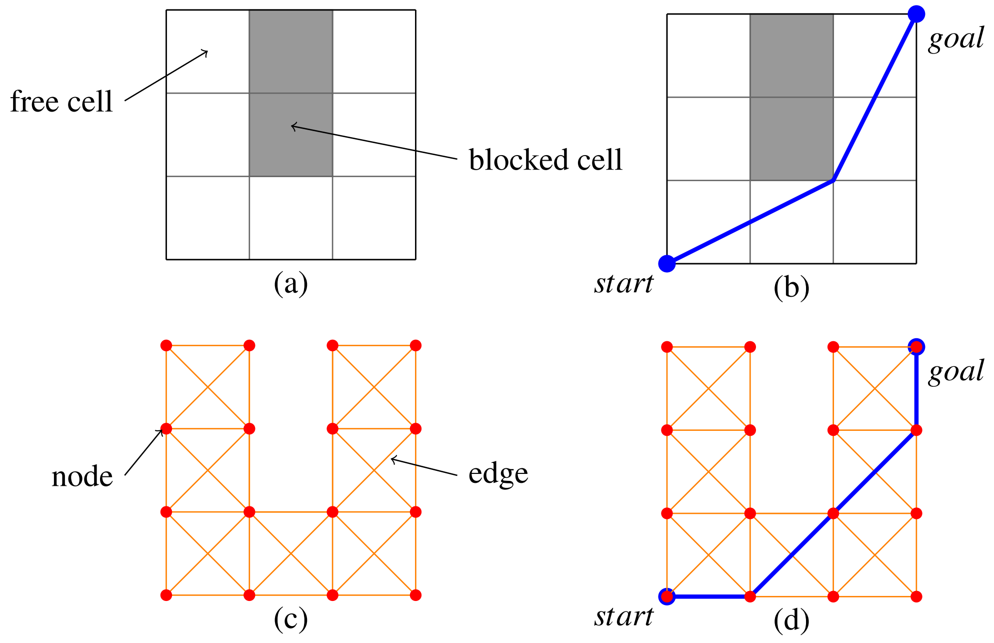


Figure 1.1: Shortest path through a map vs. shortest path through the graph representing the map

The structure of this graph is based on the grid structure of the map. If a different graph representation is chosen (such as a graph with a finer mesh, or a 'visibility graph' as described in the next chapter) then a shorter path could be achieved, but in general such a graph would require many more nodes and edges which would cause vastly 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 a space-efficient grid-based graph like that shown in Figure 1.1(c).

1.2 Related work

*A** is a well known algorithm that finds optimal paths through graphs. Applying a post-processing step to smooth and hence shorten paths returned by *A** is a technique that has been used since the earliest video games [?], but most of the research into more advanced any-angle pathfinding algorithms has taken place in the last half decade.

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

Studies such as Nash and Koenig's article in *Artificial Intelligence Magazine* [?] compare a selection of these any-angle pathfinding algorithms. However, no paper has attempted to explain and compare these algorithms within a consistent framework, and the only established authority that utilises empirical data on *Block A** is Yap et al.'s own work [?, ?], which compares only *A**, *Theta** and *Block A**.

1.3 Project goals

The goals of this project are to:

- introduce a common framework for explaining the most prominent any-angle path-finding algorithms;
- create a simulation environment that enables intuitive and informative comparison of the performance of these algorithms;
- use statistical analysis and explanations based on the framework to present conclusions on the performance of the algorithms that enhance those presented in the current literature.

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. A formal mathematical framework is defined throughout this section so that the any-angle pathfinding algorithms can be explained using a unifying graph-theoretic approach. Such an approach has not been attempted in any of the published papers on any-angle pathfinding algorithms.

2.1.1 Map

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

The map is logically divided into a grid of $N \times N$ cells of size 1×1, where cell $C_{i,j}$ includes all locations $(x,y) \in M$ where $i \leq x \leq i+1$ and $j \leq y \leq j+1$, and each cell is either 'free' or 'blocked'. An example of such a map is shown in Figure 2.1.

Valid location

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

Line of sight

A line of sight exists between locations (x_0,y_0) and (x_1,y_1) on a map if all locations that lie on the straight line drawn between them are valid — that is, 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. The existence of a line of sight between two

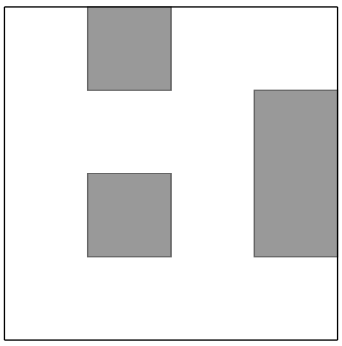


Figure 2.1: A map M of size 4×4

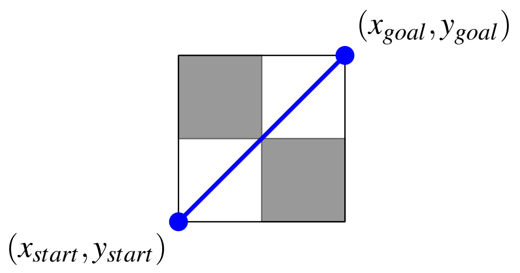


Figure 2.2: A valid path for an agent modelled as a point

locations implies that an agent can travel in a straight line between the two locations.

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 $(x, y) \in M$ where $(x_0, y_0) = (x_{start}, y_{start})$ is the *start* location of the path, and $(x_n, y_n) = (x_{goal}, y_{goal})$ is the *goal* location of the path. A path is valid if a line of sight exists between every consecutive pair of coordinates in the path: (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.

2.1.2 Graph

A graph $G(M) = (V, E)$ is a representation, or abstraction, of a map M. Each node $n \in V$ represents a valid location $(a, b) \in M$. If a line of sight exists between two locations in M then the two nodes n and n' that represent these locations are connected by an edge $e = (n, n') \in E$, and are thus called 'neighbours'. $(n, n').weight$ is the distance an agent would traverse by travelling directly between the two locations i.e. the Euclidean distance between them.

Path through a graph

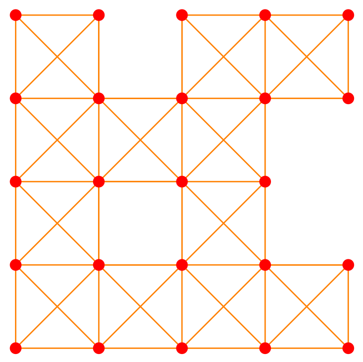
A path $P_{G(M)} = (n_0, n_1, \ldots, n_n)$ through graph $G(M) = (V, E)$ is a list of nodes $n \in V$, where $n_0 = n_{start}$ and $n_n = n_{goal}$. A path is valid if, for each pair of nodes n_i and n_{i+1}, there exists an edge $(n_i, n_{i+1}) \in E$.

There are two types of graph (see Figure 2.3):

(a) **Grid-based graph** $G_g(M)$ — a node represents every valid location that lies on a corner of a cell in M. Each node is connected to every node directly surrounding it (maximum eight) if there is a line of sight between those two nodes.[1]

(b) **Visibility graph** $G_v(M)$ — the optimal path through a visibility graph is guaranteed to correspond to the optimal path through the map that it represents, whereas the optimal path through a grid-

[1]If node n represents location (i, j), the "up to eight nodes that directly surround it" are the nodes that represent the locations $(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j+1)$, $(i+1, j+1)$, $(i+1, j)$, $(i+1, j-1)$ and $(i-1, j)$.


(a) Grid-based graph $G_g(M)$

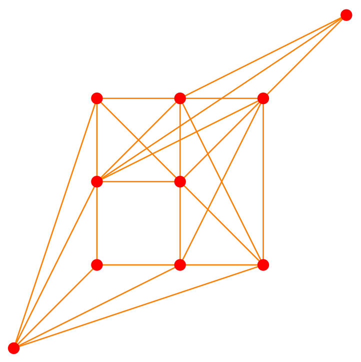

(b) Visibility graph $G_v(M)$

Figure 2.3: Graph representations of map M from Figure 2.1, where $n_{start} = (0,0)$ and $n_{goal} = (4,4)$

based graph may not [?].[2] To achieve this property, a visibility graph has a node to represent the *start* and *end* locations of the desired path, and a node to represent every location that could conceivably lie on a shortest path.[3] Each node is connected to any node to which it has a line of sight, as implied by the name ‘visibility graph’.

Lattice

The concept of a lattice is introduced because lattices are used in the upcoming explanation of ‘Discretisation’, which conceptually explains how a graph is created to represent a specific map. A lattice $L(N)$ is a graph of $N \times N$ nodes where $N \in \mathbb{Z}^+$, and can be thought of as a graph that represents a map with no blocked cells. There are two types of lattice (see Figure 2.4):

(a) **Octile lattice** $L_O(N)$ — the nodes of the lattice are arranged in a square grid, and each node is connected by an edge to the closest node, if any exists, at a bearing of any integer multiple of $\frac{\pi}{4}$ radians.

(b) **Full lattice** $L_F(N)$ — the nodes of the lattice are arranged in a square grid, and each node is connected by an edge to every other node in the lattice.

Discretisation

Discretisation is the process of creating a graph $G(M) = (V,E)$ that represents a map M. The process can be visualised as refining a lattice by removing edges and nodes until the desired graph remains. 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 on the map between the locations represented by n and n'. There are two forms of discretisation:

(a) $d_G(L_O,M) \rightarrow G_g(M)$ — an octile lattice is refined to produce a grid-based graph G_g;

(b) $d_V(L_O,M) \rightarrow G_v(M)$ — a full lattice is refined to produce a a visibility graph G_v.

A full conceptual explanation of discretisation is provided in Appendix A.1, and the implementation of discretisation is detailed in sections 3.2.2 and 3.2.3.

[2]Despite the optimality of visibility graphs, grid-based graphs are generally accepted as the preferable form of map representation for pathfinding, since for a map M of size $N \times N$, a grid-based graph has $O(N^2)$ edges, whereas a visibility graph has $O(N^4)$ edges. Therefore for large maps, grid-based graphs are a far more space-efficient representation than visibility graphs. For this reason, this investigation will focus predominantly on pathfinding algorithms applied to grid-based graphs.

[3]It can be shown [?] that any location that does not lie on the corner of a cell where exactly three of the four surrounding cells are blocked cannot conceivably lie on a shortest path.

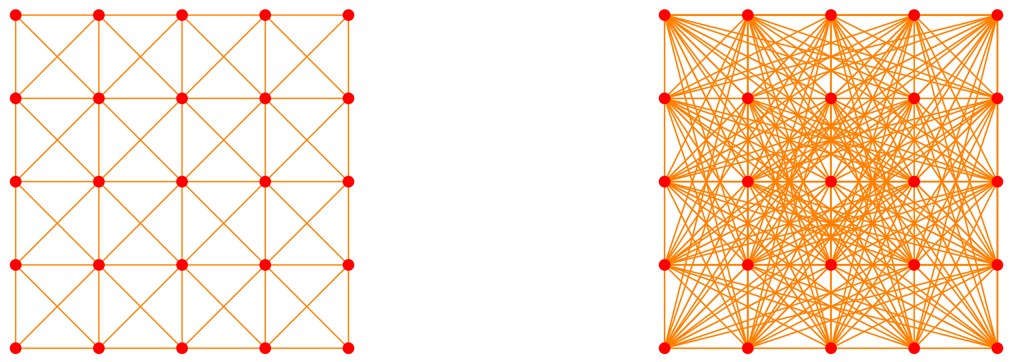

(a) Octile lattice $L_O(N)$ (b) Full lattice $L_F(N)$

Figure 2.4: Octile lattice and full lattice of size 4×4

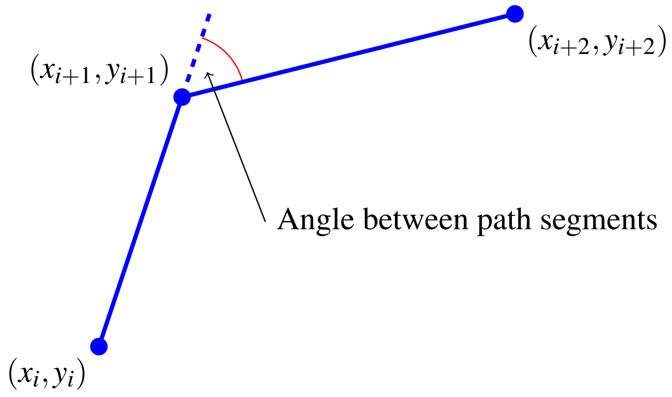


Figure 2.5: Angle between a pair of path segments

2.1.3 The any-angle pathfinding problem

The problem is to compute optimal or near-optimal paths, if they exist, between a given *start* and *goal* location in a map, by using a grid-based graph.

A path P_M through map M is optimal, denoted as P_M^*, if there do not exist any paths through M from *start* to *goal* with a shorter path length and a smaller path angle-sum, where:

Path length
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 sum of the angles 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 (see Figure 2.5), calculated as:

$$\sum_{i=0,n-2} \arccos\left(\left(\begin{pmatrix} x_{i+2} \\ y_{i+2} \end{pmatrix} - \begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix}\right) \cdot \left(\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} - \begin{pmatrix} x_i \\ y_i \end{pmatrix}\right)\right) \tag{2.1}$$

2.1.4 Solving the any-angle pathfinding problem

The steps required to find an optimal or near-optimal path P_M through a map M from *start* to *goal*, can now be stated:

1. a grid-based graph $G_g(M)$ that represents the map M is produced, where each node in the graph represents a location in the map. This process is achieved by discretisation using an octile lattice, as explained in section 2.1.2;

2. a pathfinding algorithm is applied to the $G_g(M)$, which returns a path $P_{G_g(M)} = (n_{start}, n_1, ..., n_{goal})$.[4] See section 2.2;

3. $P_{G_g(M)}$ corresponds to the path $P_M = (x_{start}, x_1, ..., x_{goal})$, using the correspondence that x_i is the location in map M that is represented by the node n_i of graph $G_g(M)$.

2.2 Any-angle pathfinding algorithms

This section starts with an introduction of pathfinding over graphs, followed by definitions of the two types of pathfinding algorithms: 'classic' and 'any-angle'. Classic pathfinding algorithms are presented first as they explain some of the important concepts required to understand the more complicated any-angle algorithms the constitute the core of this dissertation, and to serve as a useful benchmark for comparison in the Evaluation chapter. The remainder of the section describes the pathfinding algorithms in more detail.

2.2.1 Pathfinding over graphs

When applied to a graph $G(M)$, a pathfinding algorithm returns a path $P_{G(M)}$ between a given n_{start} and n_{goal}.

Section 2.1.2 introduced the concept of a graph $G(M)$ that is a representation of a map M such that each node n in $G(M)$ has an associated coordinate *coord*. In addition, all of the algorithms require graph nodes to have the following two parameters:

- *g-value* — the path length of the shortest path between n_{start} and n that the algorithm has found thus far;

- *parent* — a pointer to the previous node in the shortest path between n_{start} and n that the algorithm has found thus far. Therefore, the shortest path between n_{start} and n is found by recursively following the *parent* pointers from n to n_{start}.

At the start of the algorithm $n.g = \infty$ and $n.parent = \perp$ for all nodes (since the algorithm hasn't found any paths between any nodes at this point).[5] On termination, if a path exists then $n_{goal}.g$ is the path length of the shortest path found by the algorithm from n_{start} to n_{goal}, and this path is found by recursively following the *parent* pointers from n_{goal} to n_{start};[6] if a path doesn't exist (so that n_{start} and n_{goal} fall into distinct connected components) then $n_{goal} = \perp$.

2.2.2 Types of pathfinding algorithms

A summary of our findings so far motivates the need for a more advanced type of pathfinding algorithm than the classic pathfinding algorithms. As seen in the Introduction chapter, even when a path through a grid-based graph $G_g(M)$ is optimal, the corresponding path through the map M may not be. Since classic pathfinding algorithms find optimal paths through graphs, and because the decision has been made to use grid-based graphs, it is clear that using classic pathfinding algorithms may give sub-optimal

[4] *Block A** operates in a different way, and is dealt with separately.

[5] Where $\perp$ denotes that the parameter value is undefined.

[6] The pathfinding algorithm guarantees $n_{start}.parent = \perp$.

paths through maps.

Any-angle pathfinding algorithms aim to improve upon classic pathfinding algorithms by taking a grid-based graph $G_g(M)$ and selectively adding a small number of edges to $G_g(M)$ (i.e. ‘augmenting’ $G_g(M)$) so that it closely resembles a visibility graph in the region that surrounds the path. In this way, any-angle pathfinding algorithms aim to benefit from the efficiency of grid-based graphs and the optimality of visibility graphs, though tradeoffs are required on both sides. These explanations are now formalised:

Classic pathfinding algorithm
A classic pathfinding algorithm will return the optimal[7] path $P^*_{G(M)}$ through $G(M)$.[8]

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

2.2.3 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* (see Algorithm 1).

Overview
Starting at n_{start}, *Dijkstra* selects and then processes (or ‘expands’) one node at a time (see ‘Expansion’ paragraph below). It is a provable [**?**] invariant of the algorithm that when a node n is selected to be processed, $n.g$ is equal to the length of the shortest path in the graph to n from n_{start}. For this reason, once a node has been expanded, it need not be expanded again (this would incur unnecessary work).[9] When n_{goal} is expanded, $n_{goal}.g$ is the length of the shortest path in G from n_{start} to n_{goal} (according to the invariant condition), so the algorithm terminates.

Expansion
Dijkstra selects the next node n to expand from *openSet* which is a priority queue[10] that stores nodes in increasing order of their *g-values*. To expand n: 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 it had found so far (as defined by the *parent* pointers from n to n_{start}) is longer than the path to n_{neigh} that is made up of the shortest path found so far to n and then from n to n_{neigh},[11] which can be stated as the condition:

$$n.g + (n, n_{neigh}).weight < n_{neigh}.g \tag{2.2}$$

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

- updating the *parent* of n_{neigh} to n;
- updating the *g-value* of n_{neigh} to $n.g + (n, n_{neigh}).weight$.

[7]The definition of an optimal path $P^*_{G(M)}$ through a graph is analogous to the definition of an optimal path P_M in subsection 2.1.3.

[8]Recall that $P_{G(M)}$ corresponds to P_M, but the optimal path $P^*_{G(M)}$ through $G(M)$ does not necessarily correspond to the optimal path P^*_M through M — this idea was illustrated in the Introduction chapter.

[9]A set *closedSet* ensures that this does not occur.

[10]The set and priority queue methods $add()$, $pop()$ and $contains()$ are defined in the usual way.

[11]This condition relies on the provable [**?**] fact that any sub-path of a shortest path is itself a shortest path.

Finally, n_{neigh} is added to *openSet* if it is not already in it, and then the next node to be expanded is selected.

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($G$, $n_{start}$, $n_{goal}$)
    $openSet \leftarrow \perp$
    $closedSet \leftarrow \perp$
    $n_{start}.g \leftarrow 0$
    $openSet.add(n_{start})$
    while $openSet \neq \perp$ 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 $\perp$
def Update($n_{neigh}$)
    if $n_{curr}.g + (n_{curr}, n_{neigh}).weight < n_{neigh}.g$ then
        $n_{neigh}.g = n_{curr}.g + (n_{curr}, n_{neigh}).weight$
        $n_{neigh}.parent = n_{curr}$
        return $true$
    else
        return $false$
```

2.2.4 A*

$A*$ is based on *Dijkstra's shortest-paths* algorithm and also finds the optimal path P^*_G through a graph G, but uses a heuristic h to reduce the number of node expansions required [**?**].

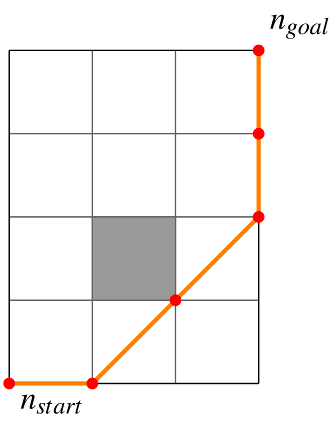
In addition to a *g-value* and a *parent*, each node also has a:

- *h-value* — Euclidean distance between n and n_{goal}: a cheaply computable monotonic estimatemof the actual shortest path length between n and n_{goal}.[12]

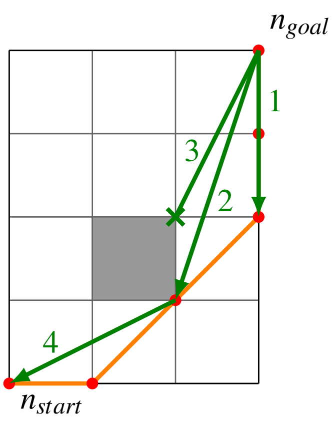
While *Dijkstra* preferentially expands nodes with low *g-value*s (by utilising a priority queue called *openSet*, which sorts using *g-values*), $A*$ preferentially expands nodes with low *f-score*s (by utilising a priority queue called *openSet*, which sorts using *f-values*), where a node n's *f-score* is the algorithm's current estimate of the shortest path from n_{start} via n to n_{goal} — that is to say:

$$n.f = n.g + n.h \tag{2.3}$$

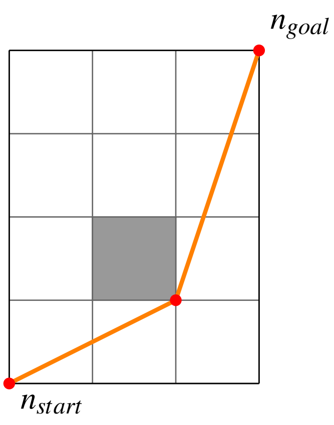
[12]The monotonicity of Euclidean distance as a heuristic ensures that $A*$, like *Dijkstra*, is complete (if a path exists, it finds it) and optimal (if a path is found, it is a shortest distance path).



(a) Original path



(b) Line of sight tests



(c) Improved path

Figure 2.6: *A* with post-smoothing*

The pseudo-code for $A*$ differs only from *Dijkstra* in the `Update` subroutine, where the *h-score* must also be set (see Algorithm 2).

Algorithm 2: `Update` from A*

```
def Update($n_{neigh}$)
1   if $n_{curr}.g + (n_{curr}, n_{neigh}).weight < n_{neigh}.g$ then
2       $n_{neigh}.g \leftarrow n_{curr}.g + euclidean(n_{curr}, n_{neigh})$
3       $n_{neigh}.h \leftarrow euclidean(n_{neigh}, n_{goal})$
4       $n_{neigh}.parent = n_{curr}$
5       return $true$
6   else
7       return $false$
```

2.2.5 A* with post-smoothing

A with post-smoothing* is the first any-angle pathfinding algorithm to be introduced in this project. It applies a post-processing step to the path $(n_{start}, n_1, \ldots, n_{goal})$ returned by $A*$, which 'smoothes' and therefore shortens this path (see Algorithm 3).

The smoothing is achieved by changing the *parent* pointer for certain nodes in the original path — for example, the *parent* of a node n_j in the original path is defined as n_{j-1}, but *A* with post-smoothing* may change $n_j.parent$ so that it points to a different node in the original path (call it n_i) that is closer to the *start* of the path than n_{j-1} (i.e. $i < j-1$) — this re-parenting is only allowed if there is a line of sight between n_j and n_i (see Figure 2.6). If $G(M)$ did not have an edge between n_j and n_i, then this re-parenting procedure has the effect of augmenting $G(M)$ by adding an edge to it.

2.2.6 Theta*

Subsection 2.2.5 demonstrated how *A* with post-smoothing* performs smoothing on the path returned by the classic pathfinding algorithm $A*$. In contrast, *Theta** smoothes as it goes along by progressing in a similar way to $A*$, but also attempting to re-parent each of n_{curr}'s neighbours n_{neigh} with $n_{curr}.parent$

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 $n_{next} = \perp$ 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
        $n_{next} \leftarrow n_{next}.parent.parent$
```

at the time when n_{curr} is expanded.[13]

The pseudo-code for *Theta** differs only from *A** in the `Update` subroutine (see Algorithm 4).

Algorithm 4: `Update` from THETA*

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

2.2.7 Lazy Theta*

*Lazy Theta** attempts to refine *Theta** by finding similar paths despite performing fewer line of sight tests.

[13]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.

While *Theta** performs a line of sight test for every neighbour n_{neigh} of every node n_{curr} that is expanded, *Lazy Theta** only performs line of sight tests for every node n_{curr} that is expanded — by initially assuming that all line of sight tests pass, and only actually doing a test between a node and its parent when the node is expanded. Whenever a line of sight test fails, a costly cleanup step is required to undo the effect of an incorrect assumption (see Algorithm 5).

The paths returned by *Lazy Theta** are not always the same as those returned by *Theta** since the edge relaxation occurs at a different point in the iteration.

Algorithm 5: LAZY THETA*

```
def LazyTheta*($G$, $n_{start}$, $n_{goal}$)
    $openSet \leftarrow \perp$
    $closedSet \leftarrow \perp$
    $n_{start}.g \leftarrow 0$
    $openSet.add(n_{start})$
    while $openSet \neq \perp$ do
        $n_{curr} \leftarrow openSet.pop()$
        Initialise($n_{curr}$)
        $closedSet.add(n_{curr})$
        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 $\perp$
def Initialise($n_{curr}$)
    if LineOfSight($n_{curr}, n_{curr}.parent$) $= false$ then
        // cleanup code if line of sight doesn't pass
        $newParent \leftarrow \underset{n' \in expandedNeigh(n_{curr})}{\text{argmin}} (n'.g + (n', n_{curr}).weight)$
        $n_{curr}.parent \leftarrow n'$
        $n_{curr}.g \leftarrow n'.g + (n', n_{curr}).weight$
def Update($n_{neigh}$)
    // assume line of sight test passes
    if $n_{curr}.parent.g + (n_{curr}.parent, n_{neigh}).weight < n_{neigh}.g$ then
        $n_{neigh}.g \leftarrow n_{neigh}.parent.g + (n_{curr}.parent, n_{neigh}).weight$
        $n_{neigh}.f \leftarrow euclidean(n_{neigh}, n_{goal})$
        $n_{neigh}.parent \leftarrow n_{curr}.parent$
        return $true$
    else
        return $false$
```

2.2.8 Block A*

*Block A** is the most complex algorithm in this dissertation. It was published in 2011, and is at the cutting edge of any-angle path-finding algorithmic research.

Overview
*Block A** aims to achieve faster computation of paths by doing far fewer expansions than the algorithms so far considered, because *Block A** expands 'blocks' rather than just nodes, where a block is a graph-like structure that represents an entire sub-area of the overall map M — hence each expansion in *Block A** makes far more progress than an expansion in the other algorithms. However, the expansion of a block is more complicated than that of a node as it requires the values of the shortest paths between points that lie on different sides of a block — though this extra complexity is partly assuaged by the fact that these paths do not need to be explicitly calculated since they are available from a precalculated *Local Distance Database (LDDB).*

The remainder of this subsection contains a novel explanation of *Block A** which draws close parallels with the graph-based explanations provided in the previous subsections. Due to the complexity of *Block A**, this explanation is mathematically and notationally intensive, yet considerably more succinct that the example-based explanation found in Yap et al.'s original paper [**?**].

Blocks
A block $B_{i,j}$ of size ${N_B}^2$ is a graph-like structure that represents the area of the map M covered by all cells $C_{k,l} \in M$ where $i \le k \le i + N_B$ and $j \le l \le j + N_B$.

For each block $B_{i,j} = (V_{i,j}, E_{i,j})$, node $n \in V_{i,j}$ represents a location $(a,b) \in M$ where (a,b) lies on the boundary of $B_{i,j}$ — that is to say, where $a = i$ or $a = i + N_B$, and $b = j$ or $b = j + N_B$. n has a *g-value*, *h-value* and *parent* defined as per *A**.[14]

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 edges 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' (see Figure 2.8).[15] The edge weights for regular blocks are discovered by querying the *Local Distance Database (LDDB)*, but the weights of edges in B_{start} and B_{goal} are calculated separately — this is discussed in the Implementation chapter.

Each block $B_{i,j}$ contains nodes that are collocational with nodes from up to eight neighbouring blocks. Figure 2.7 shows collocational nodes as two or four partially overlapping red circles. *Block A** maintains the invariant for two collocational nodes n and n' that $n.g = n'.g$ and $n.h = n'.h$ — the enforcement of this invariant enables shortest path information to flow between adjacent blocks.

For the purposes of the *Block A** algorithm, each *block* $B_{i,j}$ maintains an unordered $openSet_{i,j} \subseteq V_{i,j}$, and a $heapValue_{i,j}$, where $openSet_{i,j}$ contains all the nodes in $B_{i,j}$ that have been updated since the block was last expanded (note that a block can be expanded more than once[16]), and $heapValue_{i,j}$

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

[15]This important distinction warrants 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 the shortest path between the two nodes (which may not be a straight line) is the edge weight which may range from 0 to ∞.

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

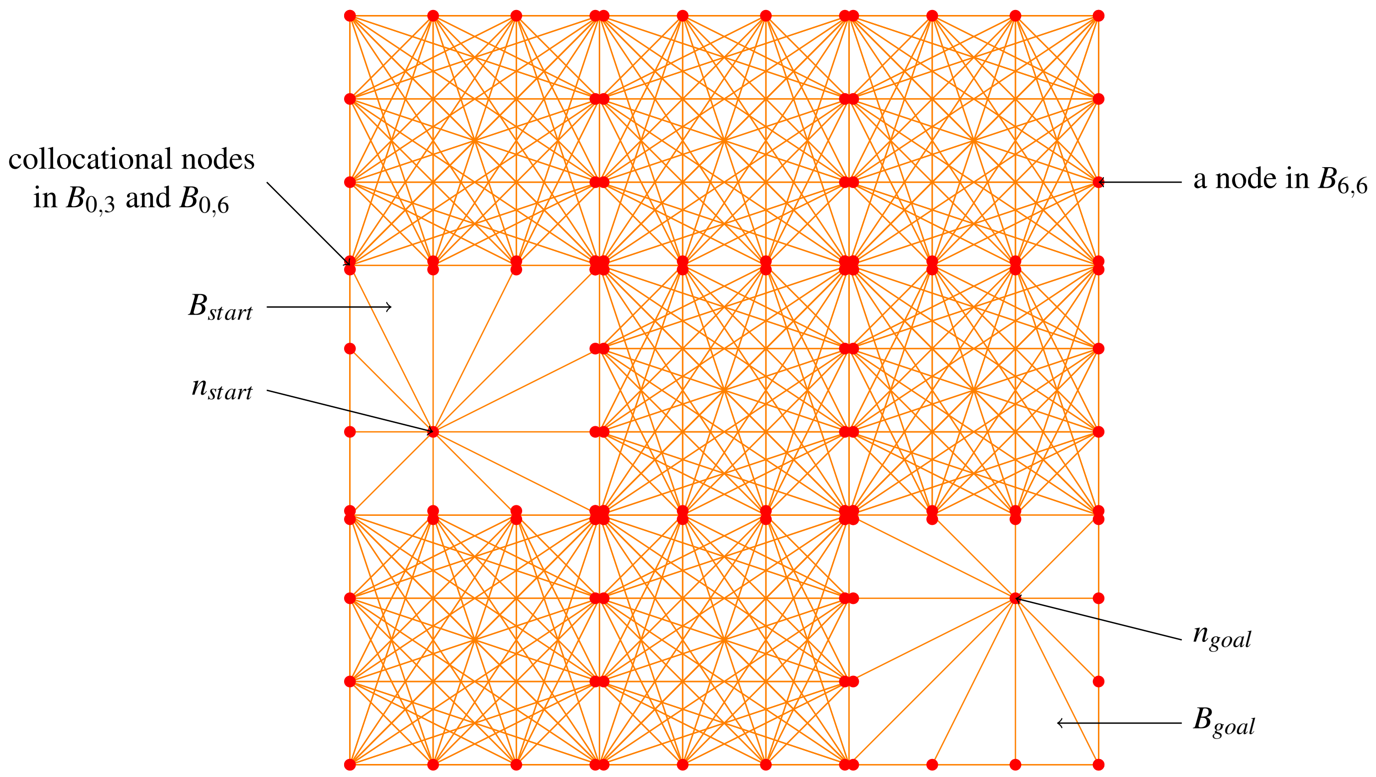


Figure 2.7: Logical view of a map of size 9×9 represented as a set of 9 blocks of size 3×3

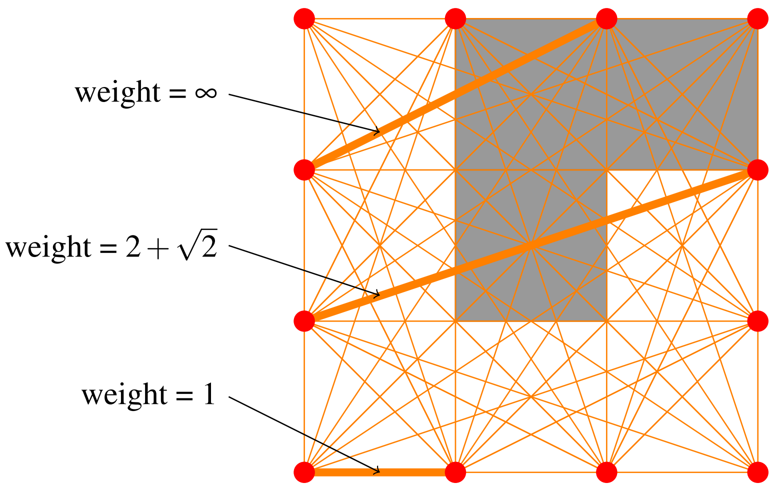


Figure 2.8: A block of size 3×3 laid over the area of map it represents, with certain highlighted edge weights labeled.

is the smallest *f-score* of the nodes in $openSet_{i,j}$,[17] and is used to order blocks in the priority queue $openSet_{blocks}$.

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)–(c,d) for all map configurations M of size $N \times N$, where $a,b,c,d \in \mathbb{Z}^+$ and (a,b)–(c,d) lies on the boundary of M.

Algorithm walkthrough (see Algorithm 6)

Initialisation — for all $n \in B_{start}$, $n.g$ is set to $(n_{start}, n).weight$, and B_{start} is added to $openSet$. For all $n \in B_{goal}$, $n.h$ is set to $(n, n_{goal}).weight$.[18]

Iteration — the iteration stage of $Block\ A*$ proceeds similarly to $A*$, though $Block\ A*$ expands blocks chosen from $openSet_{blocks}$ whereas $A*$ expands nodes. Since a block can be expanded multiple times, the shortest path may not have been reached when B_{goal} is first expanded, hence the expansion of B_{goal} cannot be used as a termination condition. Therefore a $length$ variable 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* above) 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* (ensured by the $length$ variable) then there are no possible shorter paths to B_{goal}, so the algorithm terminates.

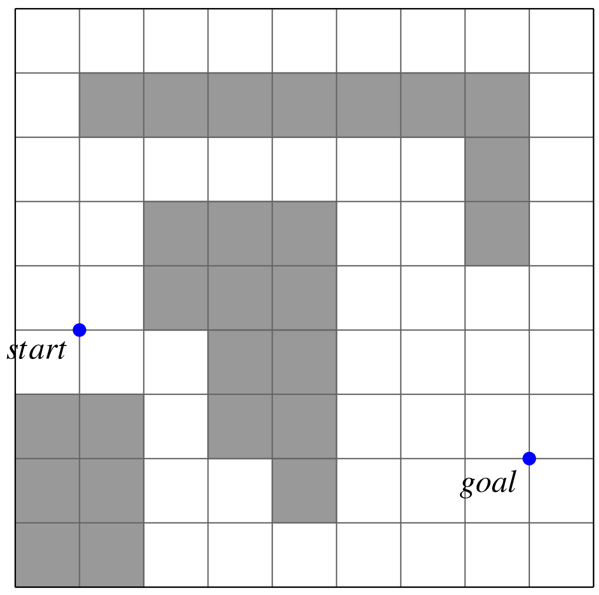
Traceback— on termination, the boundary nodes of the blocks through which $P_{G(M)}$ passes are found by recursively following the *parent* pointers from n_{goal} to n_{start} (see Figure 2.9(b)). However, to recover the nodes of $P_{G(M)}$ that do not lie on the boundaries of blocks, a traceback stage is required which involves multiple queries to the *LDDB* (see Figure 2.9 (c)). Although the details of this are not presented in Yap et al.'s paper, an explanation is presented in the Implementation chapter.

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

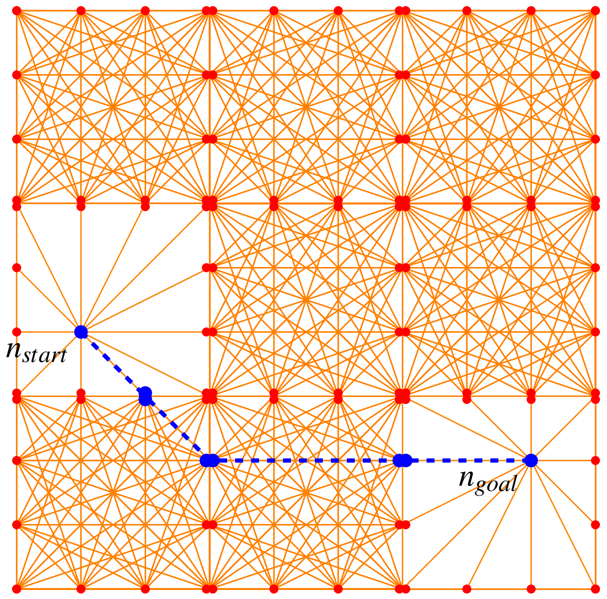
[18] The exceptional case that $B_{start} = B_{goal}$ must also be checked for. This is discussed in the Implementation chapter.

Algorithm 6: BLOCK A*

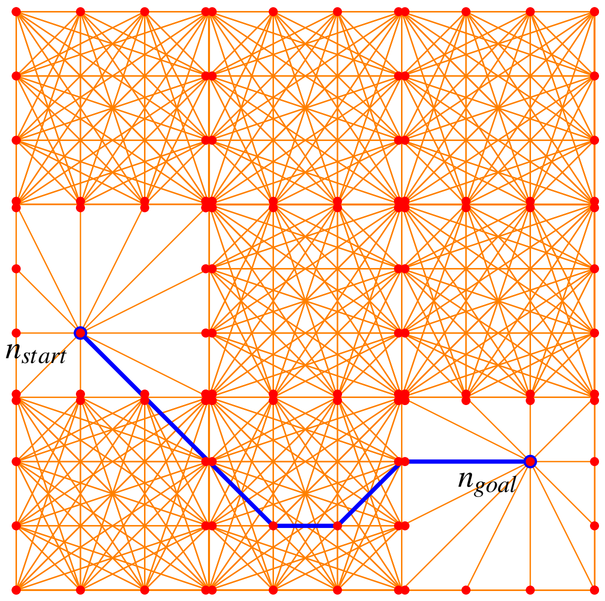
```
def BlockAStar($G$, $n_{start}$, $n_{goal}$)
    $B_{start} \leftarrow$ Init($n_{start}$)
    $B_{goal} \leftarrow$ Init($n_{goal}$)
    $length \leftarrow \infty$
    $openSet_{blocks}.add(B_{start})$
    while $(openSet_{blocks} \neq \perp) \wedge ((openSet_{blocks}.peek()).heapValue < length)$ do
        $B_{curr} \leftarrow openSet_{blocks}.pop()$
        $openSet_{curr} \leftarrow B_{curr}.openSet$
        if $B_{curr} = B_{goal}$ then
            $length \leftarrow \min_{n \in openSet_{curr}} (n.g + euclidean(n, n_{goal}), length)$
        Expand($B_{curr}, openSet_{curr}$)
    if $length \neq \infty$ then
        TraceBack($n_{goal}$)
    else
        return $\perp$
def Expand($B_{curr}, openSet_{curr}$)
    while $openSet_{curr} \neq \perp$ do
        $n_{ingress} \leftarrow openSet_{curr}.pop()$
        foreach $n_{egress} \in B_{curr}$ do
            if $n_{ingress}.g + (n_{ingress}, n_{egress}).weight < n_{egress}.g$ then
                $n_{egress}.g \leftarrow n_{ingress}.g + (n_{ingress}, n_{egress}).weight$
                foreach $n' \in n_{egress}.collocational$ do
                    $n' \leftarrow n_{ingress}.g + (n_{ingress}, n_{egress}).weight$
                    $openSet_{n'.coord}.add(n')$
                    if $openSet_{blocks}.contains(B_{n'.coord}) = false$ then
                        $openSet_{blocks}.add(B_{n'.coord})$
```



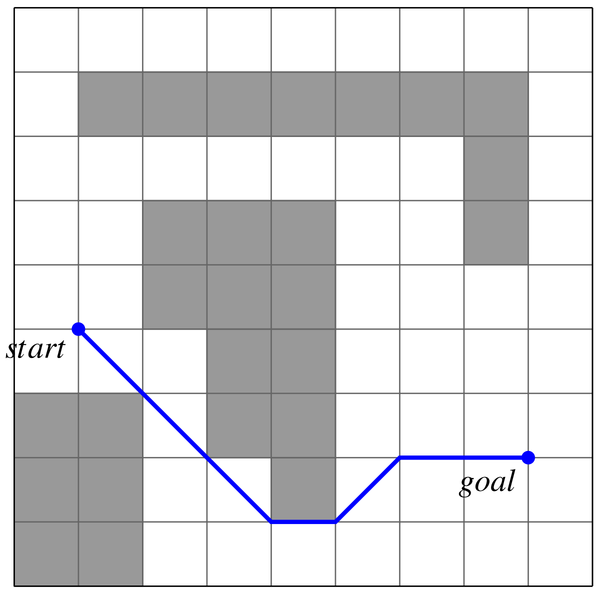
(a) Original map *M*



(b) Boundary nodes of path identified



(c) Traceback stage complete



(d) Path P_M through map *M*

Figure 2.9: Solution of *Block A**

2.3 Requirements analysis

As specified in the Project Proposal (see Appendix B), 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

		Priority			
ID	Functional requirements	M	S	C	W
1	The system shall load one of a collection of maps from the generator	🟩			
2	The system shall load one of a collection of maps from a save file		🟨		
3	The system shall create a grid-graph from a given map	🟩			
4	The system shall create a visibility graph from a given map			🟧	
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	🟩			
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	🟩			
7	The system shall display the numeric statistics for each path for the current map	🟩			
	Non-functional requirements				
1	The system shall be designed in a modular way to allow easy extension for new components		🟨		

2.3.2 Map generation

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

2.3.3 Algorithm implementation

		Priority			
ID	Functional requirements	M	C	S	W
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	🟩			
2	The system shall allow arbitrary start and end coordinates for any map		🟨		
	Non-functional requirements				
1	The system shall be designed in a modular way to allow easy extension for new algorithms		🟨		

2.3.4 Data gathering

		Priority			
ID	Functional requirements	M	C	S	W
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	🟩			
	Non-functional requirements				
1	The system shall be designed with a clear API that enables quick and easy data gathering	🟩			

2.4 Design model

The plan for the implementation phase was based on that presented in the Project Proposal (see Appendix B).

An ‘Incremental build model’ was used, with new modules being developed and tested separately before being integrated into the work program. This model was chosen as debugging is relatively fast as changes between iterations are relatively small, and it allows frequent reviews of the software system — this was particularly useful when implementing different experimental versions of *Block A**’s *LDDB*.

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.

2.5 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. Various libraries were used, including:
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. Various packages were used, including:
Tikz — enables programmable diagram creation;
algorithm2e — enables integration with the parent document;
listings — displays colour formatted code for a range of programming languages;
todonotes — enables clear organisation of tasks that haven't been completed.

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.

3 | Implementation

This chapter provides an overview of the implementation of the project and explores certain performance-critical features in further detail.

Figure 3.1 provides an simplified view of the flow of the user interface and engine of the program, and is included to serve as a useful framework with which to describe the implementation of the program.

The basic control flow upon which Figure 3.1 is based is explained below:

- **Map & algorithm selection** — the user selects a map using one of the three methods below, selects a *start* and *goal* location and selects a pathfinding algorithm;
  - **MapGenerator** — allows the program to automatically generate a map according to user-selected parameters;
  - **MapCreator** — creates the map using a 'paint brush' style editor;
  - **MapLoader** — loads a map that has been previously saved;
- **GraphGenerator** — the system creates a graph representation of the map;
- **Pathfinding algorithm** — the system runs the selected pathfinding algorithm on the graph;
- **Visualizer** — the system displays the path, and statistics about the path, in the visualiser;
- **Data export** — the statistics about the path may be exported in CSV format.

3.1 Map selection

This section introduces how maps are implemented, and the three methods of selecting a map — automatic map generation, map creation, and loading a previously saved map.

3.1.1 Maps

A map is implemented with a `Map` object, which is a square array of `Cell` objects. Each `Cell` object has a Boolean instance variable that records whether or not that cell is blocked.

3.1.2 Map generation

The system provides a module to automatically generate maps according to a given set of parameters, which define the size $N \times N$ of the map, the percentage C of the cells of the map that are blocked, and a clustering score D which defines how likely the blocked cells are to be found in clusters as opposed to spaced evenly throughout the map (see Figure 3.2). This facility is required so that large volumes

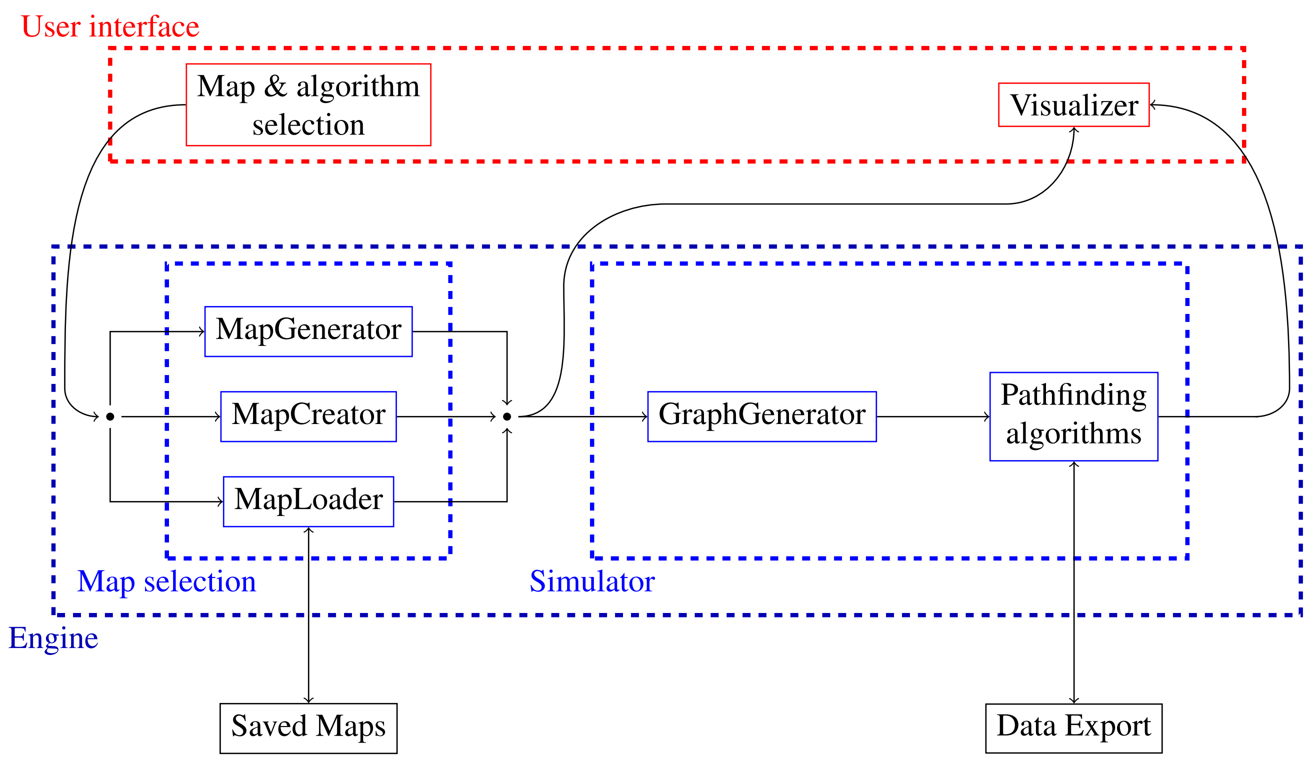


Figure 3.1: Flow of the user interface and engine

of maps can be created to allow rigorous statistical analysis of the algorithms over maps with certain known properties. The bespoke algorithm devised for this project to pseudo-randomly create such maps is now explained.

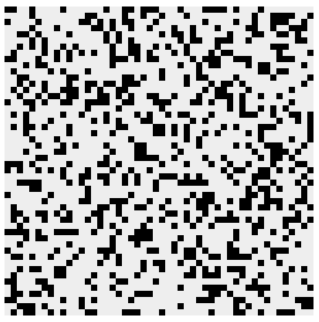
Overview
Starting with a blank map (i.e a map where every cell is free), each iteration of the algorithm chooses one unblocked cell of the map to be blocked, until $C\%$ of the cells have been blocked — i.e. until C/N^2 iterations have completed. To record which cells were blocked on previous iterations and to decide which cell to block on the next iteration, the algorithm maintains a matrix m of size $N \times N$, where at any point in the algorithm the value of element $m_{x,y}$ represents the 'potential' of cell (x,y) — where the potential of element $m_{x,y}$ represents the relative probability that (x,y) will be chosen to be blocked on the next iteration. To be more precise:

$$P((x,y)\text{chosen to be blocked on the next iteration}) = m_{x,y} / \sum_{i,j} m_{i,j} \tag{3.1}$$

Initially, the potential of every cell is 1, to represent that there is an equal chance of $1/N^2$ of any cell being chosen to be blocked in the first iteration.

If cell (x,y) is chosen to be blocked, $m_{x,y}$ is set to 0 to ensure that it cannot be chosen again, and the potentials of the eight cells that surround (x,y) are increased by a value which depends on the clustering score D, to ensure that if $D > 0$ then subsequent iterations are more likely to block cells that are spatially close to cells that were blocked on previous iterations, and hence create clusters of blocked cells. The amount by which the potentials of the cells surrounding (x,y) are increased when (x,y) is blocked models a crude approximation of a potential field, or gravity well, around (x,y) (see Figure 3.3).

Implementation
A single iteration of the algorithm, which is given in the **repeat ... until** block in the pseudo-code in


(a) $D = 0$

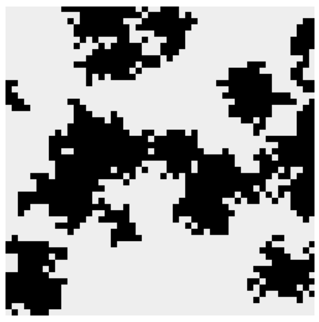
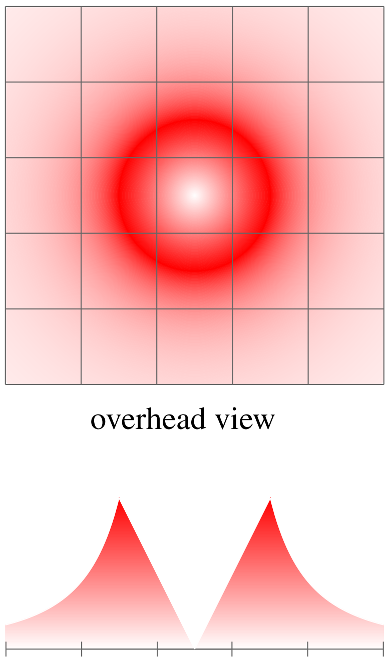
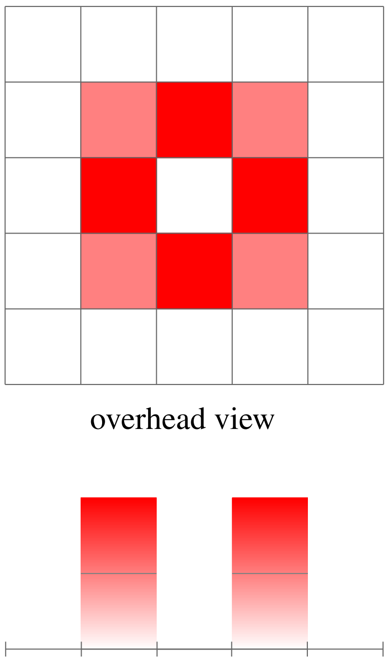

(b) $D = 50$

Figure 3.2: Close up of generated maps of coverage $C = 30\%$ and different clustering scores D



(a) Gravity well



(b) Approximation

Figure 3.3: The potential field of a gravity well and an approximation to a gravity well

Algorithm 7, can now be summarised: a random number r between 0 and $\sum_{i,j} m_{i,j}$ is selected, and m is traversed row-by-row until the cumulative sum of the potentials of the elements traversed is at least r, at which point the traversal stops and the cell that has been reached is set as blocked.

Now that (x,y) has been chosen as the cell to be blocked, the potentials of (x,y) and the surrounding eight cells are modified according to the gravity well model (i.e. $m_{x,y}$ is set to 0, the horizontal neighbours $m_{x-1,y}$, $m_{x+1,y}$ and the vertical neighbours $m_{x,y-1}$ and $m_{x,y+1}$ are increased by $2 \times D$ and the diagonal neighbours $m_{x-1,y-1}$, $m_{x+1,y-1}$, $m_{x+1,y+1}$ and $m_{x-1,y+1}$ are increased by D), and the next iteration commences (see Figure 3.4).

After C/N^2 iterations, $C\%$ of the cells of the map are blocked, and the algorithm terminates.

Algorithm 7: GENERATEMAP

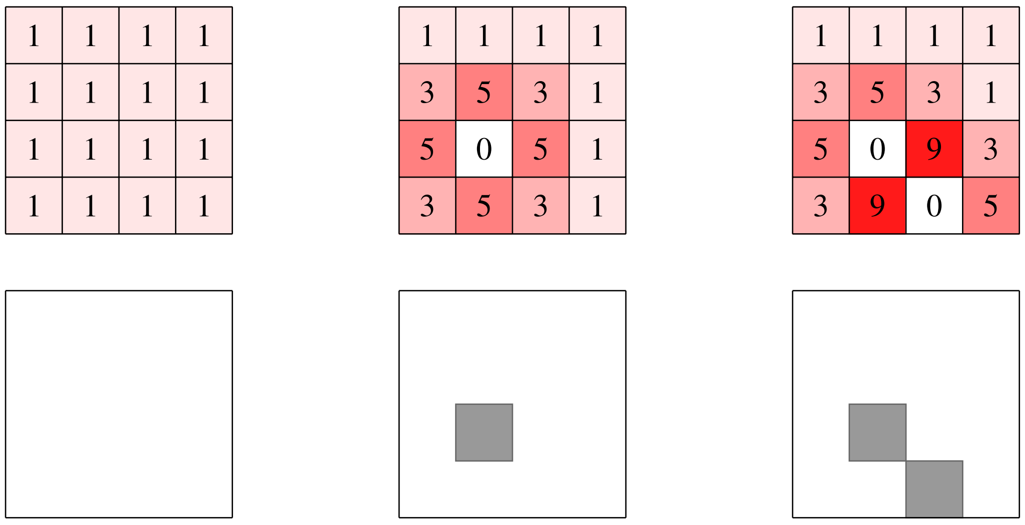
```
def GenerateMap($m, C, D$)
    repeat
        $r \leftarrow random(0, \sum_{i,j} m_{i,j})$
        $i, j \leftarrow 0$
        while $r \geq 0$ do
            $r \leftarrow r - m_{i,j}$
            if $i < R - 1$ then
                $i \leftarrow i + 1$
            else
                $i \leftarrow 0$
                $j \leftarrow j + 1$
        SetAsBlocked($i, j$);
    until $(C/N^2) times$
def SetAsBlocked($m_{i,j}$)
    $m_{i,j} \leftarrow 0$
    foreach $m_{k,l}$ in $horizontalOrVerticalNeighbour(m_{i,j})$ do
        if $m_{k,l} \neq 0$ then
            $m_{k,l} \leftarrow m_{k,l} + D$
    foreach $m_{k,l}$ in $diagonalNeighbour(m_{i,j})$ do
        if $m_{k,l} \neq 0$ then
            $m_{k,l} \leftarrow a_{k,l} + 2 \times D$
```

3.1.3 Map creation

The system provides a tool for manually creating custom maps. This tool enables the creation of maps that test specific edge cases of the algorithms (see Figure 3.5).

The Map Creation tool is accessed through the UI. The user is presented with a blank map of size 100×100, 200×200 or 400×400, and chooses a brush or eraser from a range of different sizes. The brush creates blocked cells wherever the user clicks and drags on the map, whereas the eraser creates free cells.

The map size and brush size selections are implemented with radio buttons and drop-down combination boxes from *Java*'s *Swing* library. The click and drag capability is provided by a `MouseMotionListener`,



(a) Initialisation (b) Iteration 1: r = 5.87 (c) Iteration 2: r = 9.21

Figure 3.4: First two iterations of `GenerateMap`, with *N*=4 and *D*=2. The top row shows the square matrix *m* of potentials, and the bottom row shows the map itself.

which calls a specific method any time the mouse's location changes whilst the mouse button is held.

3.1.4 Loading a map

Maps can be saved and loaded, which allows statistical analysis of the performance of the pathfinding algorithms to be performed on a fixed set of maps, and also allows the correctness of the pathfinding algorithms themselves to be thoroughly tested with a suite of edge case maps.

The *Load* and *Save* buttons in the UI are `JButtons` from the *Swing* library, which call a `JFileChooser` when clicked (via the `ActionListener` that is attached to the `JButton`), which handles the creation of a load/save dialog box, and returns the file path of the map file that the user wishes to load/save.

The map file is a platform-independent binary representation of a `Map` object, which is created when a map is saved by *serialising* the `Map` object, and can be *deserialised* to load a map. *Serialisation*, which is the mechanism for creating a binary representation of an object, occurs when an object is passed as an argument to *Java*'s `ObjectOutputStream` as long as the object's class implements the `Serializable` marker interface, which is simply a way of telling the compiler that this class can be *serialised*.

3.1.5 Start and goal point selection

The selection of the start and goal points of the path works in a similar way to the map creation tool: radio buttons allow the user to specify which of the two points the user wishes to select, and a `MouseListener` calls a method to record where on the map the user clicked.

3.2 Graph generation

This section introduces how graphs are implemented, and how both grid-based and visibility graphs are generated from maps. The generation of visibility graphs requires the explanation of the *Line of Sight* algorithm, which is also used in the implementation of some of the pathfinding algorithms.

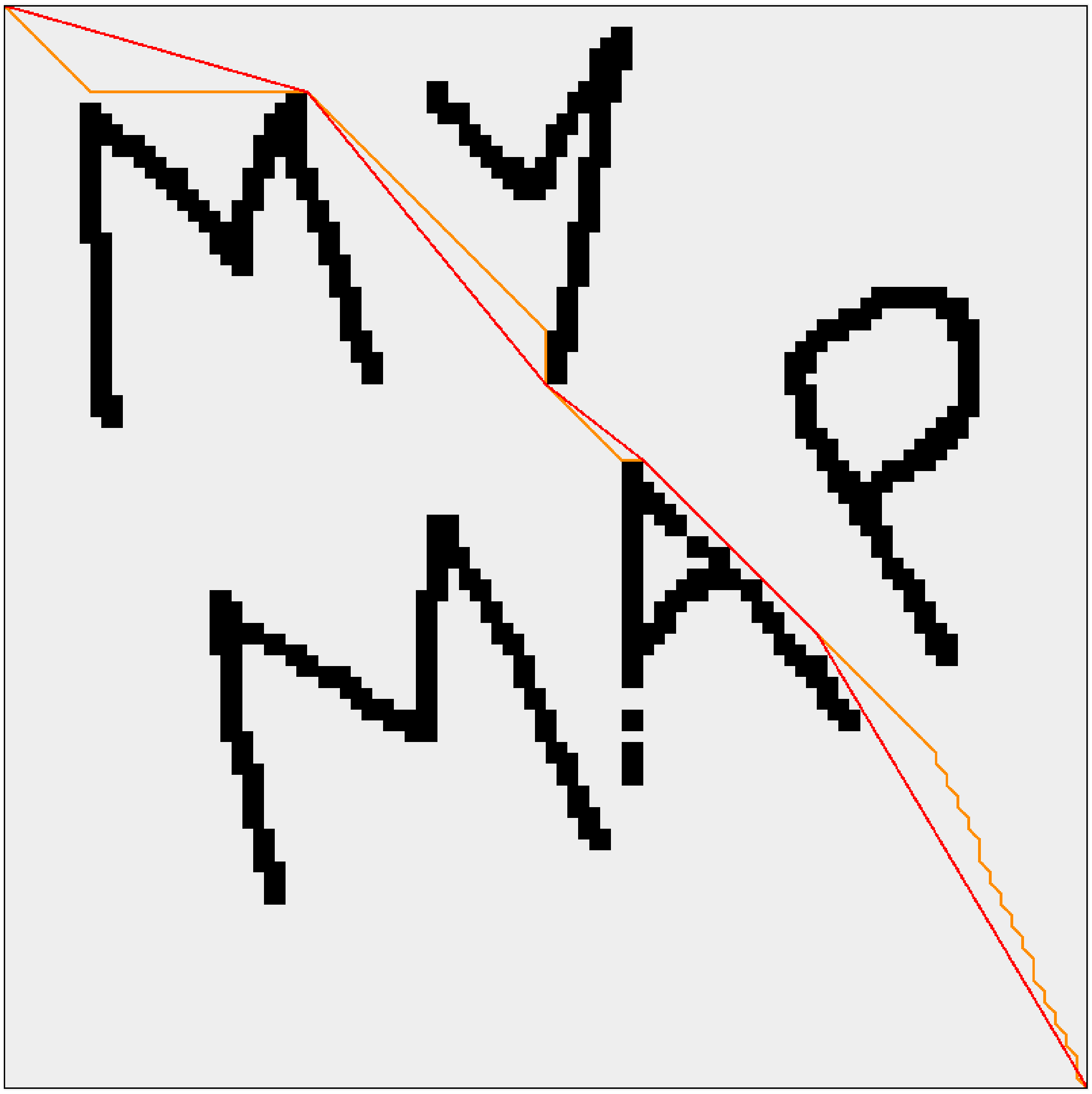


Figure 3.5: A map created with the Map Creation tool and solved by *A** (orange) and *A* with post-smoothing* (red)

3.2.1 Graphs

A graph is implemented with a `Graph` object, which is a set of `Node` objects. Each `Node` object has two floating-point instance variables to store its *g-value* and *f-value*, a `Node` instance variable which points to its parent, and a `Coordinate` object to store the location on the map that the node represents.

As explained in the Preparation chapter, every node expansion in the pathfinding algorithms sequentially iterates over the list of neighbours of that node. Therefore although edges are normally understood as a global property of a graph, the decision was made to implement edges as a `LinkedList<Node>` instance variable of each `Node` object, where the list contains the neighbours of that node. This allows fast and efficient access to the neighbour list of each node.

3.2.2 Grid-based graph generation

Section 2.1.2 presents a conceptual explanation of *discretisation*, which is the process of generating a graph from a map. In this Implementation chapter, the process of generating a graph from a map is referred to as *graph generation*, to ensure that there is no confusion between the conceptual explanation (which was designed to assist the explanation of the algorithms within a graph-theoretic structure) and the actual implementation (which was designed to be computationally efficient).

Implementation
Starting with an empty set of nodes, the algorithm performs two loops:

1. for each location on the map that lies on the corner of a cell and is a *valid location*, a new node is inserted into the set — that is to say, i is iterated over $0, 1, \ldots, N-1$ and j over $0, 1, \ldots, N-1$, and a new `Node` object with coordinate (i, j) is inserted if (i, j) is a *valid location*;[1]

2. for each node in the set, any of its eight neighbours are added to its neighbour list if:

    - *diagonal neighbour* — the cell that lies between the locations that the two nodes represent is not blocked;
    - *horizontal or vertical neighbour* — at least one of the cells that lie on either side of the location of the edge between those nodes is free.

3.2.3 Visibility graph generation

As discussed in the Preparation chapter, this project focuses on the performance of pathfinding algorithms on grid-based graphs. However, the paths produced by visibility graphs are used as a baseline for comparison in the Evaluation chapter, as they are guaranteed to be optimal.

Implementation
Visibility graph generation is similar to that of grid-based graphs. Starting with an empty set of nodes, the algorithm performs two loops:

1. for each location on the map that lies on the corner of a cell, a new node is inserted into the set if exactly three of the surrounding four cells are free, or if the location is where either the *start* or *goal* of the path was selected by the user.

2. for each node in the set, any of the other nodes in the set are added to its neighbour list if a line of sight exists between the locations that the two nodes represent.

[1]A *valid location* in a map M is defined in 2.1.1 as 'any location $(x, y) \in M$ that lies in a free cell or on the boundary of a free cell'.

3.2.4 Line of Sight algorithm

In the previous subsection, step 2 requires an algorithm to test whether a line of sight exists between two locations. The Preparation chapter also explained that many of the any-angle path finding algorithms require a line of sight algorithm.

The *Line of Sight* algorithm is based on the pseudo-code in the publication of the *Theta** algorithm [**?**], which itself is a derivative of *Bresenham's line drawing algorithm* [**?**]. *Bresenham's line drawing algorithm* determines which pixels in a raster display should be plotted in order to form a close approximation to a straight line between two given points *a* and *b*, whereas the *Line of Sight* algorithm determines whether any blocked cells are intersected by the straight line between two given points *a* and *b*. *Bresenham's line drawing algorithm* is a useful framework as it avoids any floating-point calculations when the endpoints of the line of sight are integers — this has dual benefits:

- the algorithm is fast;
- the algorithm does not suffer from rounding errors inherent in floating-point calculations.

A notable alteration to *Bresenham's line drawing algorithm* is that while *Bresenham* draws one pixel per column (or one per row) of the raster display, the line of sight algorithm checks every cell through which the line passes, which may require multiple cells to be checked per column (or per row).

For the purposes of clarity, the pseudocode presented in Algorithm 8 assumes that the straight line between locations *a* and *b* is in 'octant 1' - i.e. the angle that the line *ab* makes with the horizontal is between 0° and 45°.

Algorithm 8: LINEOFSIGHT

```
def LineOfSight (a,b)
1     $i \leftarrow a.x$
2     $j \leftarrow a.y$
3     $i_b \leftarrow b.x$
4     $j_b \leftarrow b.y$
5     $\Delta x \leftarrow i_b - i$
6     $\Delta y \leftarrow j_b - j$
7     $s \leftarrow 0$
8     while $i \neq i_b$ do
9         $s \leftarrow s + \Delta y$
10        if $i \geq \Delta x$ then
11            if $cell_{i,j}.isBlocked()$ then
12                return $false$
13            $j \leftarrow j + 1$
14            $s \leftarrow s - 1$
15        if $s \neq 0 \wedge cell_{i,j}.isBlocked()$ then
16            return $false$
17        if $\Delta y = 0 \wedge cell_{i,j}.isBlocked() \wedge cell_{i,j-1}.isBlocked()$ then
18            return $false$
19        $i \leftarrow i + 1$
20    return $true$
```

3.3 Algorithm data

The system uses a `MapInstance` object to encapsulate the concept of a map — with an associated *start* and *goal* location, the graph that represents that map, and the paths and path statistics calculated for that map by the pathfinding algorithms.

Each `MapInstance` has instance variables to store a `Map` object, the grid-based `Graph` and visibility `Graph` objects that represent that map,[2] and a list of `AlgorithmData` objects to store the paths and path statistics for the any-angle pathfinding algorithms that have been run on this map (see Figure 3.6).

There is a concrete class for each pathfinding algorithm that inherits from the abstract class `Algorithm-Data` — which has an *enum* type denoting which pathfinding algorithm the object is holding data for, a public method `go()` which is called to run the pathfinding algorithm, and getter methods which return statistical data about the path found, such as:

Path exists
whether or not the pathfinding algorithm found a valid path between the *start* and *goal*;

Path length
the path length;

Cumulative path angle
the path angle-sum;

Graph calculation time
the duration between the call to `generateGraph()` and `generateGraph()` returning, calculated with `System.nanoTime();`

Path calculation time
the duration between the call to `getPath()` and `getPath()` returning, calculated with `System.nanoTime();`

Number of nodes expanded
the number of nodes that were expanded by the pathfinding algorithm.

3.4 Pathfinding algorithms

To emphasise the close relationship between the algorithms and to allow for maximum code re-use, the concrete classes that implement the algorithms are arranged in a hierarchy that closely reflects the relationship between the algorithms. However, despite *Block A** having a similar basic structure to *A**, a compromise had to be made by having *Block A** inheriting directly from *AlgorithmData* as the actual implementation of *Block A** is too different from *A** to allow any meaningful inheritance in the code (see Figure 3.7).

The inheritance caused by the hierarchy also ensures that any performance differences between algorithms is due to the different nature of each algorithm and not different implementations of similar concepts.

The remainder of this section explains the decisions and strategies employed when implementing the pathfinding algorithms.

[2]`Map` and `Graph` objects have been introduced in subsections 3.1 and 3.2.

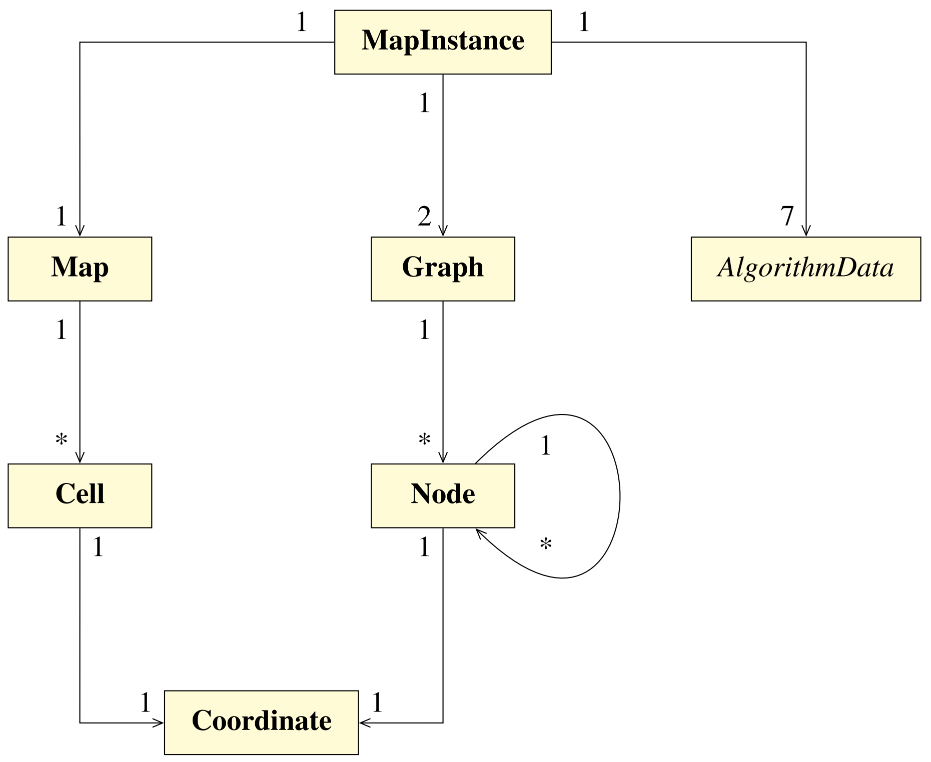


Figure 3.6: Composition of `MapInstance`

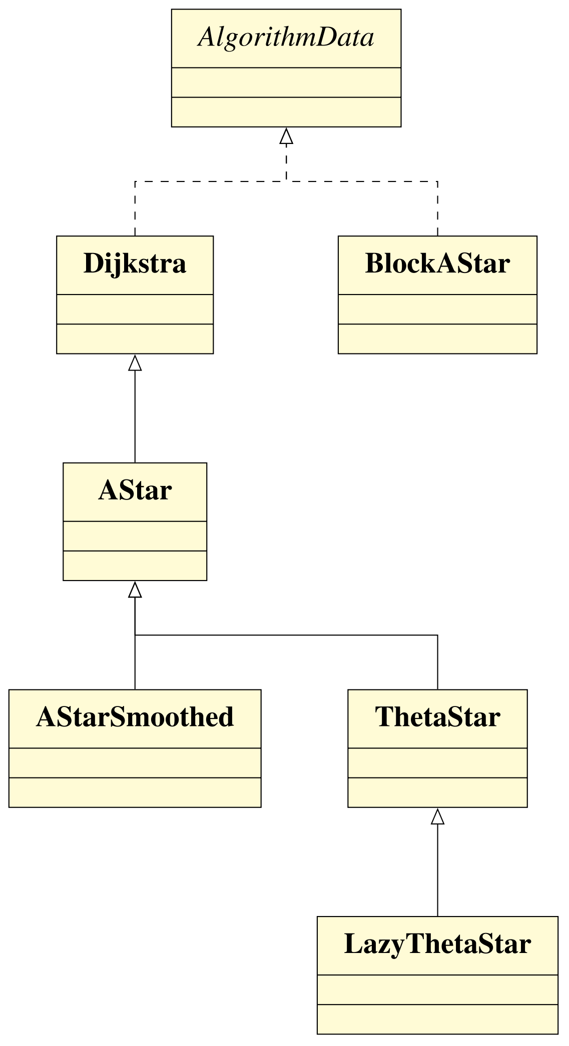


Figure 3.7: Inheritance structure of algorithm implementations

3.4.1 Dijkstra's shortest paths

The implementation is based on the pseudo-code seen in Algorithm 1 in the Preparation chapter. By using the `Graph` and `Node` classes introduced in subsection 3.2.1, the translation from pseudo-code to *Java* code is transparent. The notable implementation decisions are detailed below:

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

Distance calculations
since there are a known set of possible node locations in a map of a given size, costly square root calculations are avoided by using a pre-calculated lookup table for finding the Euclidean distance between coordinates. This is also employed by *A** and its derivatives when calculating *h-value*s.

Extensibility
to allow the inheritance hierarchy shown in Figure 3.7, `Dijkstra` (i.e. the implementation of *Dijkstra*) includes a call to two methods which are required by algorithms that inherit from `Dijkstra`, but are not actually needed by `Dijkstra` itself. The first is `initialise(Node n)` which is called when a node `n` is popped from the `openSet` (which is used by *Lazy Theta**), and the second is a `postProcessing(Node n)` step before the goal node is returned (which is used by *A* with post-processing*). In `Dijkstra` these methods have empty bodies.

3.4.2 A*

`AStar` inherits from `Dijkstra`, and only overrides the `updateCost()` method in accordance with the pseudo-code in Algorithm 2.

3.4.3 A* with post-smoothing

`AStarPostSmoothing` inherits from `AStar`, and implements the post-smoothing by overriding the `postProcessing()` method which had an empty body in `Dijkstra` and `AStar`. As per the majority of the implementations of `Dijkstra` and `AStar`, the code required for the implementation of `AStarPostSmoothing` is a transparent translation of the pseudo-code presented in Algorithm 3.

3.4.4 Theta* and Lazy Theta*

The implementations are based on the pseudo-code in Algorithm 4 and Algorithm 5:

Theta*
inherits from `AStar`, and only overrides the `updateCost()` method. `ThetaStar` uses the implementation of the *Line of Sight* algorithm introduced in subsection 3.2.4;

Lazy Theta*
inherits from `ThetaStar`, and overrides the `initialiseNode()` and `updateCost()` methods. Note that `LazyThetaStar` is the only class to implement `initialiseNode()`.

3.4.5 Block A*

The implementation of *Block A** is by far the most complex, and hence interesting, of all the pathfinding algorithms —partly because it is the most intricate, and partly because Yap et al.'s paper left large parts of the implementation details unspecified. This provides an opportunity to investigate some different implementation strategies which are described in this section and then compared for efficiency and performance in the Evaluation chapter.

LDDB — overview

Recall that subsection 2.2.8 defines a Local Distance Database *LDDB* for block size $N \times N$ as '$LDDB_N$, a pre-computed database that holds the path length and inflection points of the optimum paths between all *ingress-egress* pairs of locations (a,b)–(c,d) for all map configurations M of size $N \times N$, where $a,b,c,d \in \mathbb{Z}^+$ and (a,b)–(c,d) lies on the boundary of M'. Since each block expansion requires knowledge of the shortest paths between *ingress-egress* pairs of coordinates that lie on the boundary of the block, *Block A** speeds up the expansion of blocks by using the *LDDB* to avoid explicitly calculating the paths.

Since there is no publicly available library containing *LDDB*s for *Block A**, the first challenge was to create the *LDDB*s. Although *Block A** only uses an *LDDB* of a single block size, multiple *LDDB*s are required for this project so that performance tradeoffs between *LDDB*s of different block sizes can be investigated. By definition an *LDDB* is applicable to any map, so the *LDDB*s only need to be calculated once for each block size.[3] A method is required to create the *LDDB*s automatically, since for blocks of size $N \times N$ there are 2^{N^2} possible blocks each with $(4 \times (N-1))^2$ pairs of $ingress - egress$ coordinates,[4] which gives over 9 million optimal path calculations for a block size of 4×4.

Each $LDDB_N$ contains:

```
(1)   for every possible map of size N × N:
(2)       for every possible ingress – egress coordinate pair:
(3a)          the shortest path between that pair, and
(3b)          a list containing every intermediate coordinate on that path.
```

The above steps were implemented as follows:

(1) 'every possible map of size $N \times N$' is enumerated by converting each integer from 0 to $2^{N^2} - 1$ to the corresponding map representation — since a map of size $N \times N$ can be represented as a binary integer of N^2 bits where each bit represents a cell in the map, and that cell is free if the corresponding bit is 0 and is blocked if the bit is 1 (see Figure 3.8);

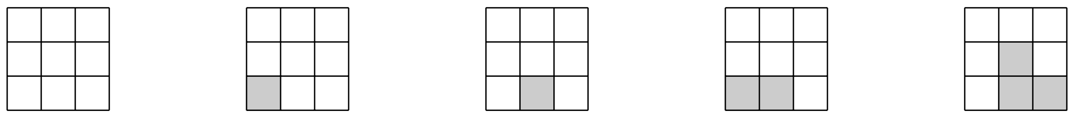
(2) 'every possible $ingress - egress$ coordinate pair' is enumerated by looping over every possible *egress* coordinate for every possible *ingress* coordinate, where the possible *ingress* and *egress* coordinates are the locations of all nodes that lie on the boundary of a block — that is to say, all nodes whose coordinates are (i,j), where $[(i = 0 \vee i = N) \wedge (0 \leq j \leq N)] \vee [(0 \leq i \leq N) \wedge (j - 0 \vee j = N)]$;

(3) the shortest path through a map between a given $ingress - egress$ coordinate pair is found by converting the map to a visibility graph and then finding the optimal path through it with $A*$ (with $ingress = start$ and $egress = goal$). The length required by (3a) is the final value of $n_{goal}.g$,

[3]As explained later in this section, the three block sizes that are investigated in this project are 2×2, 3×3 and 4×4, so only three *LDDB*s were required: $LDDB_2$, $LDDB_3$ and $LDDB_4$.

[4]There are $(4 \times (N-1))$ boundary nodes per block, therefore there are $(4 \times (N-1))^2$ pairs of $ingress - egress$ coordinates per block.

and the list of intermediate coordinates required by (3b) is found by removing the coordinates of $start$ and $goal$ from the list of coordinates in the path returned by $A*$.



(a) 0 (b) 1 (c) 2 (d) 3 (e) 22

Figure 3.8: Integer encoding scheme of maps

LDDB — uncompressed implementation

It is necessary to store the paths and intermediate nodes calculated in steps (3a) and (3b) in a database structure that is compact in memory (so that as much of it as possible can fit into cache), and fast to load and query. Since these constraints are fundamental to the operation of the algorithm, any library or third party database implementation cannot be guaranteed to be sufficiently specialised for the task, so a custom database is implemented using arrays and hash tables to ensure optimal performance and minimum space overhead.

Queries to the *LDDB* require arguments that describe:and will lead to high performance operations on this database

1. **the configuration of the block** (i.e. the configuration of the sub-map that the block represents) — the simple bitwise scheme shown in Figure 3.8 is used to describe the configuration of a block, and the 32 bits of the *Java* `int` primitive is sufficient to encode all possible block configurations of all the block sizes N that are considered in this project:[5] $N = 2, 3, 4$. This encoding scheme has dual benefits:

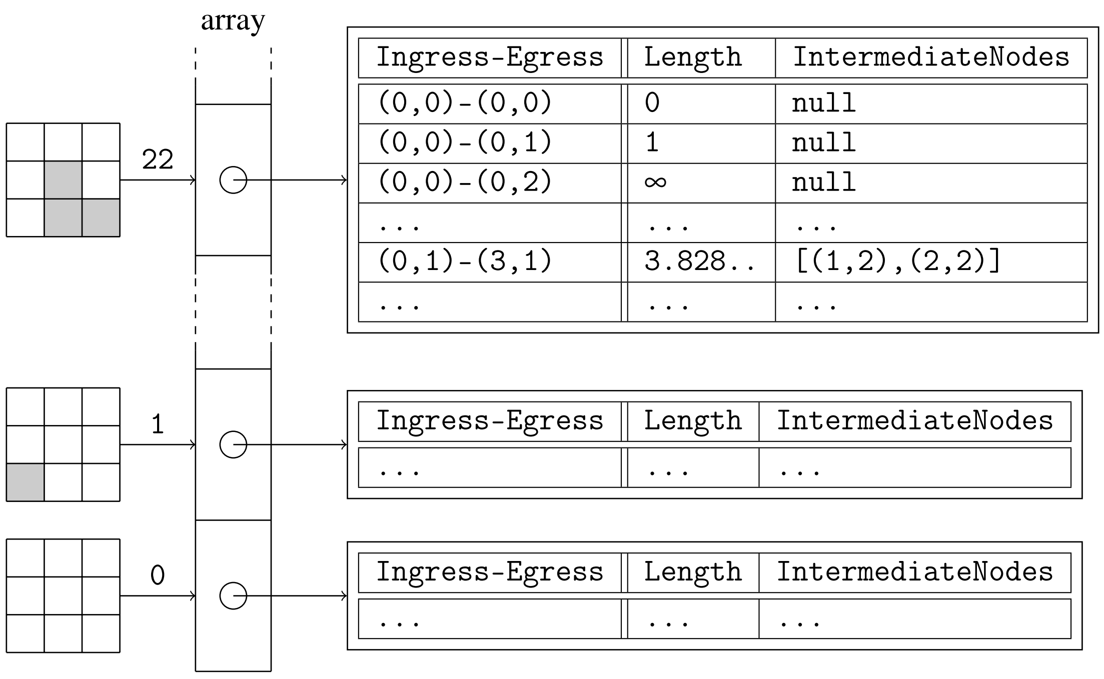
   - the encoded integers are a small (32 bit) value that can be cheaply stored within the `Block` object itself;
   - since these encoded integers form a sequential sequence, they allow the $LDDB_N$ to be implemented as an array of size $2^{N^2} - 1$ which can be indexed by using the encoded integer. This is advantageous as arrays are the most efficient storage format in terms of space and random-access time, and will lead to high performance operations on this database.

2. **an** $ingress - egress$ **coordinate pair** — the $ingress - egress$ coordinate pairs do not fill a sequential space so it would be spatially inefficient to have these coordinate pairs indexing into an array as the array would be sparse. Therefore, the $ingress - egress$ coordinate pairs index into a hash table, which provides relatively fast random access and space-efficiency for a non-sequential, non-uniform key space.

Query results from the *LDDB* are either the length of the shortest path between the nodes, i.e. (3a), or a list of the intermediate coordinates on the shortest path between the *ingress* and the *egress*, i.e. (3b).

Therefore the underlying implementation of the uncompressed implementation database is an array of `HashMaps` — one `HashMap` per block configuration, with the array indexed by the encoded integer that represents the block, and each `HashMap` mapping a key (a `Pair` of $ingress - egress$ `Coordinates`) to a value (a `Pair` consisting of the length (as a `double`) of the shortest path between the *ingress* and the *egress*, and an `ArrayList` of the intermediate `Coordinates` on that path).

[5]The reasons for this are explained in section 4.4.2.



Ingress-Egress	Length	IntermediateNodes
(0,0)-(0,0)	0	null
(0,0)-(0,1)	1	null
(0,0)-(0,2)	∞	null
...	...	...
(0,1)-(3,1)	3.828..	[(1,2),(2,2)]
...	...	...

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

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

Figure 3.9: Extract of the uncompressed implementation of *LDDB*$_3$, detailing part of the `HashTable` for the block configuration which has an encoded integer representation of 22

The next three subsections build upon this uncompressed implementation by describing some different compression techniques that were devised for the *LDDB*, and three different strategies for obtaining the edge weights in the special case blocks. All of the different resulting forms of *LDDB* involve tradeoffs between space and time which are analysed in the Evaluation chapter.

LDDB — bitwise-compressed implementation
An implementation was devised that makes use of a further bitwise encoding scheme to encode `PairOfCoords` and `List<Coordinate>`:

1. each `Pair` of *ingress-egress* `Coordinates` can be represented with a unique integer in a `byte`'s worth of space. An efficient hash function was devised to convert from a `PairOfCoords` object to this encoded byte representation that uses bit shifting to represent each coordinate with 6 bits, since the range of x and y in each `Coordinate` is 0-4 for a sub-map of size up to $N = 4$, which is representable with 3 bits (see Figure 3.10).

2. the `List` of intermediate `Coordinates` can be represented by a unique code that fits into the 32 bits of an integer by using the scheme identical to point 1, since the maximum[6] number of intermediate nodes on a shortest path is 4 for a sub-map of size up to $N = 4$, which gives a maximum total of $4 \times 6 = 24$ bits.

Furthermore, the length of a path (stored as a 32 bit `float`) and the integer that represents the intermediate coordinates can then be compressed into one 64 bit `long` integer (which is the native register size of most modern architectures), to produce one 64 bit *LDDB* entry per $ingress - egress$ pair.

LDDB — geometrically-compressed implementation
The bitwise-compressed implementation of the database uses bitwise encoding schemes to compress the database. Further approaches to compression are possible by taking advantage of the symmetry between different block configurations and $ingress - egress$ pairs:

[6]Obtained by querying the length of the intermediate coordinate list for every entry in the uncompressed *LDDB*$_4$.

```java
private int getCode(Coordinate c1, Coordinate c2) {
  int code = 0;
  code = code | c1.getX();
  code = code << 3;
  code = code | c1.getY();
  code = code << 3;
  code = code | c2.getX();
  code = code << 3;
  code = code | c2.getY();
  return code;
}
```

Figure 3.10: Encoding scheme for `PairOfCoords` in the bitwise-compressed *LDDB*

1. some block configurations are 90°, 180° or 270° rotations of other block configurations;
2. the shortest path between an $ingress - egress$ pair a and b is the reverse of the shortest path between an $ingress - egress$ pair b and a.

These two properties can be used to leverage redundancy in the database as follows:

1. The data for only one of the four rotations of block configuration needs to be stored in the database. A simple policy is used to determine which of the rotations is stored — the rotation with the smallest encoded integer representation (as introduced for the uncompressed implementation of the *LDDB*) is selected to be stored. Note that if the block configuration does need to be rotated in the query, then the intermediate nodes in the query result will need to be rotated by the same amount in the opposite direction.
2. The data for only one of the two 'reflection' pairs of $ingress - egress$ coordinates a, b and b, a needs to be stored in the database. A simple policy is used to determine which one of these two pairs is stored — if $a.y < b.y$ or $(a.y = b.y) \wedge (a.x \leq b.y)$ then the pair will be found as a, b in the database, otherwise they are found as b, a. Note that if the $ingress - egress$ pair needs to be reversed in the query, then the intermediate nodes in the query result will also need to be reversed.

Figure 3.11 gives an annotated explanation of a query to a bitwise-and-geometrically compressed *LDDB* — see Appendix A.6 for an extended explanation. In the actual implementation, some of this work is saved by storing the rotation value (calculated in step (b)) in an instance variable of each `Block`. Furthermore the coordinate rotations (calculated in steps (b) and (g)) take only one of four forms (since the blocks have rotational symmetry of up to four), hence the pre-rotation to post-rotation (and vice versa) coordinate mappings are obtained by querying one of four small pre-calculated tables stored as `static` variables of the `Block` class.

Special case blocks

As explained in 2.2.8, the edge weights for $block_{start}$ and $block_{goal}$ may not be present in the *LDDB* because the *LDDB* only contains data for paths between *ingress* and *egress* coordinates (which lie on the boundaries of blocks), yet *start* and *goal* are not guaranteed to lie on the boundary of a block. Therefore, alternative methods must be used to obtain the edge weights for $block_{start}$ and $block_{goal}$:

Option 1: **compute the edge weights at runtime**, by using a similar method to that which is used to compute the shortest paths contained in the *LDDB* — i.e. by generating a visibility graph to represent the sub-map that $block_{start}$ represents, and using $A*$ to find the shortest path between n_{start} and all boundary nodes of $block_{start}$. This procedure is then repeated for $block_{goal}$. In the

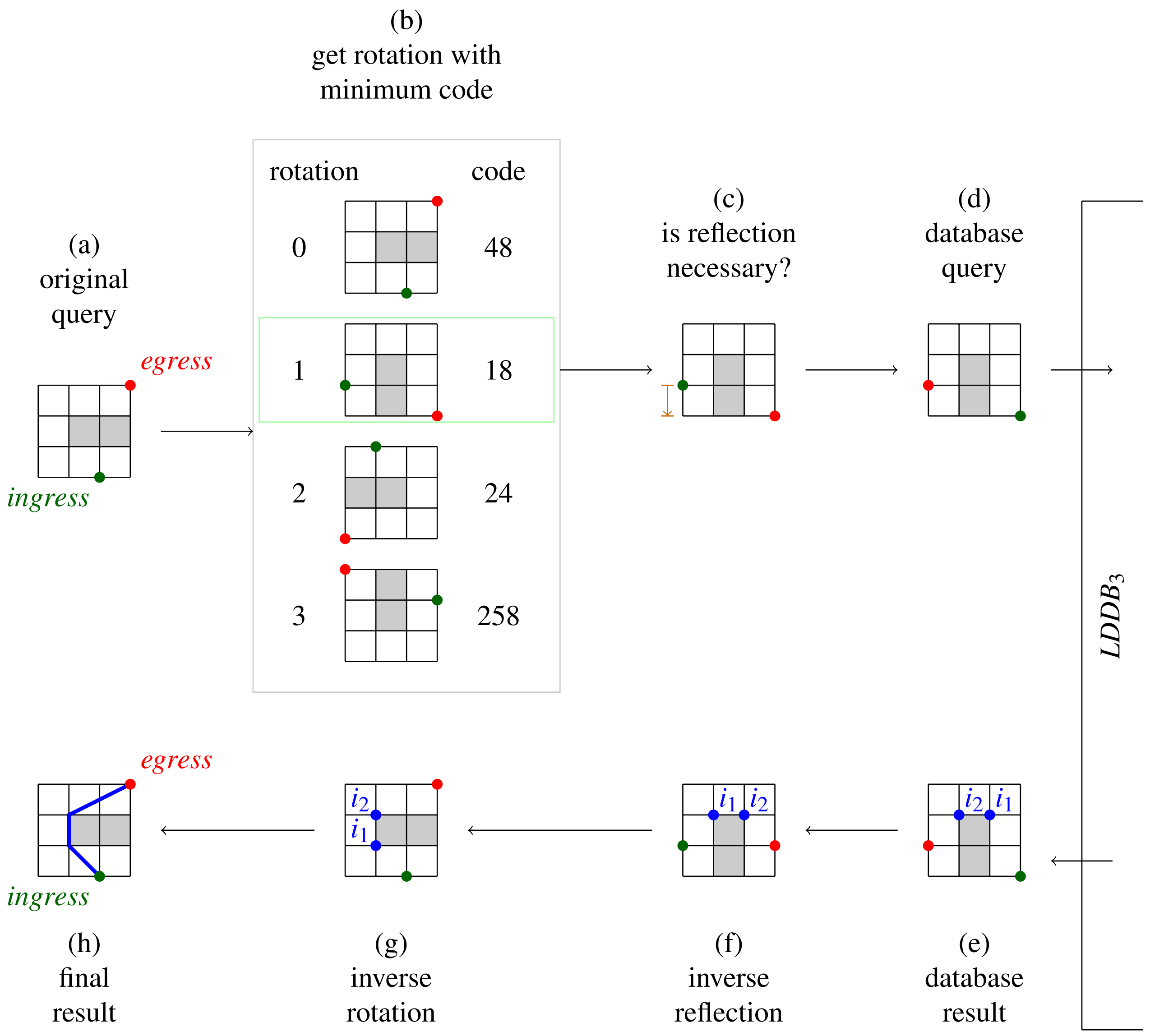


Figure 3.11: Annotated explanation of a bitwise-and-geometrically-compressed *LDDB*$_3$ query

exceptional case that $block_{start} = block_{goal}$, this method can be slightly modified to return the shortest path directly.

Option 2: **extend the LDDB** so that it also includes data for all possible edges in $block_{start}$ and $block_{goal}$. The unextended *LDDB* (regardless of whether of not it is bitwise or geometrically compressed) contains details of shortest paths from *any boundary* node to *any boundary* node in a block, so there are two possibilities for extension:

(a) **semi-extended** *LDDB* contains details of shortest paths from *any* node to *any boundary* node in a block. This *LDDB* will have all data required by *Block A** for all blocks, except for in the exceptional case that $block_{start} = block_{goal}$, in which case a visibility graph of the block must be calculated and solved with *A** as in *Option 1.*

(b) **fully-extended** *LDDB* contains details of shortest paths from *any* node to *any* node in a block. This *LDDB* will have all data required by *Block A** for all blocks.

Algorithm
Having carefully set up the problem in the Preparation chapter, the core part of the implementation of the algorithm is a translation of the pseudo-code in Algorithm 6 using similar design choices to the other pathfinding algorithms. However, the pseudo-code does not specify the implementation of the creation of `Init`, `TraceBack` or the creation of blocks, which also require the application of many of the techniques and design choices seen in the other algorithms, but to a far more complicated level. The `BlockAStar` class itself is over 400 lines of *Java* (excluding comments) — note that this class does not even include the implementation of blocks or the creation and decoding of the various *LDDB* implementations (see Figure 3.12).

Traceback
Finally the traceback stage, which is unspecified in Yap et al.'s paper, is explained as follows. The path P_{G_M} is produced by:

- adding all nodes obtained by recursively following the *parent* pointers from n_{goal} to n_{start};
- removing all but one of any set of co-located nodes — this occurs when the path crosses block boundaries;
- adding any intermediate nodes where the path traverses a block — these are obtained by querying the *LDDB*.

3.5 Visualizer

The maps and paths are displayed by implementing the `paintComponent()` method of the visualizer panel.

The UI also allows the user to display or hide different combinations of paths and the nodes which were expanded in the calculation of those paths for all of the different pathfinding algorithms (see Figure 3.13), by using *Swing* components whose `ActionListeners` send messages to the class that contains the visualizer.

3.6 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. All unit tests use the *JUnit* library.

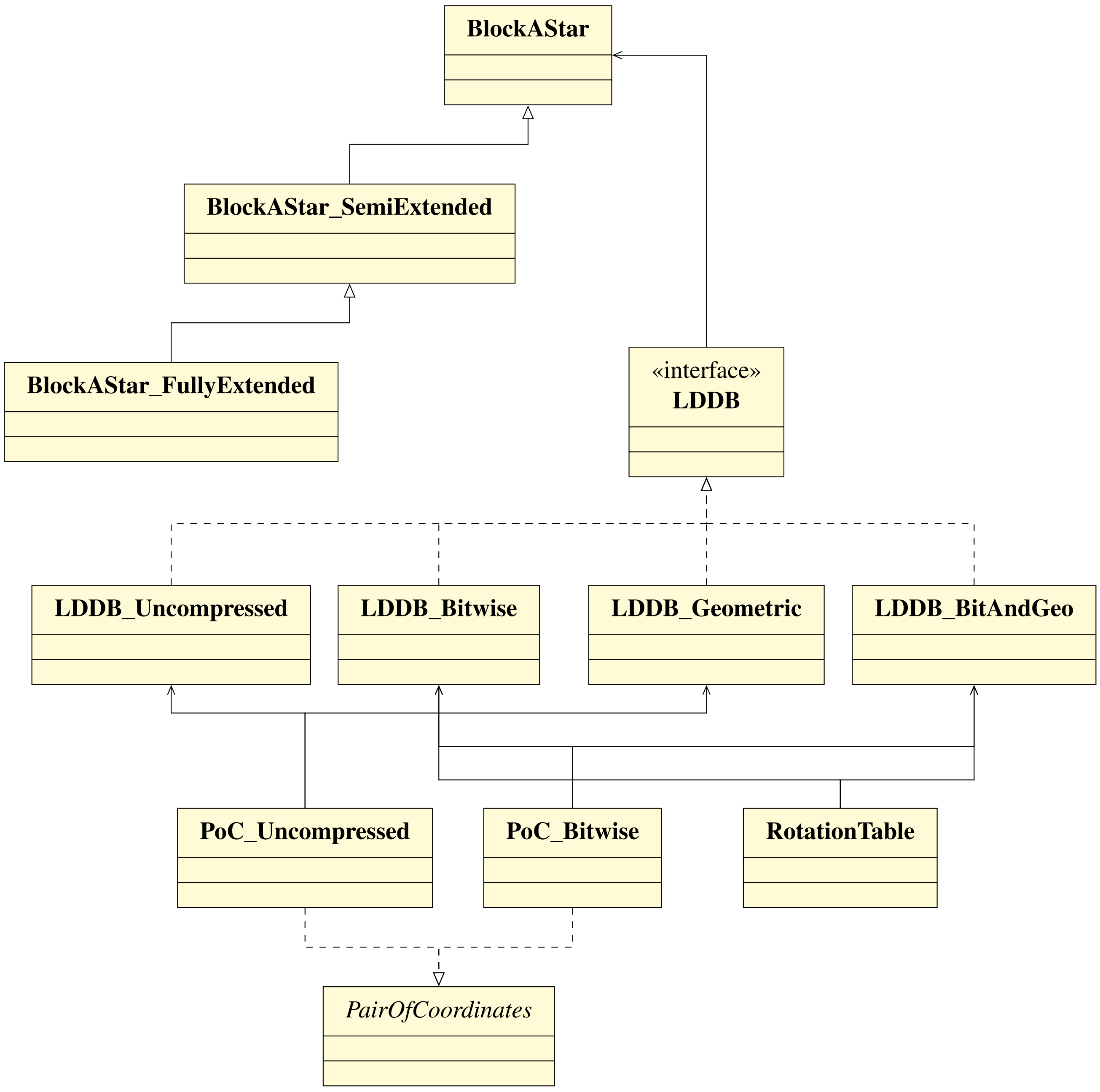
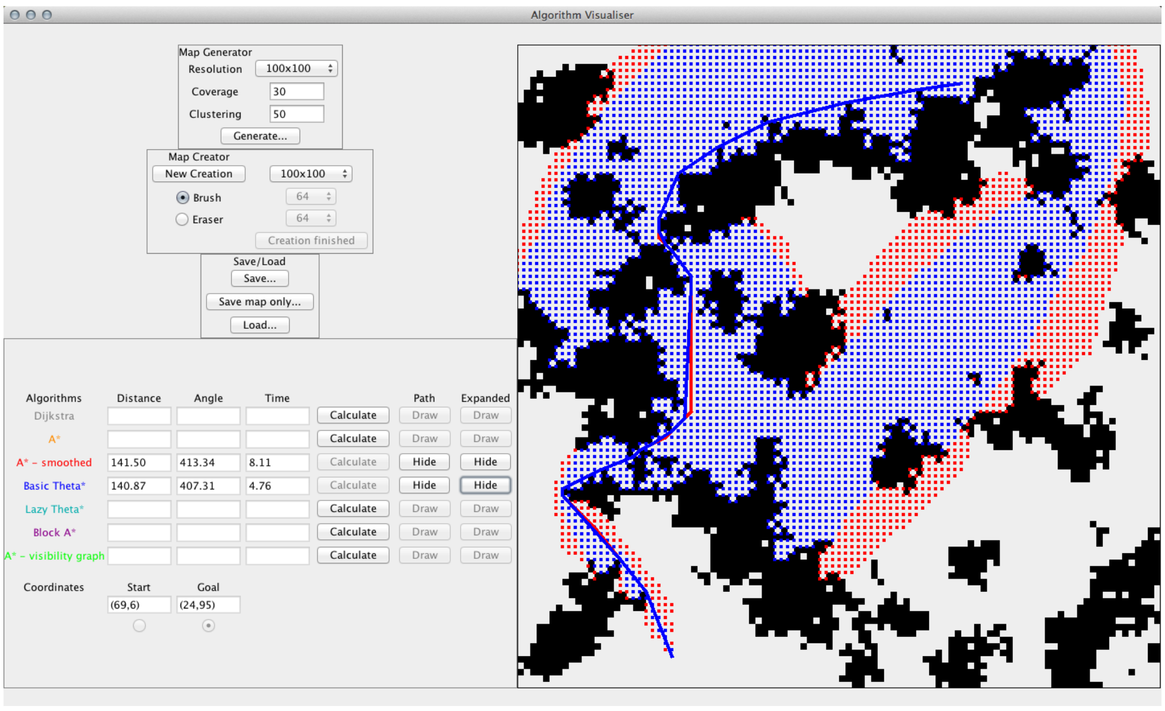


Figure 3.12: Class structure of different implementations of *Block A**



Map Generator
Resolution 100x100
Coverage 30
Clustering 50

Map Creator
100x100
[✓ Brush] 64
Eraser 64

Algorithms	Distance	Angle	Time		Path	Expanded
Dijkstra				Calculate	Draw	Draw
A*				Calculate	Draw	Draw
A* – smoothed	141.50	413.34	8.11	Calculate	Hide	Hide
Basic Theta*	140.87	407.31	4.76	Calculate	Hide	Hide
Lazy Theta*				Calculate	Draw	Draw
Block A*				Calculate	Draw	Draw
A* – visibility graph				Calculate	Draw	Draw

Coordinates	Start	Goal
	(69,6)	[✓ (24,95)]

Figure 3.13: Graphical User Interface — the visualiser is shown on the right side

Simulator

Core structural classes — basic functionality is unit tested (see Figure 3.14).

Pathfinding algorithms — correctness of all paths returned is optionally verified, using the following invariants:

- if one algorithm finds a path, all algorithms find a path;
- the lengths of the paths returned by the algorithms obey the inequalities: *Dijkstra* = $A*$, $A* \geq A*$ *with post-smoothing*, $A* \geq$ *Theta*$*$, $A* \geq$ *Lazy Theta*$*$, $A* \geq$ *Block* $A*$, all algorithms $\geq A*$ *on a visibility graph*.

Data extraction scripts

Exported CSVs — unit tests compare the contents of CSV files to the pathfinding data output of the API.

3.7 Data extraction

The engine has a simple API which allows scripts to access the engine directly, without any interaction with the UI. The open source package `CSVWriter` is used to write the pathfinding data obtained from the API to CSV files (see Figure 3.15). This data is then processed by *R* scripts (see Figure 3.16) embedded in the LaTeX source file, which parse the imported CSVs into *dataframes* and apply various statistical analyses to produce the data summarised in the Evaluation chapter.

```java
public class GraphGeneratorStartTest {
 @Test
 public void test() {
  int[][] array2D = new int[4][3];
  array2D[0][0]=1; array2D[1][0]=2; array2D[2][0]=4; array2D[3][0]=0;
  array2D[0][1]=0; array2D[1][1]=3; array2D[2][1]=0; array2D[3][1]=1;
  array2D[0][2]=3; array2D[1][2]=2; array2D[2][2]=0; array2D[3][2]=1;
  Map map = new Map(array2D);
  Coordinate start = new Coordinate(0,0);
  Coordinate goal = new Coordinate(4,4);
  Graph graph = GraphGenerator.generateGridGraph(map, start, goal);
  Node head = graph.getStart();
  assertEquals("Head does not have correct neighbours", head.toString(),
   "Coordinate: (0,0), Neighbours: (1,0) (1,1) ");
 }
}
```

Figure 3.14: Unit test to check that n_{start} of a grid-graph has the correct neighbours

```
"Map","Algorithm","AlgorithmTime","Distance","Angle","NodesExpanded"
"Map 1","Dijkstra","29.92716","299.244728565","675.000198016","34471"
"Map 1","AStar","23.91603","299.244728565","1215.000190772","10352"
"Map 1","AStarSmoothed","24.80595","289.930193901","182.805389664","10579"
```

Figure 3.15: An extract from a CSV file exported by `DataExtract`

```r
plotFigure <- function() {
    df <- read.csv("200_20_50.csv")
    AStar  <- (subset(df,Algo=="AStar"))[,"Expansions"]
    ThetaStar  <- (subset(df,Algo=="ThetaStar"))[,"Expansions"]
    BlockAStar  <- (subset(df,Algo=="BlkAStar"))[,"Expansions"]]
    DF <- data.frame(
      AStar=AStar, ThetaStar=ThetaStar, BlockAStar=BlockAStar)
    DF <- melt(DF)
    DF$variable<-factor(DF$variable,
      levels=c("AStar", "ThetaStar", "BlockAStar"),
      labels=c("{\\em A*}","{\\em Theta*}","{\\em Block A*}"))
    p <-  ggplot(DF)
    p <- p + geom_boxplot(aes(x=variable,y=value))
    p <- p + labs(x="", y="Path length/Optimal path length")
    tikz('figure.tex', width=6.5, height=4.5)
    print(p)
    dev.off()
}
```

Figure 3.16: Code snippet showing automatic generation of plots coded in *R*

4 | Evaluation

This chapter compares the performance of the any-angle pathfinding algorithms introduced in this dissertation. The first section identifies a suitable set of maps to use for the comparisons, the second section focuses on the optimality of the paths produced, and the remainder focuses on the computation time required by each pathfinding algorithm.

Throughout this chapter all maps are of a standard size of 200×200, where $start = (0,0)$ and $end = (200,200)$ unless otherwise stated. This ensures consistency across analyses.

4.1 Map generation algorithm

The bespoke map generation algorithm introduced in 3.1.2 is designed so that large volumes of maps can be created to allow for statistical analysis of the algorithms over pseudo-randomly created maps with certain known properties.

For a fixed clustering score D, Figure 4.1 shows that the graphs generated from these maps display the characteristics of a classic percolation problem [**?**]. Percolation problems manifest in random networks, where the networks tend to form a giant component when a certain parameter is above the percolation threshold, but not when the parameter is below the threshold. In this instance, the parameter is the coverage percentage C. It is also noticeable that the percolation problem characteristic is weaker for higher values of D as seen by the flatter sigmoid shape, since a positive clustering score D causes a map (and hence its graph representation) to be less random.

The confidence intervals are calculated using the standard error of $\hat{p}$, the estimated proportion in the population [**?**], as:

$$\hat{p} \pm 1.96 \times \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{4.1}$$

where n is the sample size of 400, and the multiplier of 1.96 indicates a confidence level of 95%.

This analysis reveals that the percolation threshold for generated maps is in the range of $C = 30 - 40\%$. Therefore from this point in the evaluation, all graphs are plotted from data obtained from a standard set of 100 maps of the *standard configuration* with coverage $C = 20\%$ and clustering score of $D = 50$ to ensure with high probability that the maps used for testing have a valid path from $start$ to $finish$.

4.2 Path optimality

Subsection 2.1.3 defines a path through a map M as optimal 'if there do not exist any paths through M from $start$ to $goal$ with a shorter path length and a smaller path angle-sum'. This section investigates the optimality of the paths found by both classic and any-angle pathfinding algorithms.

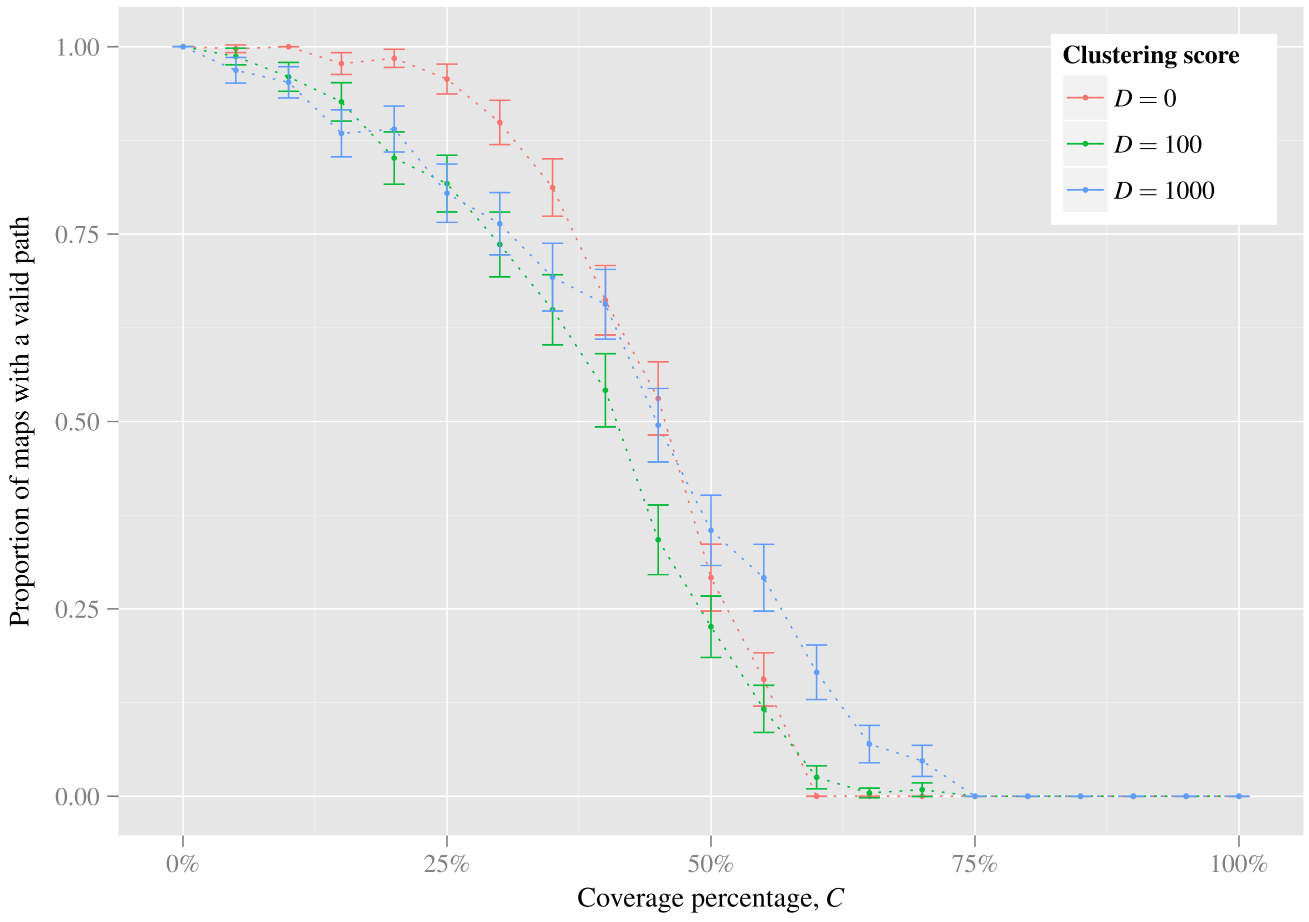


Figure 4.1: Percentage of generated maps that have a valid path between *start* and *goal* for different coverage percentages C and clustering scores D

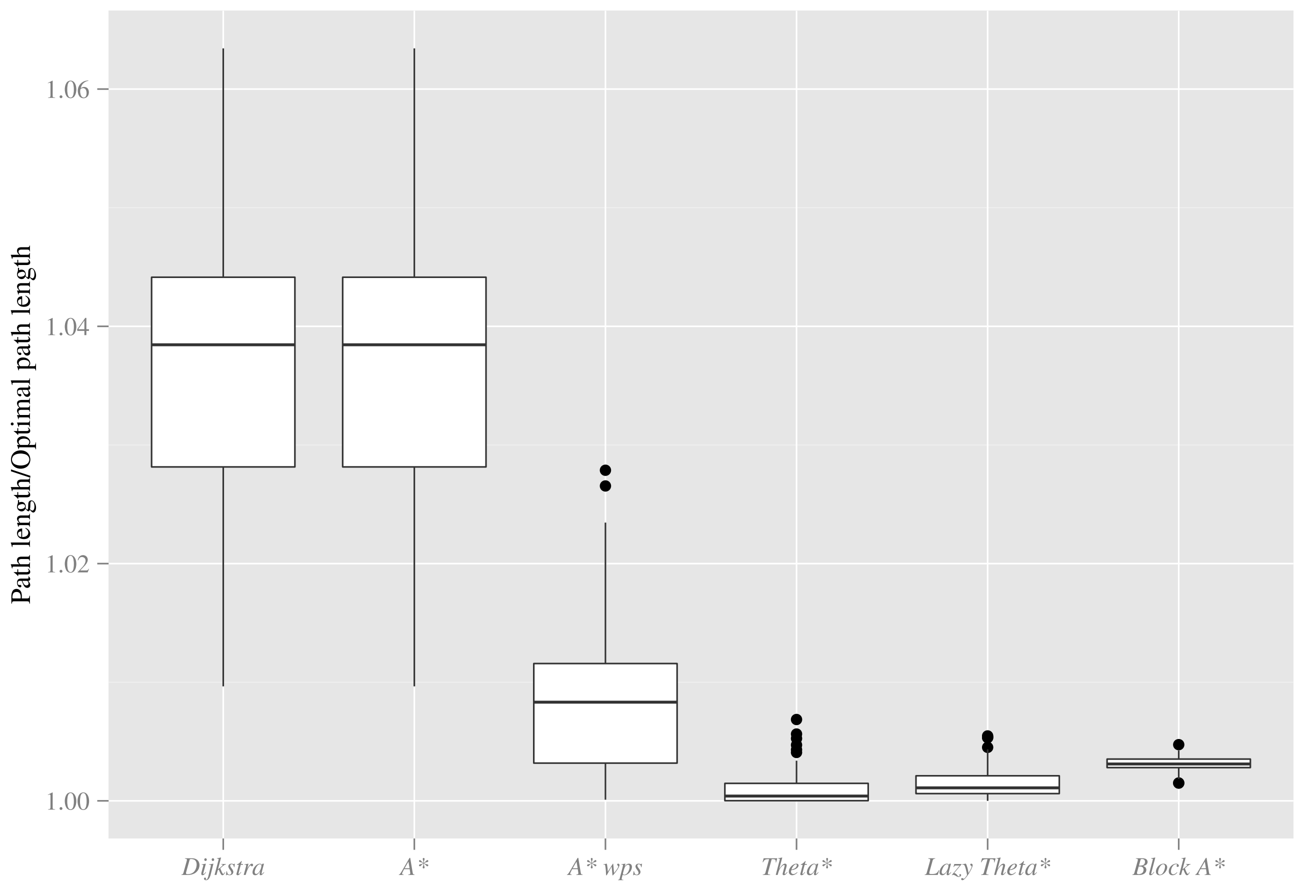


Figure 4.2: Path lengths of pathfinding algorithms as a fraction of the optimal path length

4.2.1 Path length

Figure 4.2 shows the distribution over 100 maps of the *standard configuration* of path lengths for the different algorithms as a fraction of the optimal length (where the optimal length is found by running *A** over a visibility graph).The figure confirms that the any-angle pathfinding algorithms do indeed find shorter paths than the classic algorithms, but the margins are small (the median path length for the any-angle pathfinding algorithms is approximately 4% shorter than those of the classic algorithms).

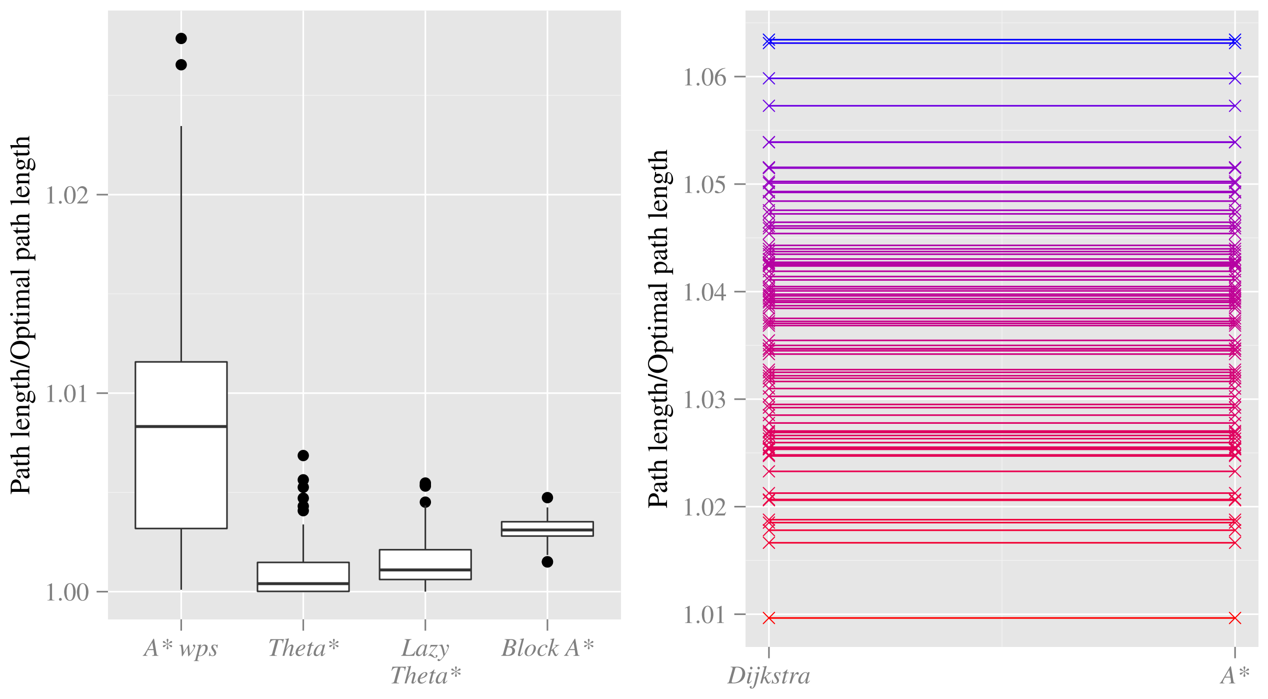
Figure 4.3(a), which is an enlarged section of Figure 4.2, emphases that there is significant overlap between the distributions of the path lengths found by the any-angle algorithms, and there is no single algorithm that is clearly superior to the others in relation to path length.

Additionally, the assertion in section 2.2 that *Dijkstra* and *A** both find identical length paths is supported by Figure 4.3(b),[1] in which data points from the same map are joined with a horizontal line.

4.2.2 Path angle-sum

Figure 4.4 reveals the core advantage of any-angle pathfinding algorithms, which is significantly smaller path angle-sums than the classic algorithms. While the any-angle algorithms improve path

[1]*Dijkstra* and *A** both find the shortest path through a *graph* — but recall that this does not imply the shortest (let alone the optimal) path through the *map* unless a visibility graph is used.



(a) Any-angle algorithms (b) Classic pathfinding algorithms

Figure 4.3: Comparing path lengths of different classes of pathfinding algorithms

length by approximately 4% from the classic algorithms (as seen in section 4.2.1), they improve the path angle-sum from a worst case of 10,000% larger than the optimal to a worst case of 250% larger for *Theta** (see Figure 4.5(a)).

In addition, Figure 4.5(a) shows that as with the path lengths, the distributions of the angle-sums have significant overlap. It is important to note that *Theta** finds the paths with both the shortest path lengths and smallest angle-sums, closely followed by *Lazy Theta**.

The significance of the differences in the path angle-sum becomes clear when the applications of pathfinding algorithms are considered. Changing trajectory is invariably a process that causes an agent to slow its rate of travel, whether the agent is humanoid or vehicular. Therefore even if two paths have the same or similar lengths, the path with the larger angle-sum will not only be temporally inefficient (i.e. it will take a longer *time* to traverse), but in addition I propose that it will look ‘strange’ to the human eye (see Figure 4.5(b)) which is undesirable if the application is using the algorithm to approximate human behaviour, such as in computer games.

4.2.3 Block A*

*Block A** paths are constrained to be at an $(integer, integer)$ coordinate at every block boundary, so that they often oscillate on either side of the optimal (see Figure 4.6(a)). This implies that running *Block A** with smaller blocks results in longer paths and larger angle-sums, since there will be more

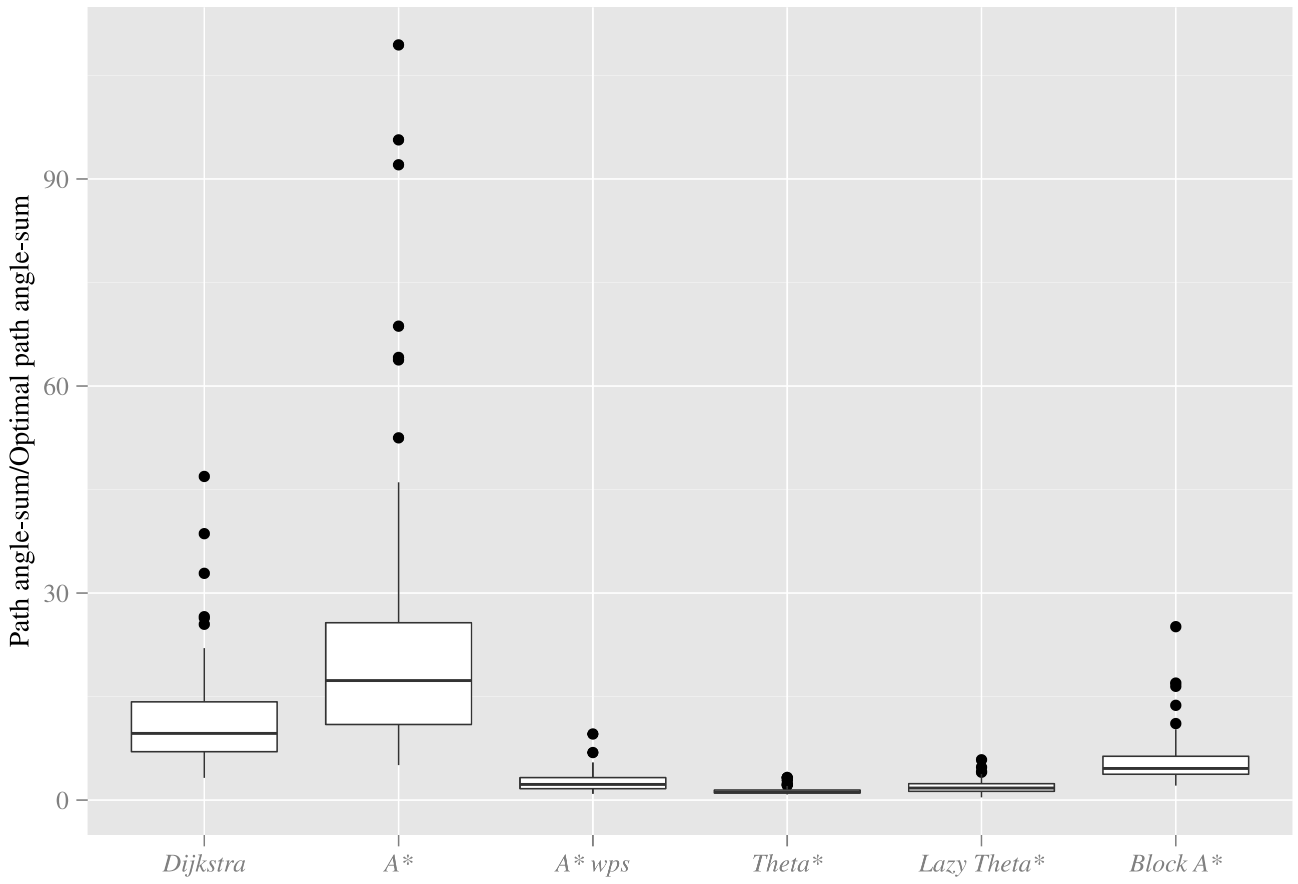
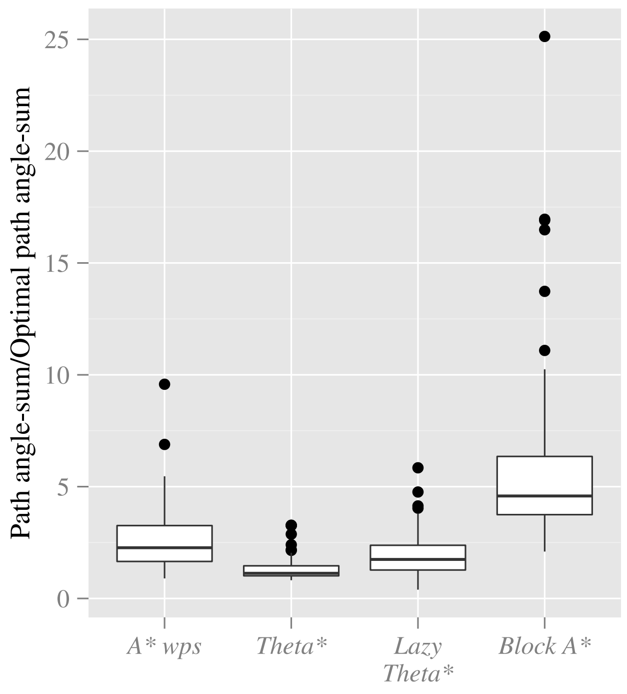
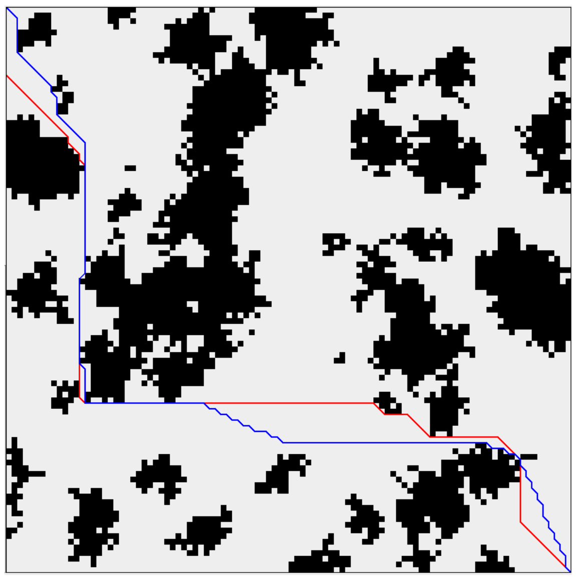


Figure 4.4: Path angle-sums computed by pathfinding algorithms as a fraction of the angle-sum of the optimal path



(a) Path angle-sums



(b) Two paths of equal length — the blue path has a larger path angle-sum

Figure 4.5: Angle-sums of paths computer by any-angle pathfinding algorithms

points on the path constrained to $(integer, integer)$ coordinates. Figure 4.6(b) confirms this.[2,3]

It should be noted that there are no published papers or meta-analyses which investigate the effect of the path angle-sum of *Block A** compared with other any-angle pathfinding algorithms. Further studies would be required to investigate the evidence presented in this subsection, the claim concerning humanoid paths, and the implications on the development of further any-angle pathfinding algorithms.

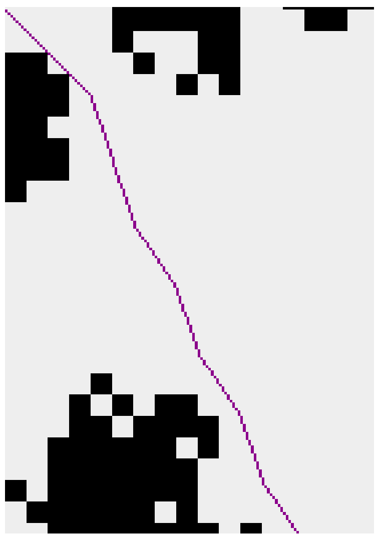
4.2.4 Explanation

In order to choose the most suitable pathfinding algorithm for a certain situation, an understanding of the reasons for the tradeoffs observed in this section is required. Therefore three of the key observations from the figures above are provided:

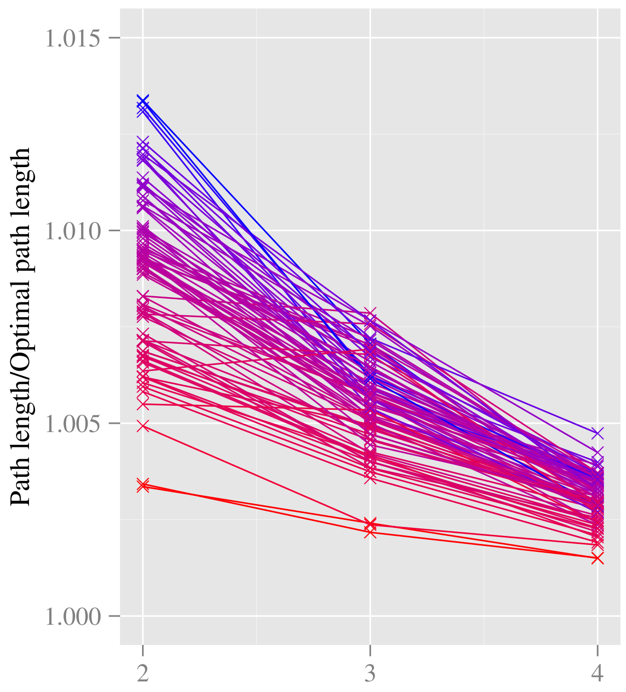
- *A** produces paths with a much larger path angle-sum that *Dijkstra* — because all subpaths on the *Dijkstra* path are optimal in length with respect to the graph, whereas *A** uses a heuristic that only guarantees that the entire path is optimal in length with respect to the graph;
- of the any-angle algorithms, *A* with post-smoothing* tends to produce the longest paths with the largest path angle-sum — because the paths returned by *A* with post-smoothing* are constrained by having their inflection points lying on the suboptimal path found by *A**;

[2]Note that all *Block A** data in this chapter uses a block size of 4×4 unless otherwise stated.

[3]This style of graph is used regularly throughout this chapter — all data points obtained from the same map are plotted with the same colour and joined with lines of that colour, where this colour is calculated according that map's y-axis value in the left most x-axis category.



(a) Path oscillates around optimum



(b) Path lengths for different block sizes

Figure 4.6: Properties of *Block A** paths

- the lowest end of the range of *Block A**'s path lengths and angle-sums is longer than that of the other algorithms — because of the $(integer, integer)$ constraint explained in the previous subsection.

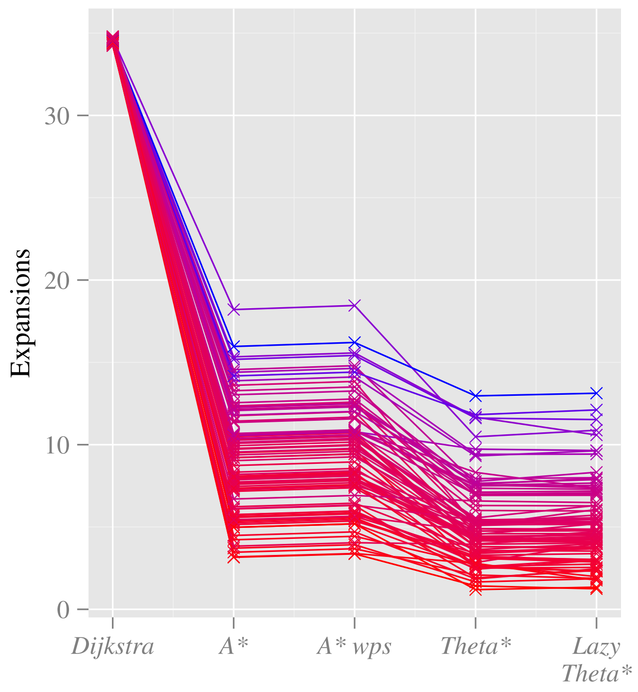
4.3 Expansions

This section follows the standard [**?**] practice of any-angle pathfinding research by utilising the number of node expansions as an approximation to a machine-independent measure of algorithmic performance.

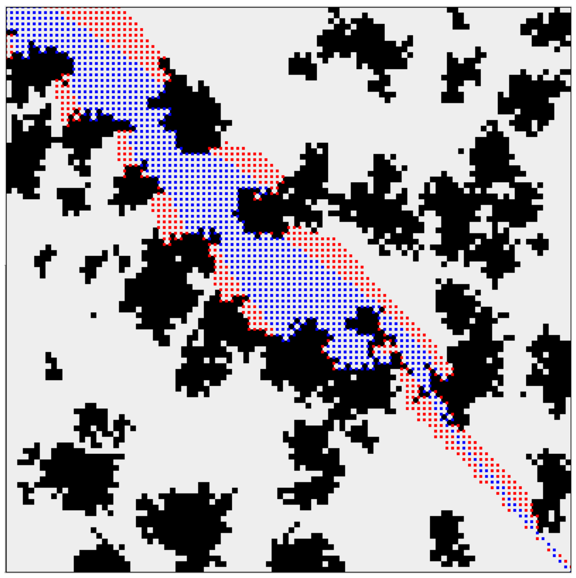
To investigate the number of *expansion*s required for an algorithm to find a path through a map, an *expansion* must be defined. The definition for *Dijkstra*, *A**, *Theta** and *Lazy Theta** is obtained directly from the definition of a node expansion found in the pseudo-code provided in the Preparation chapter, which can be reasonably extended to the definition of *A* with post-smoothing* by including each node that is evaluated in the post-processing step. However, what constitutes an *expansion* in *Block A** is more subtle, and will be handled separately.

Figure 4.7(a) reveals three important observations which are introduced and explained below:

- *Dijkstra* expands far more nodes than the other algorithms — because it is the only algorithm that does not use the Euclidean distance heuristic;
- *A* with post-smoothing* only expands a few more nodes than *A** — since the path returned by *A** (of which the post-smoothing step processes a subset) includes only a small subset of the nodes in the graph;



(a) Expansions by different algorithms



(b) *Theta** (blue) expands fewer nodes than *A** (red)

Figure 4.7: Number of expansions required by pathfinding algorithms

- *Theta** and its derivative *Lazy Theta** consistently expand approximately 25% fewer nodes than *A* with post smoothing* — this surprising result is visualised in Figure 4.7(b), which shows the nodes expanded by *A** and *Theta** for a certain map. This difference occurs because the *g-score* of a given node in a map will generally be higher when the graph is solved by *A** than by *Theta** (because *Theta** interleaves smoothing with exploration) while the *h-score*s will be identical. Therefore the effect of the heuristic on the *f-score* (which is used to select which node to expand next in the algorithms) is less significant when *A** is used.[4]

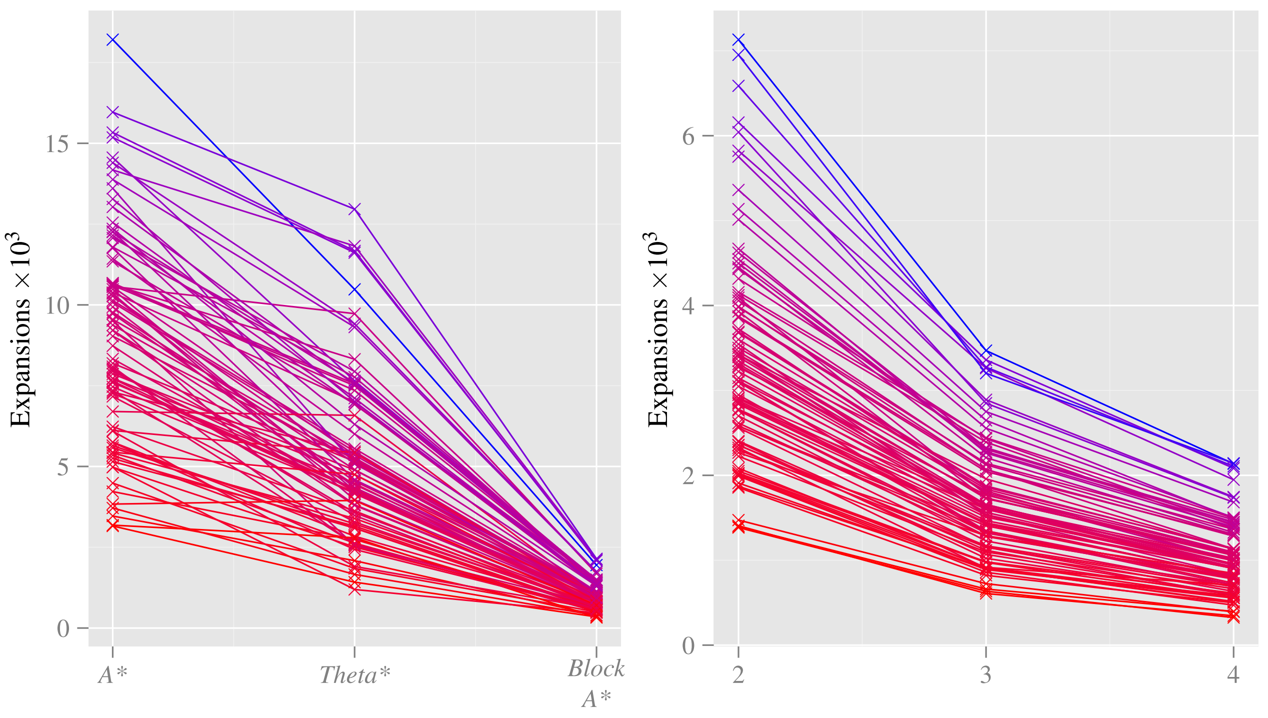
4.3.1 Block A*

In Yap et al.'s publication of *Block A**, an *expansion* in *Block A** is defined as a block expansion. With this definition, Figure 4.8 confirms the findings of Yap et al. by showing that *Block A** requires far fewer expansions than *A** and *Theta**, and that the advantage of using larger blocks is that fewer expansions are required.

However, even despite the author's warning that "the number of expansions done by each algorithm. . . needs to be read with care", I propose that this is a needlessly misleading metric.[5] Lines 1 – 3 of `Expand` of Algorithm 6 reveal that each expansion of $block_{i,j}$ requires, for each node n in $openSet_{i,j}$, an attempt to relax every edge to which n is connected. Therefore there is a clear parallel between the processing of each node in $openSet_{i,j}$ during *Block A** and the regular node expansion seen in the other pathfinding algorithms, since both attempt to relax the edge between the node and its fixed number

[4]The development of *Weighted A** [**?**], which was published many years before *Theta**, artificially achieves this effect by increasing the *h-score* by a small fraction, which results in fewer node expansions at the cost of optimality.

[5]I suggested this in an email to the author, but he declined to agree.

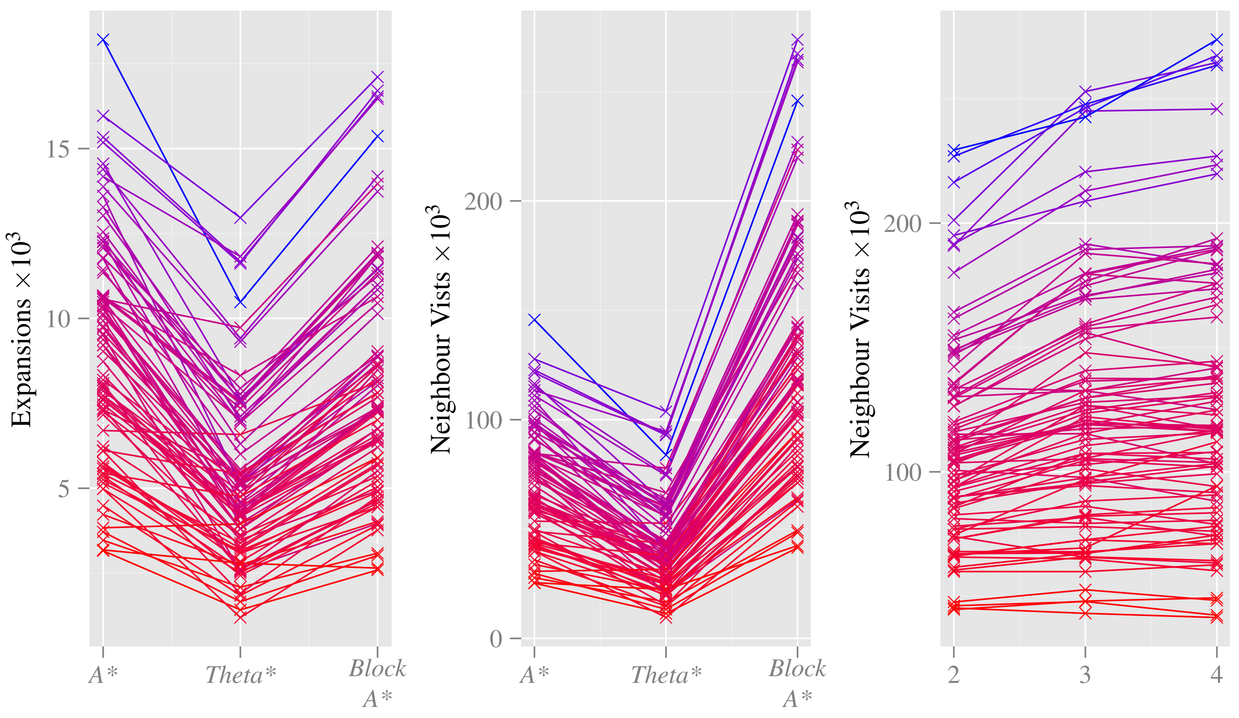


(a) Expansions by pathfinding algorithms (b) Expansions as block size changes

Figure 4.8: Results that agree with Yap et al.'s paper —
where an expansion in *Block A** is defined as an expansion of a block

of neighbours. Figure 4.9(a) indicates that when this metric is used the performance of *Block A** in comparison to these algorithms is less favourable.

Furthermore, a node expansion in *Block A** requires more attempted edge relaxations than the other algorithms because *Block A** uses a completely different graph structure — therefore, using the number of attempted edge relaxations (normally known as *neighbour visits*) instead of node expansions provides a normalised metric. All of the pathfinding algorithms perform a maximum of 8 *neighbour visits* per node expansion, whereas for *Block A** the number of *neighbour visits* increases with block size, from a constant 8 for a block size of 2×2, to 12 for 3×3 and 16 for 4×4. Figure 4.9(b) shows that the performance of *Block A** appears even less favourable with this metric, and Figure 4.9(c) shows that the total number of neighbour visits actually increases when *Block A** is run with a larger block size. Therefore, using neighbour visits (a normalised version of expansions) as a performance predictor implies that the entire premise behind *Block A** may not be sound.



(a) Expansions (b) Neighbour visits (c) Neighbour visits

Figure 4.9: Expansions and neighbour visits in *Block A** — where an expansion in *Block A** is defined as an expansion of a node

4.4 Computation time

The computation time of a path through a map is critical to the performance of an application that utilities a pathfinding algorithm.

4.4.1 Benchmarking

The primary metric used for comparison between algorithms is the median execution time — this choice is motivated by the fact that the user has extremely limited control over when the *Java Virtual Machine* initiates garbage collection, so the mean and maximum times for a certain algorithm can be skewed by garbage collection. However, the maximum execution times (caused by garbage collection) are not ignored, but discussed separately in subsection 4.4.2.

The execution times computed in this section use a *Java* call to the system clock: `System.nanoTime()`. A microbenchmarking tool such as *Caliper* [**?**] was considered for a more fine-grained approach, but such tools are still too coarse-grained to give more informative timings than `System.nanoTime()`.[6]

4.4.2 Block A*

In order to compare the execution time of each algorithm, an *LDDB* implementation must be chosen for *Block A**. The size on disk of each possible implementation of the *LDDB*, derived from the different

[6] *Caliper* can test to "typically in the sub-microsecond range", whereas an expansion in an any-angle pathfinding algorithm is approximately one microsecond.

	Size on disk for $LDDB_N$ (kB)								
	Standard			Semi-extended			Fully-extended		
Compression type	N=2	N=3	N=4	N=2	N=3	N=4	N=2	N=3	N=4
Uncompressed	27	31	36	1788	2506	3564	401000	677000	1156800
Bitwise	11	13	15	726	1057	1520	156000	275000	482000
Geometrically	6	6	7	248	358	468	51000	90000	143000
Bitwise & geometrically	3	3	3	104	154	203	20,000	37000	59000

Table 4.1: Size on disk (in kilobytes) of different *LDDB* implementations

	Execution time for $LDDB_N$ (ms)		
Compression type	$N = 2$	$N = 3$	$N = 4$
Uncompressed	10.53	9.53	15.79
Bitwise	11.05	9.78	8.97
Geometrically	12.25	10.75	10.28
Bitwise & geometrically	14.38	11.91	11.44

Table 4.2: Execution times (in milliseconds) of different semi-extended *LDDB* implementations

compression and extension options detailed in 3.4.5, are shown in Table 4.1.

Computation of special case blocks

The 'Special case blocks' paragraph of 3.4.5 presents three options for obtaining the edge weights for special case blocks. The first option, which is to compute the edge weights of special case blocks at runtime (and hence the algorithm will only require a standard (i.e. unextended) *LDDB*), increases the time of computation by approximately 50% compared with the other two options. On the other hand, the fully-extended option is unnecessary since in this investigation there is no possibility of the *start* and *goal* being in the same block. Therefore, all further analysis will use a semi-extended LDDB.

Optimum *LDDB* implementation

Table 4.2 shows the median execution time for *Block A** over a fixed set of 400 maps for all of the possible semi-extended implementations of the *LDDB*.

The execution time generally decreases as block size increases, which is contrary to the predictions based on the number of node expansions or neighbour expansions in the previous section. On the other hand, the execution time decrease is not as significant as the prediction based on Yap et al.'s definition of an expansion as a block expansion — this implies that all of these are poor performance predictors. This issue will be discussed in the next section.

It can also be deduced that the *bitwise-encoded* database manages faster execution than the *uncompressed* implementation since its reduced size enables more of it to be stored in higher levels of cache, whereas such an advantage is outweighed in the implementations that use *geometric compression* by the fact that the encoding and decoding is so costly.

However, it should be noted that median execution time does not tell the whole story. For example, Table 4.2 does not communicate that *LDDB*s with larger block sizes take much longer to load into memory every time the program is run because of their large size. Furthermore, large blocks have

	Load time for $LDDB_N$ (s)		
Compression type	$N = 2$	$N = 3$	$N = 4$
Uncompressed	0.130	1.335	334.071
Bitwise	0.012	0.390	115.863
Geometrically	0.011	0.174	42.510
Bitwise & geometrically	0.005	0.061	13.161

Table 4.3: Load times (in seconds) of different semi-extended *LDDB* implementations

	Max. execution time for $LDDB_N$ (ms)		
Compression type	$N = 2$	$N = 3$	$N = 4$
Uncompressed	117.78	98.6	108.4
Bitwise	105.6	93.4	422.4
Geometrically	168.9	115.5	682.0
Bitwise & geometrically	192.6	100.8	93.4

Table 4.4: Maximum execution times (in milliseconds) of different semi-extended *LDDB* implementations

extremely long *maximum* execution times because the larger *LDDB* size requires the *Java Virtual Machine* to initiate larger and more frequent garbage collection disruptions as the free heap space for the rest of the program is smaller — in fact, it is also reasonable to suggest that a reduced free heap space is also the explanation for the high median execution time for the *uncompressed* implementation when the block size is 4×4. Conversely, smaller block sizes require a more dense graph structure which causes larger garbage collection disruptions.

For these reasons, a compromise of the *bitwise-compressed* implementation of block size 3×3 is used for the remainder of this project due to its consistency and low memory footprint.[7]

4.4.3 Pathfinding algorithms

Figure 4.10 shows the execution times of the different pathfinding algorithms. In general, *Block A** computes paths in the fastest time, followed by *Lazy Theta** and then *Theta**, all of which are faster than the classic algorithms — although there is significant overlap between the distributions. These results are the first *independent* verification of the performance benefits of *Block A** claimed by Yap et al., although other papers have quoted the results [**?**, **?**].

Whilst the results confirm some of the broad predictions obtained from using the number of expansions (an architecture-agnostic metric) as a predictor of performance, the correlation is far from exact. This is exemplified by Figure 4.11, which highlights that not all expansions are equal, and confirms the claim made in section 4.3.1, which follow naturally from the graph-theoretic description of the algorithms developed for this dissertation, that defining an expansion in *Block A** as the expansion of a block (as per Yap et al.'s paper) gives unreasonably high predictions of performance, and that *if* expansions are used as a performance predictor then a more consistent definition of an expansion for *Block A** is to use the number of *nodes* expanded.

[7] To avoid frequent, costly garbage collection while creating graph structures for block size of 2×2 and to avoid heap overflows while loading *LDDB*s of block size 4×4, the tests were run with 2GB of heap space. The bitwise-compressed implementation of block size 3×3 does not have such excessive memory requirements.

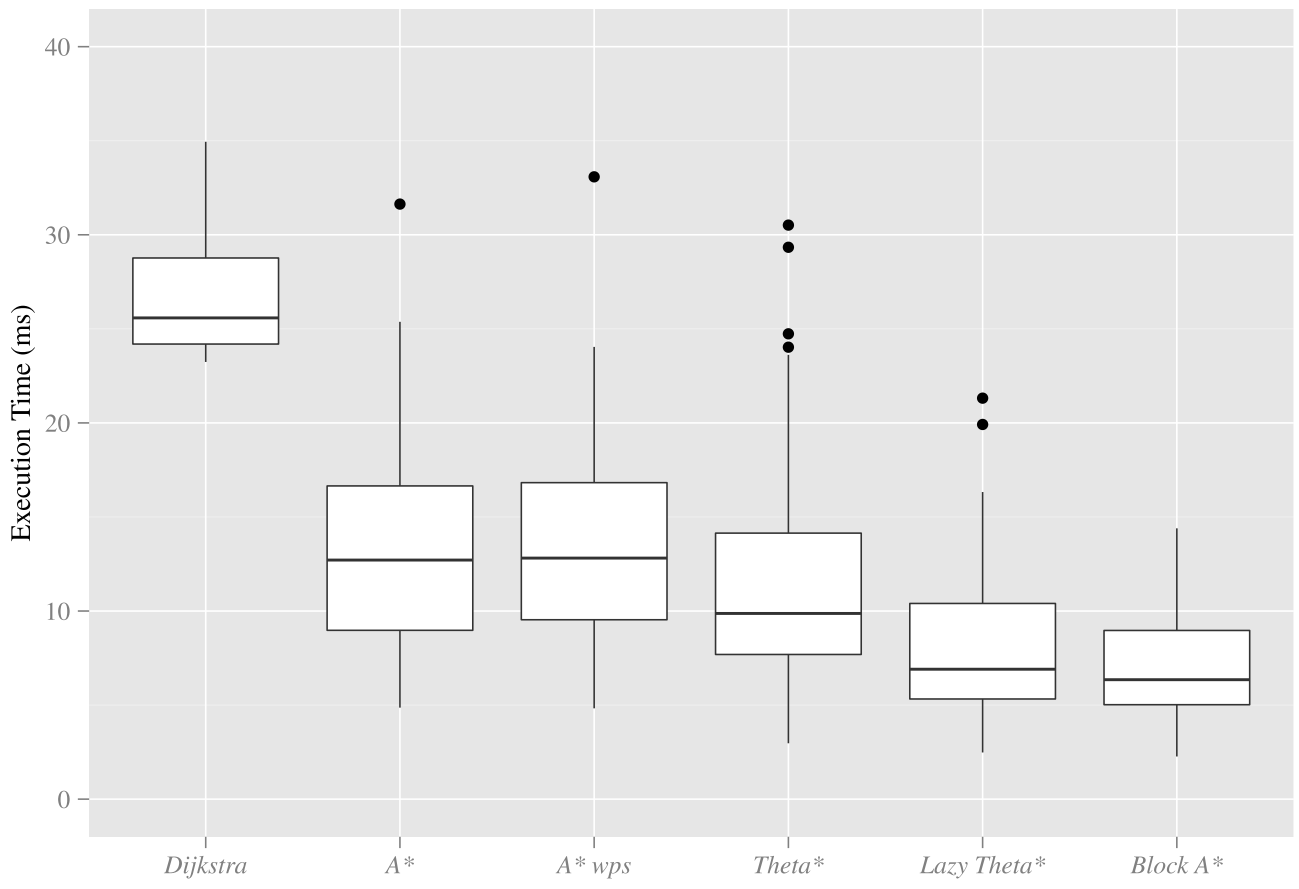


Figure 4.10: Execution time of pathfinding algorithms

The consistent framework developed in the Preparation chapter allows explanation of this Figure 4.11:

1. *Dijkstra* expansions are faster than those of *A** because they do not require the calculation of the *h-value*;

2. *Lazy Theta** expansions are faster than those of *Theta** because they only perform one line of sight test per expansion, whereas *Theta** performs up to eight;

3. an extremely fine-grained benchmarking tool would be required to establish why the expansion of a node in *Block A** is slightly faster than expansions in the other algorithms. However, Figure 4.12 strongly implies that it is due to the shorter[8] and therefore faster[9] priority queue *openSet*.

[8]The main priority queue in *Block A** is shorter than in the other any-angle pathfinding algorithms because each queue element represents a larger portion of the map.

[9]The time required for an update to a priority queue increases as queue length increases [**?**].

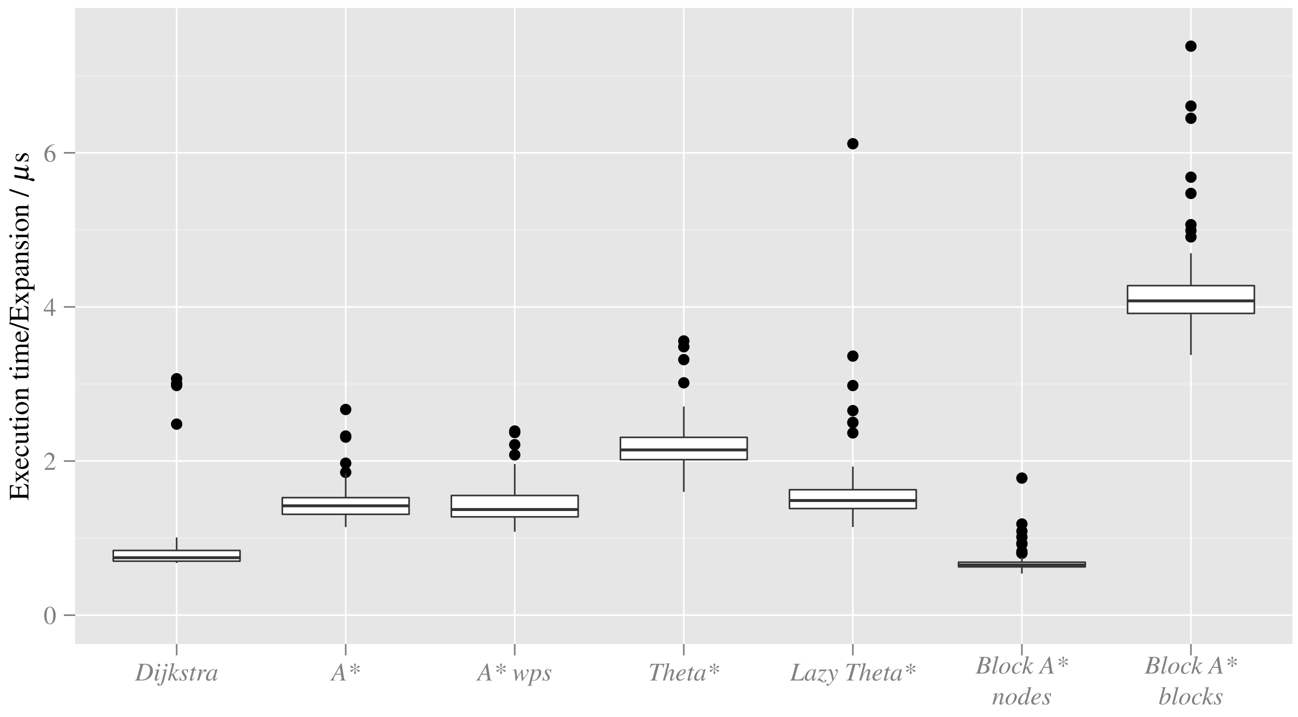


Figure 4.11: Time per expansion — expansions are a poor performance predictor

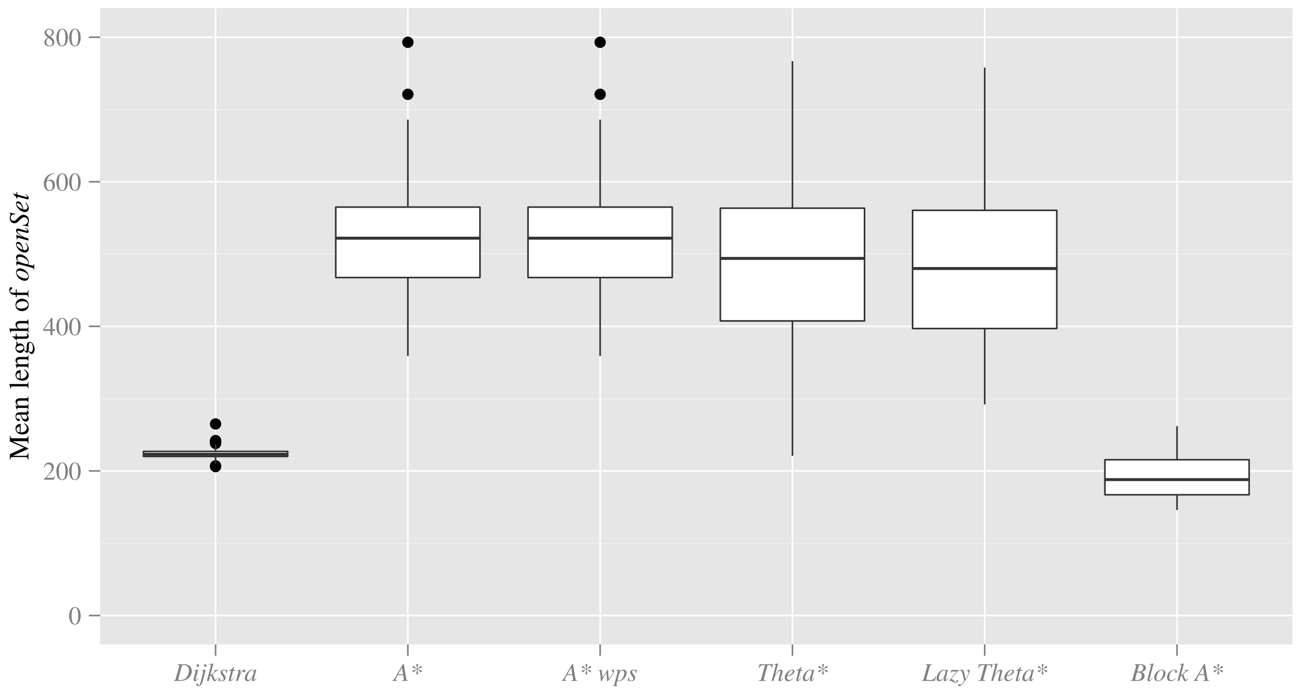


Figure 4.12: Mean priority queue length of pathfinding algorithms

5 | Conclusions

5.1 Summary

This dissertation has introduced a consistent framework with which to explain the most prominent any-angle pathfinding algorithms, along with a detailed discussion of performance-enhancing implementation strategies and a thorough analysis of the implementations using multiple performance metrics, thus achieving the primary goals and two of the extension of the original Project Proposal. This work was motivated by the lack of coherence in the current literature, and the fact that no other paper has attempted to independently verify the benefits of *Block A**.

The Evaluation chapter confirms that:

- not only do any-angle pathfinding algorithms find paths that are shorter and have a lower angle-sum that the classic algorithms, but
- they also manage this in a shorter time.

In particular, *Theta** computes paths that are shortest and have the lowest angle-sum, while *Block A** computes paths in the shortest time.

This dissertation also reveals that the standard practice of using the number of expansions as a predictor of performance can be misleading. This motivates the need for a consistent framework for explaining any-angle pathfinding algorithms — a need which this dissertation attempts to satisfy.

5.2 Further work

The in-depth discussion of the properties, strengths and weaknesses of the algorithms in this dissertation provides guidance towards potential areas of further investigation:

1. the main drawback of *Block A** is its large path-angle sum. However, the analysis of *A* with post-smoothing* shows that a large path-angle sum can be significantly reduced by running a computationally cheap post-processing step;
2. the main drawback of *Theta** is the eight line of sight tests that are required for every node expansion. However, not only are the line of sight tests independent, which may be exploited with parallelisation, but they also have the overlapping subproblem property, which may be exploited with dynamic programming;
3. the relative importance of path length vs. path angle-sum in making a path appear 'humanoid' is an important consideration for computer games, yet is not accounted for in the literature.

A Ancillary explanations of algorithms and concepts

A.1 Discretisation

Grid-based graph: $d_G(L_O, M) \rightarrow G_{M,G}$
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. If none of the (up to) four cells surrounding a node are free then that node is removed,[1] 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: $d_V(L_O, M) \rightarrow G_{M,V}$
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. 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 — since such nodes are guaranteed to not be on the shortest path. 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.

A.2 A* with post-smoothing

This section is in reference to the pseudo-code in Algorithm 3:

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 location represented by n_{curr} to the location represented by $n_{curr}.parent.parent$ (line 6). If a line of sight exists, $n_{curr}.parent$ is set to $n_{curr}.parent.parent$ (line 7) — 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 $n_{curr}.parent.parent$ (line 14, and note that $n_{curr}.parent.parent$ is the node on which the line of sight test failed), and the next iteration commences. When $n = n_{start}$, the algorithm terminates (lines 12-13).

A.3 Theta*

This section is in reference to the pseudo-code in Algorithm 4:

[1]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 $0 \leq a,b \leq N$.

If a line of sight exists between $n_{neigh}.coord$ and $n_{curr}.parent.coord$ (line 1), *Theta** (notionally) creates an edge $e = (n_{neigh}, n_{curr}.parent) \in G$ and attempts to relax e (lines 2-5). However, 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}), as per the `Update` subroutine of $A*$ (lines 10-13).

A.4 Lazy Theta*

This section is in reference to the pseudo-code in Algorithm 5:

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 (i.e. *Lazy Theta** performs the update part of relaxation without first performing the test in equation (2.1), lines 21-24). *Lazy Theta** only actually performs that line of sight test if the n_{neigh} itself is ever expanded (i.e. now the new n_{curr}), by calling `Initialise` when n_{curr} is popped off *openSet*. If the line of sight does not in fact exist, `Initialise` alters n_{curr} accordingly (lines 17-20).

A.5 Line of Sight algorithm

This section is in reference to the pseudo-code in Algorithm 8:

As stated in 3.2.4, the *Line of Sight* algorithm is based on the pseudo-code in the publication of the *Theta** algorithm [**?**],[2] which itself is a derivative of *Bresenham's line drawing algorithm* [**?**].

For the purposes of clarity, the pseudocode presented in Algorithm 8 assumes that the straight line between locations a and b is in 'octant 1' - i.e. the angle that the line ab makes with the horizontal is between 0^o and 45^o.

The key variables of the algorithm are:

i and j — integers that represent the coordinate of the cell being considered. The algorithm only considers cells that the line ab passes through.

s — a value that represents the point at which the line ab intersects $i+1$, with respect to the current j value, i.e. when the algorithm is considering cell (i, j) while doing a line of sight test between location a at (i_a, j_a) and location b at (i_b, j_b), $s = (y(i+1)/j)\Delta i$, where $y(i+1)$ is the y-coordinate of the point at which the line ab intersects $i+1$, and $\Delta i = i_b - i_a$. The s value determines how many cells in column i above the current cell ((i, j) will be checked.

Algorithm - assuming octant 1
The algorithm starts by considering the cell with coordinate a. If every cell in column i_a that the line ab passes through is free, then column i_{a+1} is considered, and so on. If column i_b is reached without any blocked cells detected, the algorithm returns *true*, otherwise it returns *false*.

s determines which cells the algorithm needs to check to find out whether they are blocked:

- $s > \Delta i$ indicates that the line ab intersects $i+1$ somewhere above cell (i, j) — so the algorithm checks whether (i, j) is blocked. If it is then the algorithm returns $false$ but if it isn't then j is increased so that the next cell to be considered is $(i+1, j)$ (see Figure A.1, cell $(0,0)$).

[2] I have re-named the value f that is used in the *Line of Sight* algorithm in the publication of the *Theta** algorithm as s, to avoid confusion with the $f-value$ of a node that is used elsewhere in this dissertation.

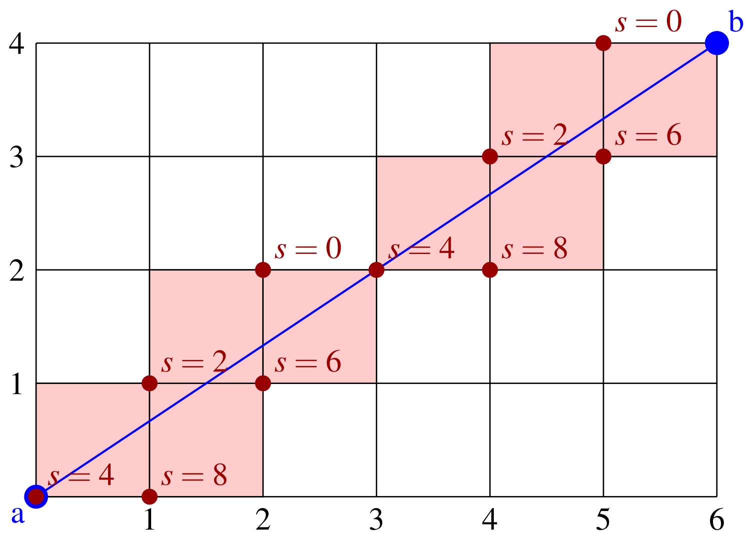


Figure A.1: Cells checked by the Line of Sight algorithm, where $\Delta i = 6$ and $\Delta j = 4$

- $0 < s \leq \Delta i$ indicates that the line ab intersects $i+1$ in the cell currently being considered — so the algorithm checks whether (i,j) is blocked. If it is then the algorithm returns $false$ but if it isn't then i is increased so that the next cell to be considered is $(i+1,j)$ (see Figure A.1, cell $(1,0)$).
- $s = 0 \wedge \Delta j \neq 0$ indicates that the line ab is intersects $i+1$ at exactly $j+1$ — so the algorithm does not need to check whether (i,j) is blocked, because of the zero-width agent assumption made in subsection 2.1.1. Therefore, i and j are increased so that the next cell to be considered is $(i+1, j+1)$ (see Figure A.1, cell $(2,2)$).
- $s = 0 \wedge \Delta j = 0$ indicates that the line ab is horizontal — so the algorithm checks whether (i,j) and $(i, j-1)$ are blocked. If it is then the algorithm returns $false$ but if it isn't then i is increased so that the next cell to be considered is $(i+1,j)$.[3] This case is not illustrated in Figure A.1.

A.6 Query to a bitwise-and-geometrically compressed *LDDB*$_N$

Figure 3.11 gives an annotated explanation of a query to a bitwise-and-geometrically compressed *LDDB* using a block size of 3×3. The steps are elaborated below:

(a) the original query requests the intermediate coordinates on the shortest path between an *ingress* of $(2,0)$ and an *egress* of $(3,3)$ on the map shown, i.e. a map of size 3×3 with integer code 48;

(b) the integer codes are calculated for all four rotations of the map, and the smallest code is used for the query. The *ingress* and *egress* are rotated accordingly;

(c) the $y - value$ of the rotated *ingress* coordinate is larger than that of the rotated *egress* coordinate. Therefore, the *ingress* and *egress* are exchanged (i.e. reflected) for the query;

(d) the final configuration for the query has been reached: map code 18 and $ingress - egress$ coordinates $(3,0) - (0,1)$ which is encoded as $011 - 000 - 000 - 001$;

[3] 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.

(e) the query returns a 64 bit integer, whose 32 most significant bits can be decoded as a `float` giving the length of the shortest path between *ingress* and *egress*, and whose 24 least significant bits are $010-010-001-010$ which is decoded as $i_1 = (2,2)$ and $i_2 = (1,2)$;

(f) since the coordinates were reflected in step (c), the reflection is undone in step (f). Therefore the first intermediate coordinate $i_1 = (1,2)$ and the second $i_2 = (2,2)$;

(g) since the map was rotated 90° clockwise in step (b), the rotation is undone in step (g). Therefore $i_1 = (1,1)$ and $i_2 = (1,2)$;

(h) the result is ready to be returned, which provides *Block A** with the shortest path between the *ingress* and the *egress*, as shown.

B | Project Proposal

Project Proposal

A Comparison of Any-Angle Pathfinding Algorithms for Virtual Agents

O. Freeman, Clare College

25th October 2013

Project Originator: O. Freeman

Project Supervisor: Dr. R. J. Gibbens

Director of Studies: Prof. L. J. Paulson

Project Overseers: Dr. S. Clark & Dr P. Liò

Introduction

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.

The environment must first be discretised into a graph so that it can be searched. Common discretisation strategies include representing the environment as a grid of blocked and unblocked cells, visibility graphs and navigation meshes. Many algorithms exist to efficiently find optimal paths through graphs, but due to its optimality and efficiency, A* is often the best choice *(Nash: 2012)*. However, the paths that A* produces are constrained to travel along the edges of the graph – in a grid graph, for example, the agent is restricted to travelling at integer multiples of 45°. Many discretisation strategies thus produce paths which are longer and have more turns than is necessary, and hence look unnatural to the human eye – a property that is undesirable when mimicking human behaviour with robots or virtual agents.

A variety of A*-inspired 'any-angle' pathfinding algorithms have been proposed to solve this problem. These include:

- *Post-processing smoothing step to A* (Millington: 2009)* – collapses multiple straight line segments using a line-of-sight algorithm.
- *Field D* (Ferguson, Stentz: 2006)* – avoids the constraint of having to travel only to existing nodes by interpolating between g-values to produce new nodes at semi-arbitrary positions.
- *Theta* (Nash, Koenig: 2007)* – allows nodes to change their parent to certain nodes within line-of-sight.
- *Block A* (Yap: 2011)* – performs search on $N \times N$ cell blocks whose internal optimal paths have been pre-calculated.

The aim of this project is to compare and contrast the performance of a selection of any-angle pathfinding algorithms. The test suite will consist of randomly generated maps of differing coverage, clustering and grid resolution to ensure that the analysis of the algorithms' properties is suitable for a wide range of application domains.

Starting Point

My current algorithmic knowledge for this project is:

- *Part 1B CST Artificial Intelligence I* – A* search.
- *Artificial Intelligence for Games (Millington, Funge)* – A* pathfinding.

My current programming experience is:

- *Part 1A CST Programming in Java.*
- *Part 1B CST Further Java* and *Group Project.*
- *Internship at NaturalMotion Inc.* -– Developed a prototype GUI in Java.

Work to be done

In order to test and compare the any-angle pathfinding algorithms, I will need to develop an application to generate maps and to run and record the performance of the algorithms. I will also need to use statistical software to evaluate the performance of the algorithms.

The project breaks down into the following main sections:

Map generation
I will develop a tool that can automatically generate maps with adjustable coverage, clustering and resolution. This may require a custom algorithm. The tool will also convert the maps into a search graph.

Testing simulator
I will develop an application that allows me to run one of a collection of algorithms on one of a collection of maps, and then visually display the resulting path and statistics about the path. Therefore the underlying framework must be modular, to allow different maps and algorithms to be used interchangeably.

Algorithm implementation
I will research and implement the any-angle path-finding algorithms.

Evaluation
I will evaluate the performance of the algorithms according to metrics such as:

Path – Path length and path straightness.
Graph – Clustering, coverage and resolution.
Computation – Execution time.

Success Criteria for the Main Result

The following core components should be produced to declare the project a success:

1. A generator of maps of varying coverage, clustering and resolution.
2. A simulator that can:
   - load one of a collection of maps from the generator;
   - run one of a collection of any-angle path-finding algorithms on the map and display the result visually;
   - return statistics about the performance of the algorithm on the map.

Possible Extensions

1. Extend investigation from grid-based map discretization to visibility graphs.
2. Implement a map editor that allows the user to create a custom map by selecting grid locations to be blocked.
3. Adapt the implementation and analyse the performance of the algorithms when the obstacles become traversable with high cost function – i.e. the agent can climb over the obstacle, but only slowly.

Resources Required and Backup Strategy

For this project I shall mainly use my own Macbook Air (1.3GHz dual-core Intel, 4GB RAM, 128GB Disk), with backup provided on cloud storage and version control provided by GitHub. If my computer fails, I will be able to recover my work from the cloud storage and continue my project on the MCS machines.

Timetable: Workplan and Milestones

Project start date: 28/10/2013.

28th October – 3rd November (1 week)

Project set up and familiarization

- Start project note book.
- Set up programming environment, including testing framework.
- Set up backup strategy.

Milestone: Comfortable with development environment.

4th November – 17th November (2 weeks)

Literature research and implementation planning

- Research and choose any-angle pathfinding algorithms.
- Research and choose data structures and algorithms that will be necessary for the map generator and algorithm simulator applications.
- Decide key interfaces of the map generator and algorithm simulator applications.

Milestone: Any-path algorithms chosen and thoroughly understood. Defining features of application identified.

18th November – 15th December (4 weeks)

Application development phase 1

- Devise architecture and class structure, including any design patterns to be used.
- Implement a prototype simulation application that renders a placeholder map and runs a placeholder algorithm.

Milestone: A working prototype implementation of the simulation application.

16th December – 5th January (3 weeks)

Application development phase 2

- Implement path visualisation in the application.
- Implement statistical measurement and feedback in the application.

Milestone: Application correctly displays path and path statistics.

6th January – 26th January (3 weeks)

Path-finding algorithm implementation

- Implement and test the chosen any-angle path-finding algorithms.

Milestone: Path-finding algorithms run correctly.

27th January – 9th February (2 weeks)

Map-generation algorithm implementation

- Implement and test the map-generation algorithm.
- Test and debug the simulation application.

Milestone: Maps generated according to the input parameters.

10th February – 23rd February (2 weeks)

Progress report

- Write the progress report.
- Prepare the presentation.
- Start gathering data for evaluation.

Milestone: Progress report and presentation complete.

24th February – 9th March (2 weeks)

Catchup and extension

- Learn R - statistical analysis software.
- Complete any unfinished tasks.
- *Optional extension 1*: Visibility graph.
- *Optional extension 2*: Custom maps.
- *Optional extension 3*: Traversable obstacles.

Milestone: Code completion.

10th March – 23rd March (2 weeks)

Dissertation phase 1

- Plan the structure of the Dissertation.
- Improve LaTeX skills.
- Write the Preparation chapter.

- Write the Implementation chapter.

Milestone: Preparation and Implementation chapters complete.

24th March – 13th April (3 weeks)

Dissertation phase 2

- Generate statistics and diagrams.
- Write the Evaluation chapter.

Milestone: Evaluation chapter complete.

14th April – 27th April (2 weeks)

Dissertation phase 3

- Write the Conclusion.
- Write the Bibliography and Index.
- Write the Appendices.
- Review the Dissertation.

Milestone: Draft Dissertation complete.

28th April – 16th May (2.5 weeks)

Dissertation phase 4

- Complete any unfinished tasks.
- Refine the Dissertation.

Milestone: Dissertation complete.