CANDIDATE'S ID NUMBER	C1749777
CANDIDATE'S SURNAME	**Mr** Walker
CANDIDATE'S FULL FORENAMES	Barrie Mckinlay

DECLARATION

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed ..... (candidate) Date 4/5/20

STATEMENT 1

This dissertation is being submitted in partial fulfillment of the requirements for the degree of .....MSc.....(insert MA, MSc, MBA, MScD, LLM etc, as appropriate)

Signed ..... (candidate) Date 4/5/20

STATEMENT 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A Bibliography is appended.

Signed ..... (candidate) Date 4/5/20

STATEMENT 3 – TO BE COMPLETED WHERE THE SECOND COPY OF THE DISSERTATION IS SUBMITTED IN AN APPROVED ELECTRONIC FORMAT

I confirm that the electronic copy is identical to the bound copy of the dissertation electronic only

Signed ..... (candidate) Date .....

STATEMENT 4

I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate) Date 4/5/20

STATEMENT 5 - BAR ON ACCESS APPROVED

I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loans **after expiry of a bar on access approved by the Graduate Development Committee.**

Signed ..... (candidate) Date 4/5/20

MSc Advanced Computer Science

Supervisor: Padraig Corcoran

Student: Barrie Walker

School of Computer Science and Informatics, Cardiff University

4th May 2020

Table of Contents

TABLE OF CONTENTS **2**

1. ABSTRACT **4**

2. INTRODUCTION **4**

3. AIM AND OBJECTIVES **5**

4. BACKGROUND MATERIAL **5**

DIJKSTRA'S ALGORITHM TO FIND THE SHORTEST PATH BETWEEN TWO NODES 5
A* ALGORITHM TO FIND THE SHORTEST PATH BETWEEN TWO NODES 6
PHYSICAL APPROACH 7

5. PROBLEM **8**

6. PROJECT APPROACH **9**

7. APPLICATION OF THE CHOSEN APPROACH **10**

EVOLUTION OF THE ALGORITHM 10
REQUIREMENTS FOR THE INITIAL VIRTUAL POSITION MAPPING FOR A NEW ALGORITHM 12
MAPPING TO VIRTUAL POSITIONS 13
A STRAWMAN FOR THE NEW ALGORITHM 14
A CONCEPTUAL COMPARISON WITH A* 14
ENHANCING THE VIRTUAL MAPPING TO COPE WITH MULTIPLE END NODES 14
MAPPING TO VIRTUAL POSITIONS SUPPORTING MULTIPLE END NODES 15

8. PRODUCTS **18**

THE NEW ALGORITHM 18
THE COMPLEXITY (ORDER) OF THE NEW ALGORITHM 19
A* WRAPPER ALGORITHM FOR MULTIPLE END NODES 20
MEASUREMENT CRITERIA 21
THE TEST APPLICATION 21
High level requirements *21*
User Interface *22*
Application Design *24*
Mapping initial coordinates to virtual positions *24*
Map data *25*
Comparison execution output *26*
Functional testing *26*
GENERATING TEST DATA 26
Comparison results execution 1 ..... *28*
REASON 2 – A* WITH EFFICIENCY WRAPPER PLATEAUS 29
REASON 3 – THE NEW ALGORITHM IS TWICE AS EFFICIENT AS A* 30
Comparison results execution 2 ..... *33*
REASON 2 REVIEW 34
REASON 3 REVIEW 35
REASON 4 - THE NEW ALGORITHM USES LESS HITS THE MORE END NODES ARE SELECTED 36

10. CONCLUSIONS **38**

11. AREAS FOR FURTHER RESEARCH AND DEVELOPMENT **39**

ADD CAPABILITY FOR PRIORITY QUEUE 39
FINDING THE SHORTEST PATH FROM MULTIPLE START NODES TO A SINGLE END NODE 39
FINDING THE SHORTEST PATH FROM MULTIPLE START NODES TO MULTIPLE END NODES 40
FINDING THE SHORTEST PATH THROUGH MULTIPLE CLUSTERS 40
REVIEW THE SCORING METHOD FOR EFFICIENCY COMPARISON 41
INCORPORATION OF REAL-WORLD TEST DATA 41
TESTING THE REGULARITY OF THE MAP 41
CLUSTER ANALYSIS 41
SHORTEST PATH IN 3 DIMENSIONS 41
MATRIX TRANSFORMATIONS 41
TRAVELLING SALESMAN PROBLEM 41
REVIEWING THE COMPLEXITY (ORDER) OF THE NEW ALGORITHM 41

12. REFLECTION/LEARNING 42

THE IMPORTANCE OF MODELLING 42
TENACITY 42
AVOID MAKING ASSUMPTIONS 42
THE IMPORTANCE OF PROTOTYPING AND UNIT TESTING 42
ANALYSIS OF OUTPUT 42
VISUALISATION OF THE TEST DATA 42
FLEXIBILITY 43

13. REFERENCES 44

14. APPENDICES 45

SHORTEST STRING PATH PROOF 45
DEFINITION OF SLACK 46
KEY SOURCE CODE 48
Mapping to the virtual line 48
The new algorithm 51
A with wrapper 55*
Creation of test data 57

1. Abstract

This project presents a new algorithm to determine the shortest path from a single start node in a geographical graph (e.g. a road network) to one or more end nodes within the graph.

Development of the algorithm begins by looking at a way of solving the shortest path problem using a physical model made up of strings to represent edges and knots to represent nodes. This concept is then developed into a new algorithm to initially find the shortest path between a single start node to a single end node and then progressing to find the shortest path from a single node to multiple end nodes.

Along with the development of the algorithm, this report also covers the development of a flexible test application with a user interface and the production of suitable test data to test the quality and performance of the new algorithm against other existing algorithms.

Using output from the test application, this project then compares the test output from the new algorithm and the A* algorithm which is an algorithm commonly used to solve this problem. The results of this comparison show a significant performance improvement to find the shortest path from a single node to multiple end nodes.

The efficiency for the new algorithm is shown to remain constant regardless of the number of end nodes being searched, whereas the complexity for A* grows linearly as might intuitively be expected. This consistency in the complexity is important in the context of processing large road networks.

In the areas for further research section, a design is shown to further enhance the new algorithm to find the shortest path between multiple start nodes and multiple end nodes and this would be expected to show an even greater performance improvement.

In conclusion this project successfully demonstrates a new algorithm which outperforms the comparison algorithms when searching for the shortest path from a single start node to multiple end nodes.

2. Introduction

The idea for this project came whilst attending a lecture on NP-Hard problems and the consideration as to whether any of these could be solved by physical constructs and subsequently whether these physical constructs could be modelled efficiently as a computer algorithm. The shortest path problem involves finding the shortest path between a start point and an end point, for example finding the shortest path between Cardiff and Birmingham on a road map of the UK. Although this is not an NP-Hard problem, the shortest path problem does have a physical means of solving it by using strings to represent roads and knots to represent junctions and places. By pulling the knots representing Cardiff and Birmingham apart, when the string becomes taut, the taut strings will represent the shortest path as one or more roads (providing the strings remain untangled). When doing this physically, the properties of nature will ensure that all potential paths are tested at the same time, something that today's computers would require a processor to run in parallel for each potential path which in a large graph would soon become infeasible due to the number of processors that would be required. This inspired an attempt to model the knot and string method efficiently in a computer algorithm and program. A huge amount of research has been done into shortest path problem solving and it has many real world applications. This project has attempted to find a new way of tackling the problem. This paper presents the development of an algorithm for this purpose from initial idea to a working version and shows how this is comparable to existing algorithms but has a significant advantage when expanding the problem to find the shortest path from one start point to any one of multiple end points.

3. Aim and Objectives

The objective of this project is to test the viability of a new shortest path algorithm to find the shortest path between a start point and one or more end points on a geographical map. The testing will demonstrate both the accuracy of the new algorithm and its indicative performance against an existing and well used shortest path algorithm (A*).

4. Background Material

Algorithms already exist to find the shortest path between a start node and a single end node in geographical maps with the most notable being Djikstra and A*. Both of these algorithms calculate the shortest path by working through the nodes in a graph but use a different methodology to identify the next node to process. A* is closest to the new algorithm and uses a heuristic calculation to prioritise node selection which calculates the shortest possible distance from visited nodes to the end node. A lot research to date has focussed on different methods to prioritise the sequence in which nodes are processed and ways to speed shortest path algorithms by performing pre-calculations.

Dijkstra's algorithm to find the shortest path between two nodes

Dijkstra's algorithm (Dijkstra 1959) works by keeping track of a tentative distance from the start node to all other nodes. Initially the start node's tentative distance is set to 0 and all other nodes are set to infinity and marked as unvisited. The algorithm will select the unvisited node with the smallest tentative distance to process. When processing a node, each connected node will be checked to see if it can be reached with a distance less than that node's current tentative distance. Once all nodes connected to the node being processed have been checked then the node being processed is marked as visited. This approach means that nodes are generally processed outwards from the start node in all directions until the end node is found as in Figure 3.1. This is inefficient for finding the shortest path between two nodes due to the number of nodes that need to be processed and can lead to very long processing times in large graphs.

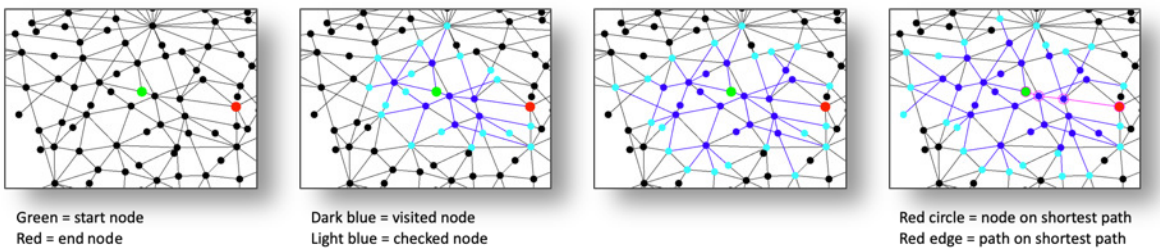


Figure 3.1. Nodes visited and checked during Dijkstra

A* algorithm to find the shortest path between two nodes

A* (Hart et al. 1968) uses a similar approach to Dijkstra but uses a heuristic function to select the next node to process. One widely known heuristic function is to calculate the Euclidian distance from a neighbour node being processed to the end node. The next node to process will always be the one which has the shortest overall distance from the start node to the end node. The overall distance is calculated as the actual distance to the neighbour node as this is known plus the Euclidian (straight line distance) from the neighbour node to the end node which can be calculated from their known coordinate positions. This has a big advantage over standard Dijkstra as the search is not carried out in all directions and once the end node has been found any paths where the heuristic distance is greater than the shortest path found can be ignored.

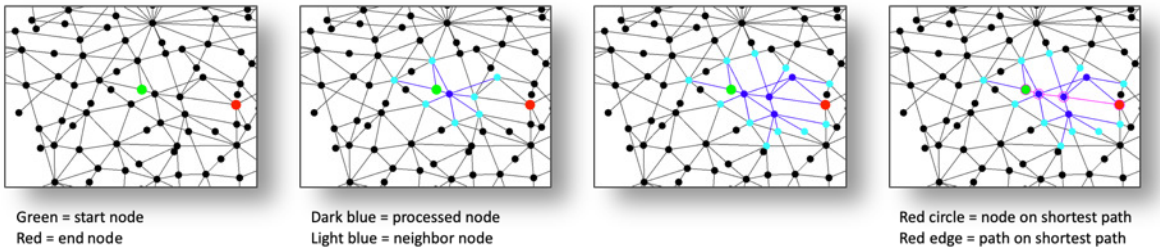


Figure 3.2. Processed and neighbour nodes in A* algorithm

Figure 3.1 and Figure 3.2 show that Dijkstra examines many more nodes that A* to find the shortest path in this example.

Physical approach

Schrijver (Schrijver 2012) discusses the history of the shortest path problem and references Minty's (Minty 1957) 'analog' computer for the shortest path problem using knots to represent nodes and the length of string imbetween knots to represent an edge and its distance (see Figure 3.3). An analog (or physical approach) such as this is not useful in practice as in large graphs it would be very time consuming and difficult to construct accurately. However, if this approach could be efficiently modelled then it could offer an alternative method to solving the shortest path problem. The new method may then give an advantage over existing algorithms such as faster processing time to find the shortest path to a single node or the ability to find the shortest path to one of multiple end nodes (i.e. a cluster of end nodes).

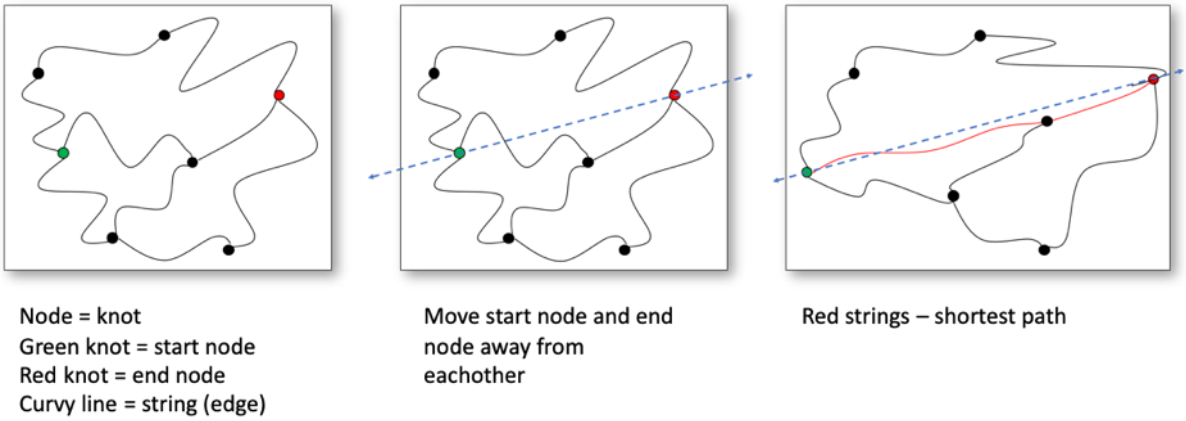


Figure 3.3. String method

Figure 3.3 illustrates how the string method works on a smaller number of knots and strings. As the start knot (green node) and end knot (red node) are moved in a straight line away (a translation of the node along the blue dotted line) from each other the shortest string path (edges highlighted in red) become taut. As other strings connecting two nodes become taut, they are translated in the direction of the moving node whilst ensuring that their Euclidean distance does not exceed the distance of the edge (i.e. the length of the string). It should be noted that the start position of the strings and the knots is unimportant and will not affect the outcome.

5. Problem

This project aims to find a new solution to the following problem by attempting to model the physical approach outlined by Minty.

Without pre-computing, find the shortest path from a single start node to a single end node or the shortest path to the nearest node in a cluster of end nodes within a geographical map. The geographical map will be represented by nodes where each node has a 2-dimensional position and edges where each edge connects two nodes and has a distance that is greater than or equal to the Euclidean distance between the connected nodes.

The problem of finding the shortest path from a single start node to a single end node is met by Dijkstra's algorithm and A* which are referred to in the Background section. A real-world example of this problem would be to find the shortest travel distance from Cardiff to Birmingham.

The problem of finding the shortest path from a single node to the nearest node in a cluster of end nodes can be met by calling Dijkstra's algorithm or A* iteratively for each node in the cluster. This paper includes a wrapper algorithm for A* to improve its efficiency for this problem. Some real-world examples of this problem are:

- From Barry, find the route to any car park in Cardiff
- From junction 35 of the M4 find the route to the nearest electric car charging point in Swansea
- In a distributed computer system, find the nearest server to access a target data set that has been replicated over servers.

Shortest path problems are simple to understand but can be computationally expensive in large graphs.

6. Project Approach

The following development stages were used in the execution of this project:

1. Analysis and Design
    a. Draft an algorithm based on the physical knot / string approach to find a solution for the shortest path problem to find the shortest path form a start node to a single end node
    b. Model the algorithm in Excel
    c. Define measurement criteria to compare the efficiency of each algorithm
2. Build and test
    a. Design and build an application in Java to execute the algorithms against test data for the Dijkstra, A* and a prototype of the new algorithm. The application should be able to consume multiple test graphs in held in json format
    b. System Test and refine the new shortest path algorithm
3. Enhance
    a. Enhance the new shortest path algorithm to cope with multiple (a cluster of) end nodes
    b. Design a wrapper algorithm to enhance A* to improve its effectiveness for multiple (a cluster of) end nodes, to be called 'A* wrapper algorithm'
4. Comparison test and report
    a. Generate test data and a test comparison function to assess each algorithm against the measurement criteria
    b. Produce a report "Algorithm Comparison Report" to determine the effectiveness of the new algorithm
5. Review and analyse the "Algorithm Comparison Report"

7. Application of the chosen approach

Evolution of the algorithm

String method observations

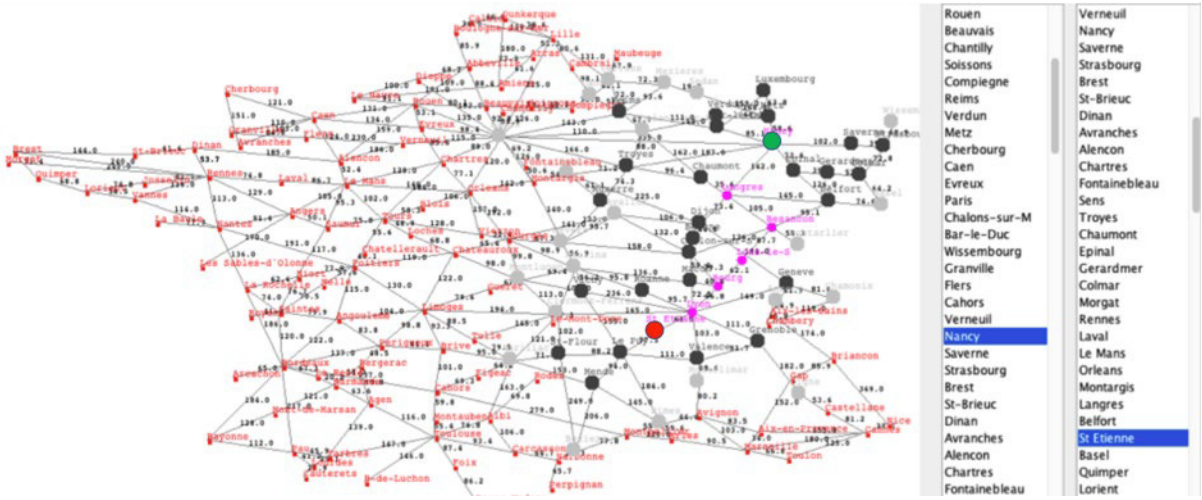
The start point in developing the new algorithm was to consider the string method and how this works. This was modelled in an excel prototype. The prototype introduced the concept of slack in the string. As the start and end knot are moved away from each other the slack in the string connecting the two knots reduces until eventually there is no slack and the string becomes taut. The shortest string path (including any knot 'nodes') will then lie on a straight line joining the start and end knot (see shortest string path proof in appendix). Any knots on the taut string and the strings connecting them resting on the straight line is the shortest path and the solution to the problem.

This work led to the following observations with respect to the string method:

1. A straight line drawn through the start and end node will be the same length as the solution path after the string method has been executed (this follows the same logic as the shortest string path proof in the appendix). In the new algorithm the straight line passing through the start and end node is referred to as the **virtual line**
2. When a knot moves in a direction that is away from a connected knot it will eventually lead to no slack between it and a connected knot after which the connected knot will be pulled in the same direction as the moving knot due to physical forces. This will lead to its position changing. In the new algorithm, its new position is referred to as its **virtual position**
3. The slack in a string is the difference between the actual string length and the Euclidean distance between the knots (see the definition of slack in the appendix). In the new algorithm, the slack is calculated using the virtual positions and the length of the string and is referred to as the **virtual slack**
4. When the start and end node are connected by one or more strings with no virtual slack then the shortest path has been found. This follows from the previous observations and supporting material.
5. The start position of the knots and the strings does not affect the ability of the string method to find a solution. Providing any tangles during execution are removed, the knots and strings can be moved to any position (within the constraints of their string length) at initiation (such as scrunched into a ball), and the method will still produce the correct answer (see the definition of slack in the appendix which defines this for two nodes on the virtual line)

Observations 1 to 4 are good candidates for a new algorithm to solve the problem, however observation 5 highlights a key area of focus. If an extreme example is considered where all of the knots including the start and end knot are initially on a single point, the string method will still work as physically when the end knot is moved away from the start knot the position of other knots will adjust instantly as per observations 1 to 4. However, it will be computationally expensive and time consuming to model all of the knots in a large graph and this would produce a similar performance and progress to Dijkstra.

To understand better the performance of the above approach, a prototype was built in Java that moved the start node away from the end node by a small amount iteratively. This then checked nodes where slack became zero and then moved those nodes. As expected, this was inefficient due to the number of calculations required when the node was moved by a short distance and the number of nodes hit. A second iteration of this prototype applied the method from both the start and the end node and checked to see where the nodes touched. Although this reduced the number of nodes being checked, again the frequent moving of the nodes by small increments led to many potentially unnecessary loops and when examining the output visually it appeared that some nodes were being processed which may not be necessary (see Figure 6.1, Early prototype example).



Key
Green node = start node
Dark grey nodes = moved nodes
Light grey nodes = nodes checked
Pink nodes = shortest path found

Figure 6.1. Early prototype output

Taking the above into account, the focus moved to the identification of which edge to process next by prioritising them as a way to improve the efficiency of the algorithm. Given the optimal shortest path is identified as a path with zero slack, therefore if a path exists from the start node to the end node with no slack, then this will be the shortest path. Continuing this thought process, then the longest path would be the one with the most slack. In conclusion, the path with the least slack will be the shortest path and the path with the most slack will be the longest path. Therefore, at each iteration of the algorithm, choosing an edge to process with the least slack should result in the shortest path being found eventually. This reasoning was key to development of the new algorithm (along with Dijkstra and A* this fits the definition of a greedy algorithm). However, the slack in an edge is dependent on the position of the two nodes at the end of the edge and it has been recognised earlier that the initial position of the knots in the string method would not affect the outcome. Consequently, the success of the algorithm depends on whether nodes can be positioned in such a way as to enable the slack to be used to efficiently prioritise the search for the next edge to process.

Requirements for the initial Virtual Position mapping for a new algorithm

The requirements should apply the observations from the string method to a new algorithm that can be used to find the shortest path in a geographical map with nodes (rather than knots) and edges connecting the nodes (rather than strings). The new algorithm will have the requirements below (note that a path is defined as a sequence of connected nodes where each connected node is joined by an edge).

1. Unlike a physical method such as a string method, computers work in a linear fashion and therefore the new algorithm must have a means to efficiently choose a single edge to process next
2. The shortest path will be the path with the least slack so an algorithm should look for edges with the least slack to process next
3. The virtual distance between two connected nodes must never exceed the edge distance connecting them
4. The initial placement of the nodes will affect the slack for each edge. In the geographical map the default start position for any node will be its geographical coordinates. However, since a path along the virtual line will be the shortest path, the requirement for the new algorithm is for any edge along the virtual line to have no slack. Similarly, edges that move away from the virtual line should have greater slack than those that move closer to it.

Mapping to virtual positions

Taking the above requirements into account, below is a method for mapping nodes to virtual positions which has some advantages and disadvantages

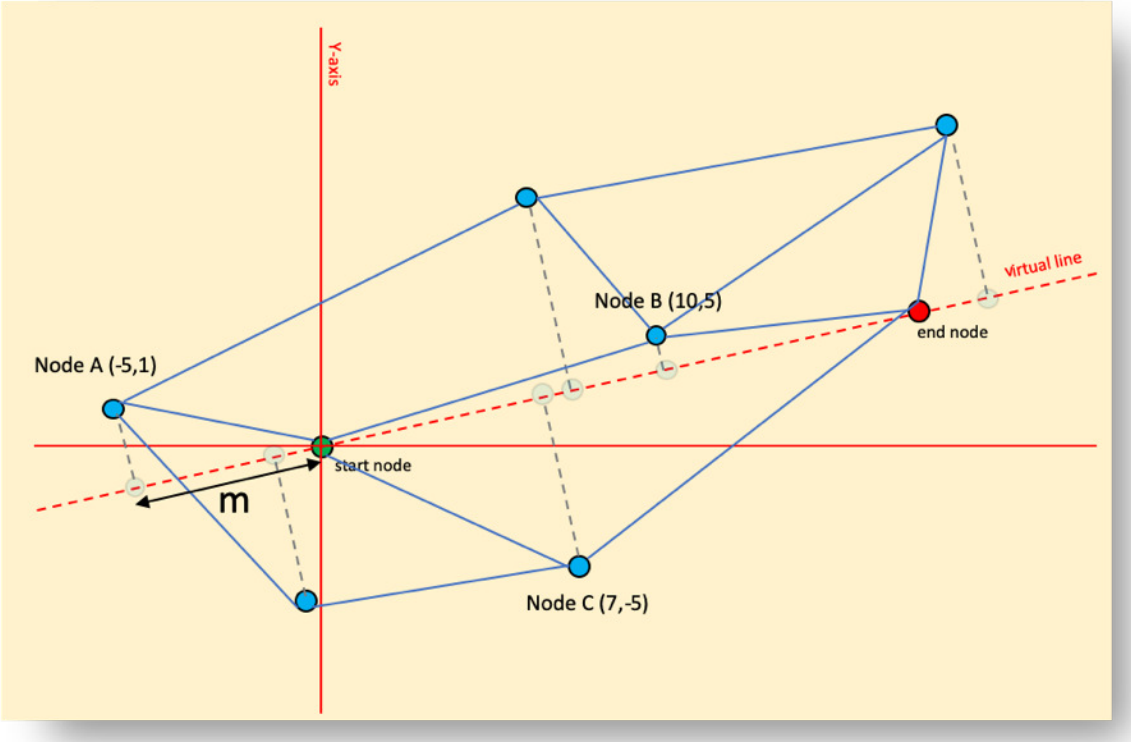


Figure 6.1. Illustration of mapping to virtual positions

1. Let vl be a virtual line passing through the start node and the end node
2. Let vl be considered the x-axis for a new virtual graph with the start node at point (0,0)
3. For each node, draw a line, pn, from the node to the virtual line that is perpendicular to the virtual line
4. Let the virtual position of the node be the point, vp, where pn intersects vl

In Figure 6.1, the distance m will be the (negative) virtual position of node A on vl. This method proved the approach in Excel modelling (see Figure 6.2) and allowed for initial development of an algorithm that could be developed and tested against Dijkstra and A*.

Edges			Step 10b				Step 20b														
From	To	Real Distance	AD	Slack	NMA		AD	Slack	NMA		AD	Slack	NMA		AD	Slack	NMA		Slack	Slack	NMA
C	B	6	4	10		10	-6	0		1	-5	1		1	-6	0		2	-8	-2	
C	F	35	-14	21	LMN		-24	11			-24	11			-25	10			-27	8	
C	A	8	7	15	C		-3	5			-3	5			-4	4			-6	2	
B	D	10			NN		-7	3			-8	2			-8	2			-8	2	
B	E	16					-13	3			-14	2			-14	2			-14	2	
B	G	27					-25	2			-26	1			-26	1			-26	1	
B	C	6					6	12			5	11			6	12			8	14	
B	A	4					3	7			2	6			2	6			2	6	
G	B	27													26	53			26	53	
G	E	12													12	24			12	24	
G	I	12													-11	1			-11	1	
G	F	31													7	38			7	38	
K	C	40																			
K	I	4																			

Figure 6.2. Example of Excel model

A disadvantage of this mapping is that all edges that are parallel to vl will have zero slack. In many cases this is not the optimum route and there is potential to improve it (covered later in this section).

A strawman for the new algorithm

The strawman below is an overview of the approach which was subsequently tested to demonstrate that it works. The full algorithm (included later) includes a number of improvements and efficiency steps.

1. Map all nodes to a virtual position on vl
2. Choose the edge with the least slack and let ls be the least slack value
3. Move the start node away from the end node along vl a distance of ls
4. Recalculate slack for each edge connected to a node that has moved
5. For any slack that is negative, move the connected node so the edge is no longer negative and store the number of moves made for each node
6. If the end node is moved, store the number of moves and continue the current iteration to see if there is a smaller number of moves for any other path
7. Repeat steps 4 to 6 until no slack is negative

A conceptual comparison with A*

- Both algorithms operate iteratively
- A* prioritises nodes to process whereas the new algorithm prioritises edges to process
- Both algorithms use a function to prioritise
- The mapping to virtual positions in the new algorithm affects the output of the priority function to determine the next edge to process
- Both algorithms tend to prioritise in the direction of the end node, however A* focuses on a specific point so the direction of selection can change whereas the new algorithm will continue to prioritise in the direction from the start node to the end node until a shortest path is (or isn't) found

Enhancing the virtual mapping to cope with multiple end nodes

The following additional requirements are necessary for handling multiple end nodes:

1. Slack should be 0 for any line passing through the start node and any end node
2. For nodes in close proximity, their virtual positions should also be in close proximity

Mapping to virtual positions supporting multiple end nodes

The initial approach for mapping nodes to virtual positions was based on the earlier version. However, to ensure that slack is zero for paths in straight lines from the start node to any end node, then any node within the boundaries of the uppermost end node and bottom most end node should maintain its Euclidean distance. This creates upper and lower boundary lines (see Figure 6.3). The virtual line is considered to bisect these boundary lines. Nodes outside of the boundaries are then mapped to a boundary line using the perpendicular intersection as before.

When modelled using Excel, this approach led to some anomalies whereby nodes that were close together geographically were moved further apart when mapping to virtual positions. This effect is worse as the angle between the boundary line increases and can lead to the virtual distance exceeding the edge distance which will break the approach (and in the real-world string example, would break the string). Therefore, a mapping is required that proportionally maps to a virtual position based on the nodes relative position between the boundary and the y-axis.

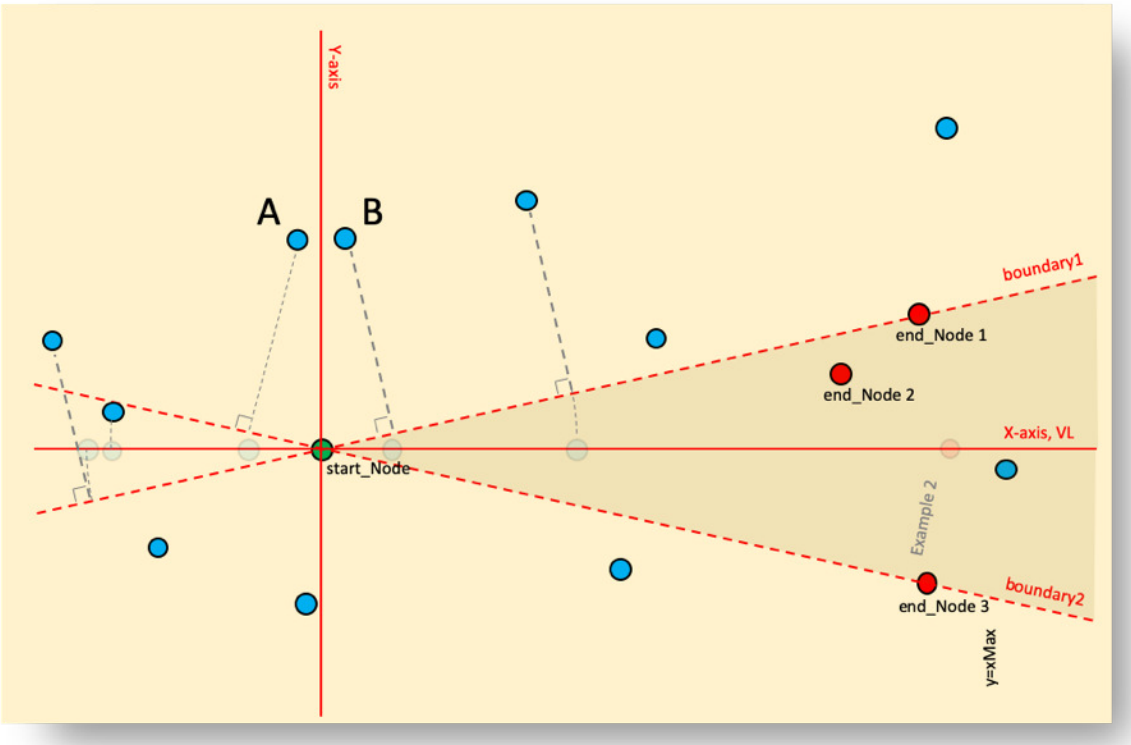


Figure 6.3. Illustration of initial mapping to support multiple end nodes

The mapping in Figure 6.4 caters for multiple end nodes using a ratio to calculate the virtual position based on the position of the node relative to the boundary line and y-axis. As found previously, this affects the calculation of slack and therefore the prioritisation of edges in the new algorithm without needing to change the algorithm. For simplicity, it is assumed that all of the end nodes are contained within a segment that is less than π radians.

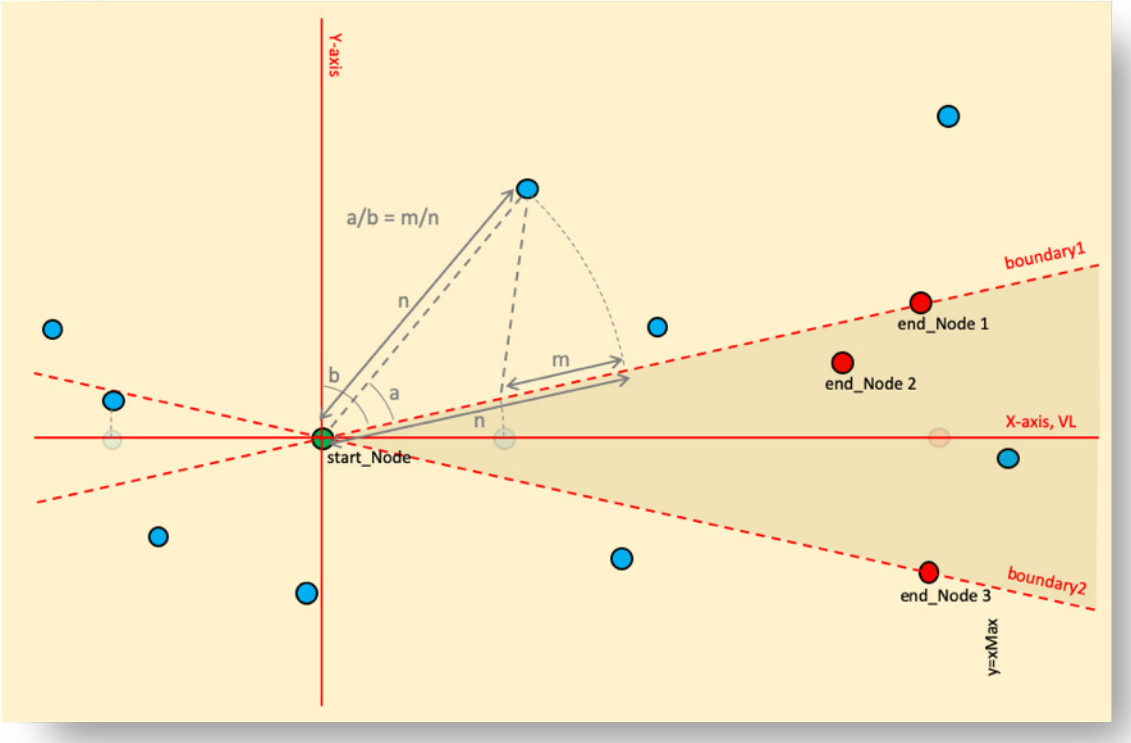


Figure 6.4. Illustration of mapping to support multiple end nodes (following translation and rotation)

The steps below reflect the mapping in Figure 6.4 (the illustration is post step 4)

1) Map two lines, boundary1 and boundary2, such that they pass through the start node and that all of the end nodes sit either on the boundary line or imbetween boundary1 and boundary 2 and the angle at the origin between boundary1 and boundary2 is < π radians
2) Let vl be the line that bisects boundary1 and boundary2
3) Translate the start node to coordinate (0,0) and translate all other nodes using the same translation
4) Rotate vl and all nodes so that vl sits on the x-axis and all end nodes have x >= 0
5) For each node within and on boundary1 and boundary2 (the shaded area in Figure 6.4) set virtual position, vp, to be the Euclidean distance from the start node
6) For all other nodes where x>=0 and y>0, use the relationship a/b = m/n to find m as follows and then set the virtual position, vp, to be n – m
   a) Let nl be the line passing through the origin and the node
   b) Let a be the angle between boundary1 and nl
   c) Let b be the angle between boundary1 and the y-axis
   d) Let n be the Euclidean distance between the start node and the node
7) For all other nodes apply similar rules as per bullet 6.

In addition to coping with multiple end nodes, the above approach also better handles edges that are parallel to the optimum solution but in a non-optimum position.

This approach was validated using Excel worksheets with a visual representation on an X Y scatter graph. Figure 6.5 shows the progression from coordinates initially following the translation of the start node to the origin, to a rotation aligning the virtual line to the x-axis and finally to virtual positions along the x-axis. End nodes are highlighted in red.

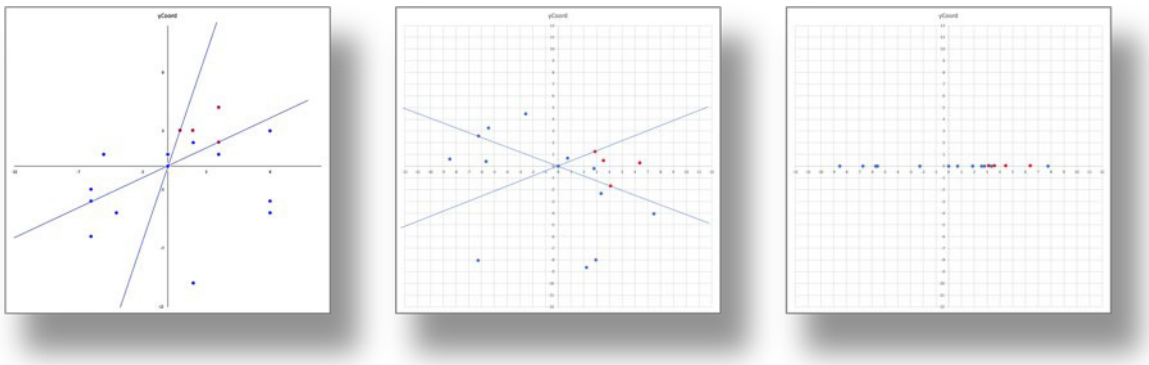


Figure 6.5. Illustration of approach for virtual mapping

8. Products

The new algorithm

```
// Walkers algorithm finds the shortest path from a single start node to one or more
// end nodes
function Walkers (start_node, {end_node1, end_node2…}, {node1, node2…}, {edge1, edge2…})

    // The nodes are mapped to virtual positions using function mapNodes
    nodes := {node1, node2…}
    edges := {edge1, edge2…}
    mappedNodes := mapNodes[nodes]
    // The hit nodes set contains a list of nodes that have been moved
    hitNodes := {start_node}
    startNode := startNode
    endNodes := {end_node1, end_node2…}

    nextEdge := findNextEdge[]

    // The algorithm will continue execution until no edge is returned from the
    // findNextEdge function
    while nextEdge is not null

        // Check whether the moves already made for the node at the start of
        // the edge and moves required (i.e. the slack) to remove slack from the edge
        // being processed are less the moves made for the node at the end of the
        // edge.  If so, this is a valid move.
        if moves[edgeStartNode] + slack[nextEdge] < moves[edgeEndNode] or
        moves[edgeEndNode] is null

            // Set the moves at the edge end node to the new lower figure
            moves[edgeEndNode] = moves[edgeStartNode] + slack[nextEdge]

            // Since we are virtually moving the start node, make the edge
            // start node active
            for each edge connected to edgeStartNode
                active[edgeStartNode] = true
                visited[edgeStartNode] = true

            // If the edge end node is a desired end node then set the
            // shortest move amount, otherwise add the edge end node
            // to the set of hit nodes.  The edge start node could be saved
            // at this point to track the shortest path.
            if edgeEndNode is an end node
                shortestMovesToEndNode = moves[edgeEndNode]
            else
                // Add the edge end node to the set of moved nodes
                add edgeEndNode to hitNodes

        else
            // If this is not a valid move, make the edge inactive
            active[nextEdge] = false

        nextEdge := findNextEdge[]

function findNextEdge ()
```

```
// Check each moved node.  Further research is required to enable a priority queue
// to be used within this function
for each currentNode in hitNodes

        visited(currentNode) := true
        leastMovesAndSlack := infinity
        nextEdge := null

        // For each connected edge recalculate the total move distance required
        // as the moves already made plus the slack for this edge
        for each currentEdge connected to currentNode where active[currentEdge] = true

                movesAndSlack := moves[currentNode] + slack

                if movesAndSlack >= shortestMovesToEndNode
                        active(currentEdge) = false

                if movesAndSlack < leastMovesAndSlack
                        nextEdge := currentEdge

        if nextEdge is null
                // if no edge is found the node can be removed from further searches
                remove currentNode from hitNodes
        else
                // Set the edge to inactive so it isn't searched again. Note, it could
                // subsequently made active again
                active[nextEdge] := false
```

The Complexity (Order) of the new algorithm

The time complexity of the new algorithm depends on the ability of the next edge function to find the next edge on the shortest path. However, this is similar to A* in that it uses a heuristic to select the next edge. In the case of A* the heuristic is based on prioritising nodes based on the lowest Euclidean distance to the end node whereas in the new algorithm the heuristic is based on prioritising edges with the least slack. Since each edge starts and ends with a node, the number of edges will depend on the number of nodes and the number of edges connected to the nodes. The branching factor is the number of edges that need to be tested at each iteration. The lower the branching factor (and therefore the better the edge selection) the more efficient the algorithm.

The order of the new algorithm for a solution is calculated as follows:

Let n = the number of nodes in the graph
Let e = the number of edges in the graph
Let b = the branching factor (the number of edges connected to a node)
Let d = the depth of the search required to find the end node where a depth of 1 is the edges and nodes connected to the start node and a depth of 2 is all of the edges and nodes connected to those and so on.
Let t = the total number of edges and nodes checked at each iteration
Let T = the total number of edges and nodes checked to find the shortest path to the end node

To calculate the average branching factor for a graph, since each edge connects to 2 nodes, then b = (e*2)/n.

Iteration 1: t = 1 + b
Iteration 2: t = b + (b*b)

Iteration 3: $t = (b*b) + (b*b*b)$
Iteration i: $t = b^{(d-1)} + b^d$

Therefore $T = 1 + b + b + b^2 + b^2 + b^3 + b^{(d-1)} + b^d$
$= 1 + 2b + 2b^2 + 2b^3 + b^d$

Taking the most significant value, the order of the new algorithm is therefore $O(b^d)$ where b is the branching factor and i is the number of iterations required to find the shortest path. This is normally written as $O(x^n)$ where x is the branching factor and d is the depth to the end node.

However, this assumes that at each iteration each connected node and edge needs to be processed, however, in the new algorithm the concept of slack I sused to reduce the number of connected edges to be processed. To better understand the Order of complexity, this will also need to take into account the impact of the new algorithm's prioritisation logic on the branching number for each iteration. This is included within this report as an area for further research.

A* wrapper algorithm for multiple end nodes

To test the efficiency of the new algorithm it will be tested against A*. The simplest method would be to execute A* to find the shortest path from the start node to each end node. However, with a large number of end nodes this is likely to be inefficient. To provide a better comparison a wrapper algorithm for A* has been developed to improve the efficiency and remove any unnecessary calculations.

Consequently, this wrapper algorithm adapts A* to more efficiently find the shortest path between a single start node and multiple end nodes. The approach is to first calculate the Euclidean distance from the start node to each end node; sort these into ascending order and then perform an A* search on each starting with the closest end node. After processing each iteration, the shortest path found so far should be stored. As soon as the Euclidean distance to an end node exceeds the least path distance found so far then the remaining end nodes can be discarded. This algorithm depends on the presumption that it is more likely that the shortest path will be to an end node closer to the start node.

```
// Function to wrap A* to improve its efficiency by discarding end nodes when it becomes
// impossible for them to hold the shortest path
function AStarWrapper (start_node, {end_node1, end_node2…}, {node1, node2…}, {edge1,
edge2…})

        nodes := {node1, node2…}
        edges := {edge1, edge2…}
        startNode := startNode
        endNodes := {end_node1, end_node2…}

        // Calculate the Euclidean distance from the start node to each end node
        for each endNode in endNodes
                euclidDist[endNode] := dist[startNode, endNode]

        // Sort the end nodes by Euclidean distance so the closest can be processed first
        sort endNodes by euclidDist into sortedEndNodes

        leastPathDistance := infinity

        // Process the end nodes in Euclidean distance order
        for each endNode in sortedEndNodes
```

```
// If the Euclidean distance for this end node is greater than the
// least path distance already found, then it is not possible for this
// end node to be a sorter path than the one already found.  Since
// the end nodes are sorted in Euclidean distance the algorithm can
// now stop
if euclidDist[endNode] > leastPathDistance
        stop

// Calculate the shortest path using A*
aStarDist := aStar (startNode, endNode)

// Check if the shortest path distance is less than the currently
// held least path distance.  The end node and the actual path
// should also be stored here if required.
if aStarDist < leastPathDistance
        leastPathDistance := aStarDist
```

Measurement criteria

There are two key measurement criteria to test the effectiveness of the new algorithm:

1. Accuracy
2. Efficiency

Accuracy

A* will be used to test the accuracy of the new algorithm. Multiple test data will be executed against both A* and the new algorithm. The shortest path distance from the start node to the selected end node will be compared between both algorithms. Note that to ensure the A* algorithm has been correctly implemented, this will initially be checked against Dijkstra.

Efficiency

Ideally, processing time would be used to compare the processing efficiency of both algorithms, however, early tests calculating the CPU time used within the executing thread have shown significant differences on repeat executions for the same test data, so this is considered unreliable and likely caused by background applications and operating system activity.

Both A* and the new algorithm work through nodes and edges so for this project efficiency will be approximated by counting the total number of nodes and edges used in each algorithm's calculations. However, this can only be an indicator of efficiency and more research is required to determine a better and more accurate measure. For example, the new algorithm needs to map each node before it is processed whereas A* does not.

The test application

Given the complexity of calculations especially in large graphs a sophisticated tool is required to develop and test the algorithms. Listed below are the high-level requirements used to develop a test application to support this project.

High level requirements

1. The application should be able to handle multiple maps
2. Maps should be held in a format that can be easily generated
3. The application should be able to execute different algorithms to find the shortest path

4. There should be a visual representation of map data to aid analysis
5. A start node and multiple end nodes should be selectable
6. The visual representation should be movable and zoomable to make it easy to work with larger graphs
7. The visual representation should have nodes and distances marked
8. Diagnostic output showing key values should be output at key points in the execution of an algorithm
9. As the algorithm is executed the visual representation should be updated with progress
10. It should be possible to step through the algorithm so progress can be seen both in the diagnostics and visually
11. The test application should count the number of nodes and edges visited
12. The test application should allow a comparison to be made between A* and the new algorithm and output the measurement criteria in a format that can be consumed by Excel for reporting

User Interface

Figure 7.1 shows the test application user interface with key elements marked with numbers in an amber circle. Each of these elements are described in more detail below.

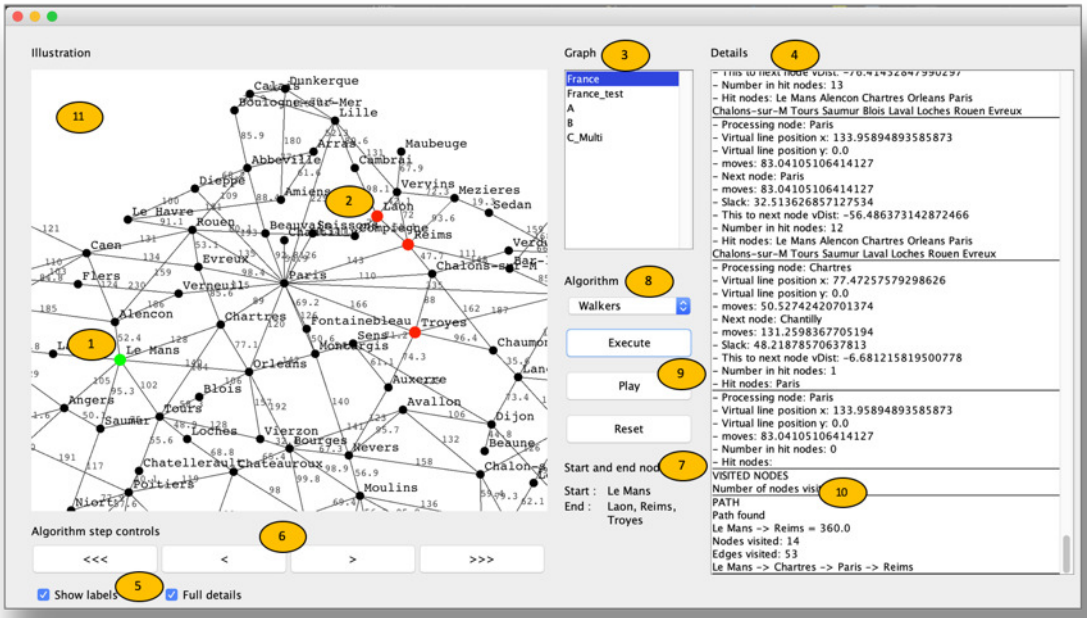


Figure 7.1. Test application user interface

Element 1

The start node is highlighted in green and is selected by pressing the left mouse button whilst it is near a node. The application calculates the Euclidean distance from the mouse to each node and will select the nearest node to the mouse pointer. Only one start node can be selected.

Element 2

End nodes are highlighted in red and can be selected by pressing the right mouse button. More than one end node can be selected. An end node can be unselected by clicking on it again. Similar to the start node selection, the application will choose the nearest node to the mouse pointer.

Element 3

This list shows the available graphs. A different graph can be selected by clicking on it.

Element 4

The details area shows diagnostic information output during the running of the algorithms such as the node being process, the position of the virtual line.

Element 5

The *show labels* checkbox will remove labels from the visual representation to give a clearer view of visual progress as in the image below (see Figure 7.2).

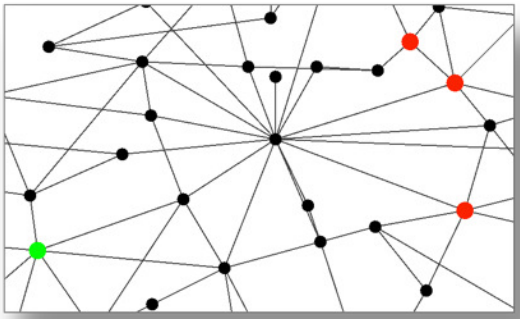


Figure 7.2. Visual representation without labels

Unchecking the *Full Details* checkbox will stop the diagnostics in the Details box being displayed. This speeds processing.

Element 6

These buttons allow the results of the algorithm to be stepped through. To accomplish the algorithm executes in full, but at each iteration it stores its main object state (see application design). '<<<' and'>>>' buttons go straight to the start or the end of the process.

Element 7

These labels show the names of the selected start and end node(s).

Element 8

This drop down allows the shortest path algorithm to be selected. Currently the following options are available.

Walkers (the new algorithm)
Full Dijkstra
Dijkstra
A*
COMPARE

COMPARE is a special function that will execute both Walkers and A*, do a comparison and output the results to a results file.

Element 9

These buttons allow the algorithm to be executed and reset (press RESET prior to re-executing). Additionally, following execution, if the Play button is pressed the visual representation will show an animation of progress in the search for the shortest path.

Element 10

The final diagnostic is the result (if a path is found). The end node, total distance, nodes visited, edges visited, and the full path are shown.

Element 11

The graph can be explored by clicking and dragging to move the view position and the mouse wheel can be used to zoom in and zoom out whilst exploring.

Following successful execution of an algorithm the view will show the shortest path by highlighting edges in red and circling nodes on the shortest path in red (see Figure 7.3). All nodes that have been hit are in dark blue, edges that have been active are marked in dark blue and nodes that have been checked (the end node of an active edge) are in light blue.

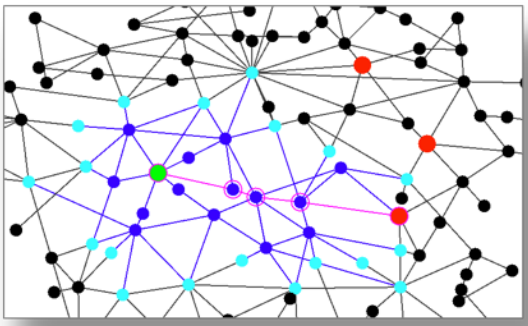


Figure 7.3. Visual representation of the shortest path

Application Design

The applications primary purpose is to support development and testing of the algorithm. To achieve this the interim state of the algorithm as it goes through each cycle is important for validation and debugging. One key design decision was the creation of a class 'ControlData' to allow the interim state to be saved during processing. The ControlData object is serialised and saved in an array at each iteration. It is this that allows execution to be stepped through with each object's state available at any point in the execution.

Since the application is user responsive a number of threads run concurrently to manage the visual representation, animation and user interface commands.

Visual representation of the data is handled through 3 different objects as follows:

a) Initial coordinates. This holds the starting coordinates for each node
b) View coordinates. This holds a representation of the viewing space taking into account any clicking and dragging and zooming of the view space
c) Display coordinates. This maps the view coordinates to the display

Mapping initial coordinates to virtual positions

An instance of translate class will perform a translation of the start node to the origin. This object can then be used to perform the same translation on all other nodes. After the translation, the boundary lines and virtual line can be calculated. A Vector class measure the angle between the

vector (0,1) and the virtual line (Vector class is based on Vector class source code (Eggen 2017)). All nodes are then rotated by this angle so the virtual line will lie on the x-axis. Following this all information to complete the mapping of a node to the virtual line is available. In the test application this is performed before the algorithm is executed. However, in a practical application it is only necessary to map points to their virtual positions when the node or edge is required which will significantly improve efficiency in a large graph. Additionally, the transformations required could be performed using matrices to further improve efficiency.

Map data

Available maps are checked at application launch. A list of available maps is held in a file as a simple string list. For each map there are two separate files containing node and edge data. One file contains a list of nodes in json objects along with their geographical coordinates and the other file contains a list of edges also in json objects (see examples below) along with the edge start node, end node and distance.

Available maps

"France"
"France_test"
"A"
"B"
"C_Multi"

Extract from node file for France map

{"name":"Dunkerque","x":392.0,"y":685.0}
{"name":"Calais","x":367.0,"y":680.0}
{"name":"Boulogne-sur-Mer","x":356.0,"y":666.0}
{"name":"Lille","x":427.0,"y":657.0}
{"name":"Arras","x":412.0,"y":630.0}

Extract from edge file for France map

{"node1Name":"Lille","node2Name":"Arras","distance":52.29999923706055}
{"node1Name":"Abbeville","node2Name":"Arras","distance":77.30000305175781}
{"node1Name":"Abbeville","node2Name":"Dieppe","distance":68.19999694824219}
{"node1Name":"Arras","node2Name":"Amiens","distance":61.599998474121094}
{"node1Name":"Dieppe","node2Name":"Le Havre","distance":100.0}

This format allows for extracts of a map to be easily created for bug fixing. For example, the map in Figure 7.4 was created to investigate a bug in finding the nearest node in multiple end node request. This took only a few minutes to create by extracting the relevant nodes and edges from the France data files.

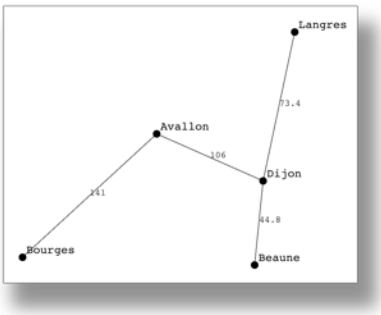


Figure 7.4. Extract from France map files for debugging

Comparison execution output

When the COMPARE function is executed it reads in a file of test conditions (see later in this section). When each algorithm is executed for a test condition it will output an instance of class AlgorithmOutput which includes the following attributes:

```
public String closestNode; // the closest node found
public Integer aStarPlusNodesHit; // the nodes tested for A* wrapper algorithm
public double distance; // the shortest path distance found
public Integer nodesHit; // the total number of nodes hit
public Integer edgesHit; // the total number of edges hit
public Integer totalHits; // the total hits (nodes + edges)
```

All of the AlgorithmOutput instances are combined into an instance of the CompareResult class which includes the following attributes.

```
public String startNode; // the name of the start node
public String endNodes; // a concatenation of all of the end nodes
public Integer numberOfEndNodes; // The number of end nodes in this test
public double minDist; // the geographical distance to nearest node
public double maxDist; // the geographical distance to furthest node
public double medianDist; // maxDistNode less minDistNode to give the breadth of the node
pool
public AlgorithmOutput aStarAlgorithmOutput;
public AlgorithmOutput aStarPlusAlgorithmOutput;
public AlgorithmOutput walkersAlgorithmOutput;
public boolean match; // do the algorithms agree
```

Each test condition will output the CompareResult object to a results file in csv format. An example of one test condition is below:

```
"Colmar","Cannes,Arles,Basel,Geneve,Aix-les-Bains,Digne,Aix-en-
Provence",7,44.94441010848846,397.24677468797654,221.0955923982325,"Basel",0,64.19
999694824219,610,1035,1645,"Basel",1,64.19999694824219,4,3,7,"Basel",0,64.1999969482
4219,1,3,4,"true"
```

Functional testing

In addition to the user interface a class TestMapInitialisation has been built to test each of the transformation functions in isolation. The output from these tests was used to compare with the excel models to ensure the accuracy of the input to the algorithms.

Generating test data

To check against the measurement criteria a significant amount of test data is required. All testing has been performed against a summarised view of the road network in France with 156 nodes and 283 edges.

A function has been built to take two inputs – the maximum number of end nodes, x, and the number of test cases, y, per end node. The function will then generate x*y test cases.

Test data is generated using the following method:

```
for 1…number of end nodes (x)
        for 1…number of test cases (y)
```

Choose a random start node
Choose a direction for the virtual line
Calculate boundary lines
Randomly choose x end nodes that are within the boundary lines

In creation of the test data the maximum, median and minimum distance from the start node to the end nodes is calculated. This is to allow the relative size of the cluster to be used in analysis if necessary. Each test case is saved as json in an instance of a TestDataInput class.

```
public Integer startNode; // The start node
public ArrayList<Integer> endNodeAL; // The end node array
public double minDist; // the geographical distance to nearest node
public double maxDist; // the geographical distance to furthest node
public double medianDist; // the median distance
```

Each test is saved as an array element within an instance of a TestData class.

```
public int maxNumEndNodes; // The maximum number of end nodes in test data i.e. 1..n
where n is maxNumEndNodes
public int numberCases; // The number of test cases for each number of end nodes e.g. 100
cases for 10 end nodes
public ArrayList<TestDataInput> testCase;
```

9. Algorithm comparison reports and analysis

Comparison results execution 1

After performing initial testing to prove the quality of the test reports, test cases were created for 1 to 20 end nodes with 100 test cases in each resulting in 2,000 test cases overall. The results of these tests are below.

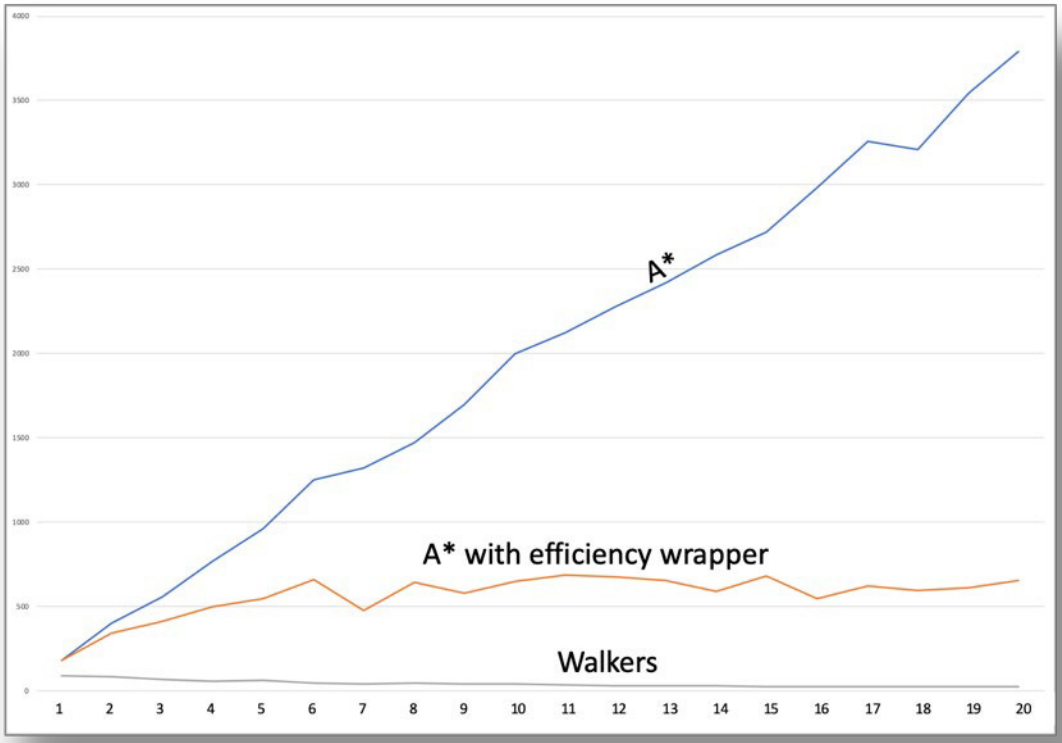


Figure 8.1. Average hits per number of end nodes

The graph in Figure 8.1 shows total number of hits (nodes plus edges) on the y-axis against the number of end nodes in the shortest path execution along the x-axis.

- A* represented by the blue line
- A* wrapper represented by the yellow line
- The new algorithm (Walkers) represented by the grey line

Results from the above 3 algorithms were executed and compared and it should be noted that for every test case all 3 algorithms agreed on the shortest path thereby satisfying the quality requirement.

These results are highlight some very interesting areas for the following reasons:

1. A* seems to follow a predictable linear growth. It seems intuitive that the number of hits increase linearly with each additional end node and this is evidenced in the graph.
2. A* with an efficiency wrapper also increases as the number of end nodes increases but then plateaus at about 6 end nodes and from there on appears to follow a straight line path
3. The new algorithm (Walkers) has roughly half of the hit nodes as A* for one end node which indicates using this metric that it is more efficient as A*

4. The new algorithm (Walkers) uses less hits the more end nodes are selected, so it actually becomes more efficient the more end nodes to search for

Given reason 1 is as expected, reasons 2 to 4 are investigated in the rest of this section.

Reason 2 – A* with efficiency wrapper plateaus

The A* wrapper algorithm depends on the likelihood that one of the nearest end nodes is more likely to be the shortest path. Figure 8.1 therefore looks to be an indication of the probability of this being the case. Figure 8.2 shows the number of end nodes on the x-axis and the average number of executions of A* required for each test case to find the shortest path. This does appear to support this hypothesis, but it doesn't appear to fully explain it as the trend still appears to be upwards whereas in Figure8.1 it plateaus.

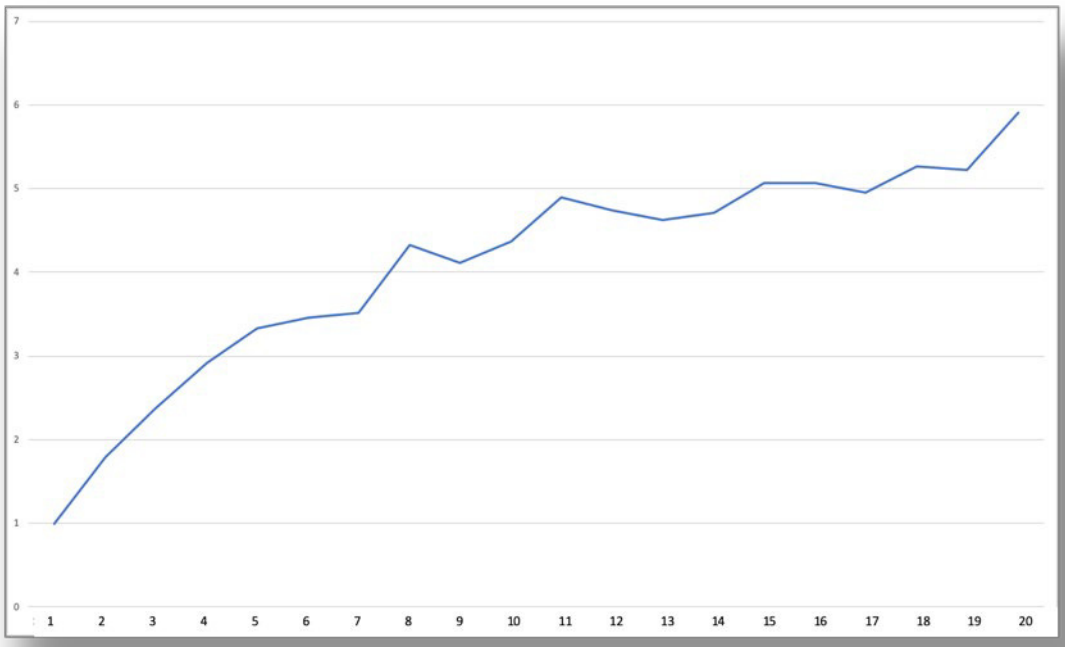


Figure 8.2. Average number of end nodes tested for A* with efficiency wrapper

One other factor that influences the A* with efficiency wrapper execution total number of hits is how far away the end node is from the start node. With more end nodes, the likelihood is that more of the end nodes will be closer to the start node. Taking this into account alongside the number of end nodes hit may explain the shape of the trend line.

The plateau is not the whole story though as this could hide some extreme variation in the hits per test case. Figure 8.3 shows the variation in nodes and edges hit (on the y-axis) for the 100 test cases (on the x-axis) where there are 20 end nodes. Even with the A* wrapper algorithm there is huge variation in hits with a minimum nodes hit value of 3 and a maximum nodes hit value of 3,573 (1,191 times bigger). For comparison, in Walkers the minimum is 2 and the maximum is 152 (76 times bigger).

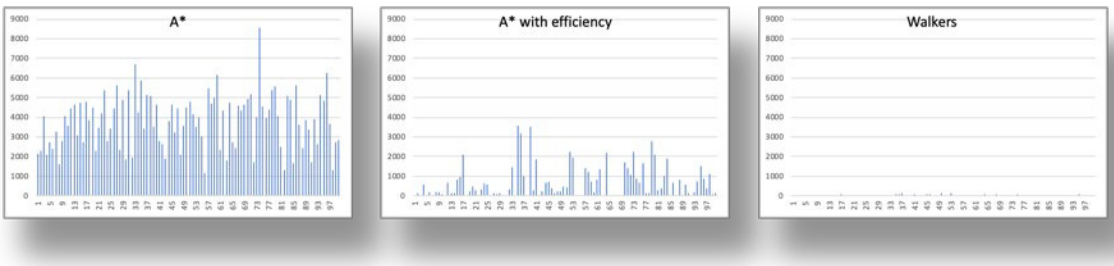


Figure 8.3. Average number of end nodes tested for A* with efficiency wrapper for 20 end nodes.

This would seem to indicate that the new algorithm will be more consistent in terms of efficiency.

Reason 3 – The new algorithm is twice as efficient as A*

Note that this is based on the given measurement criteria. The below graphs give a visual comparison between A* and the new algorithm for the same search. Although at first glance, A* looks to have covered a larger area, closer examination of the Walkers output shows that edges to the light blue coloured nodes on the A* picture are shaded red. This may indicate that an anomaly in the scoring approach is skewing the result for one end node in favour of Walkers algorithm.

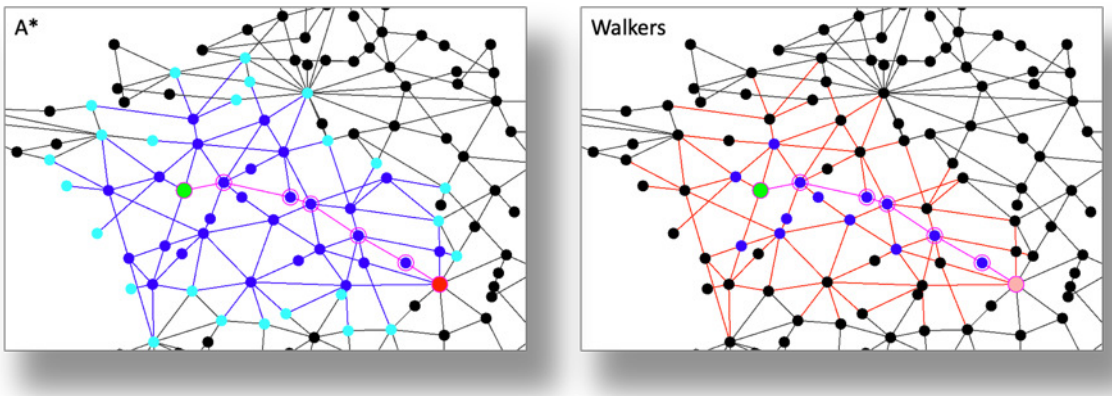


Figure 8.4. Visual map output comparison of A* versus the new algorithm

To understand in this in more detail, Figure 8.5 explores more fully the relationship between nodes and edges hit with each of the 100 test cases on the x-axis and the y-axis showing total nodes and edges hit for that test case (nodes are the blue line and edges the amber line). The graphs look at the one end node data only and are sorted by increasing total nodes hit.

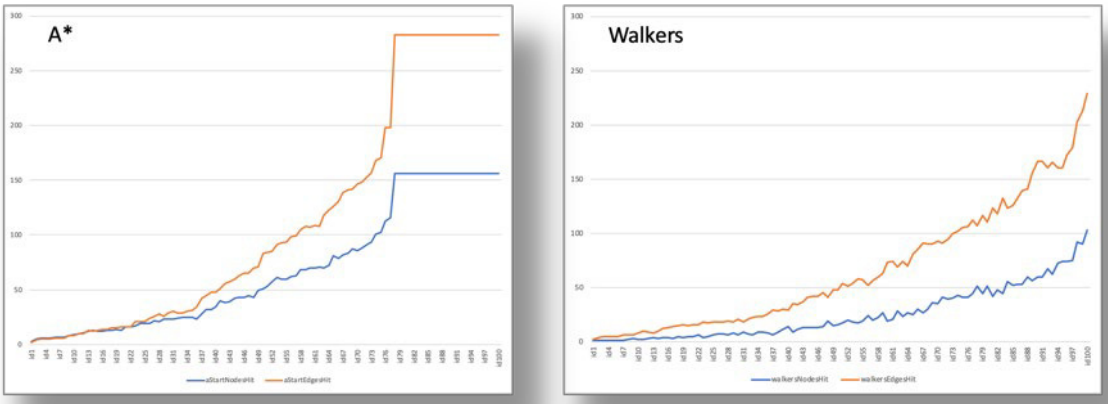


Figure 8.5. Ordered nodes hit for one end node of A* versus the new algorithm

The above charts are very illuminating. Looking at the A* graph, both the node hits and the edge hits seem to plateau at the 78th test case. The plateau is at 156 for nodes and 283 for edges which corresponds to the total number of nodes and edges in the graph. So from the 78th test case (c 20%) of the test cases A* has needed to examine every node and edge to find the shortest path.

Looking at the Walkers graph, the graph keeps growing in a smooth curve and doesn't reach the total number of nodes or edges in any test case. This seems to reflect a smooth growth and no anomaly in the reporting data for the new algorithm. To investigate this further, Figure 8.6 contains the search output for the 78th test case (Digne to Carcassone) which is the first test case to plateau in A*.

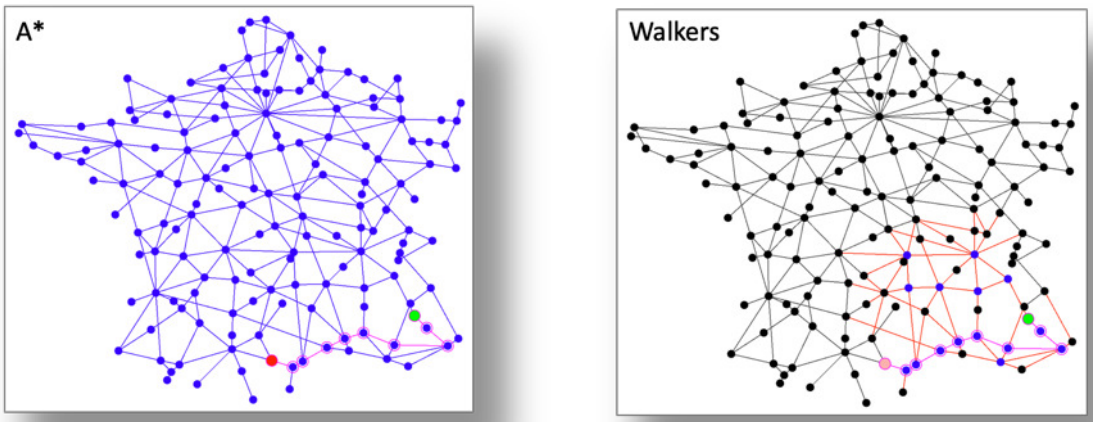


Figure 8.6. Example case with one end node with large difference

The above graphs show a remarkable difference between A* and Walkers which this time points to a possible anomaly in the A* algorithm result. Running the same search using Dijkstra brings up the following result:

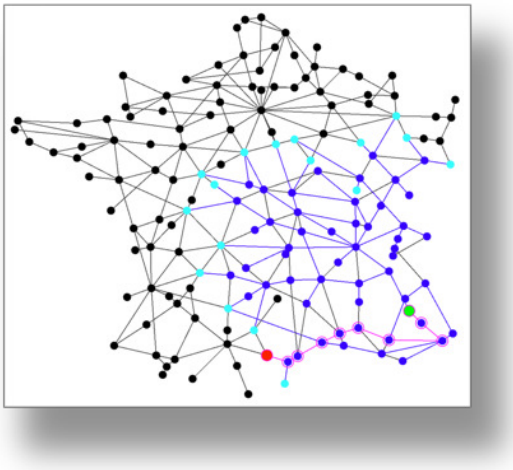


Figure 8.6. Visual comparison of A* versus the new algorithm

Figure 8.6 seems to show Dijkstra outperforming A* and therefore strongly points to a bug in the implementation of A*. Investigation identified the problem to be a condition in the code to check for the end node being found:

```
if (currentNode.nodeIndex == mycd.endNode)
```

The above code is failing to work in all cases as the node index and endNode are being held as Integer objects and this is attempting to find if they are referring to the same object and therefore causing an unreliable result. This code line was replaced with the following line to ensure the value of the Integer object is compared and not the object:

```
if (currentNode.nodeIndex.equals(mycd.endNode))
```

Retesting Digne to Carcassone for A* then produces the following result in Figure 8.7.

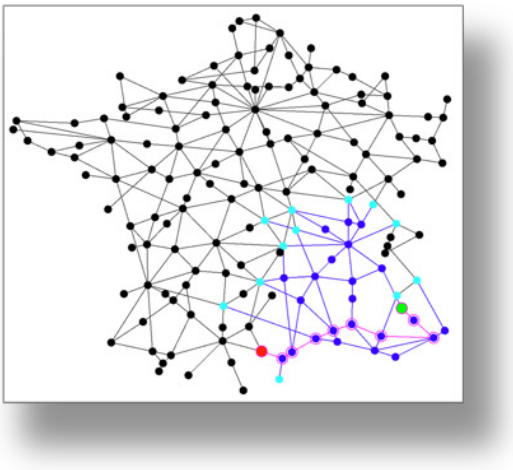


Figure 8.7. Visual comparison of A* versus the new algorithm

This looks more reasonable. This potentially invalidates the previous findings. Therefore, the comparison tests have been re-executed as follows and referred to as execution 2.

Comparison results execution 2

Re-running the comparison with the updated code produces similar results but with improved efficiency for A* and A* wrapper. However, the new algorithm now appears to have a similar performance to A* for one end node. See Figure 8.8 to show execution 2 against execution 1 for comparison and also a large image of execution 2 for detail. As before the number of end nodes is on the x-axis and the y-axis shows the number of executions.

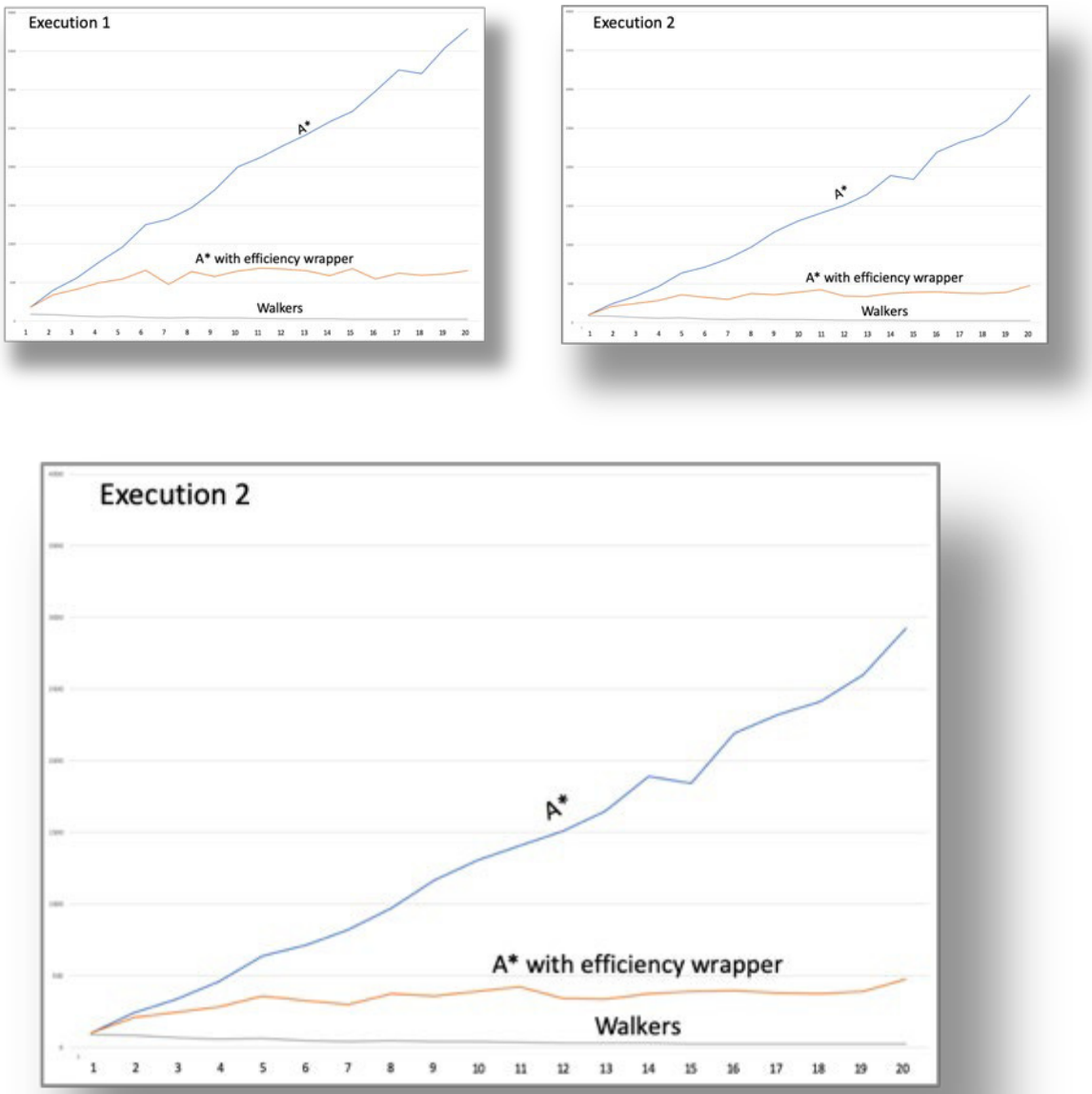


Figure 8.8. Comparison results execution 1 versus execution 2

Reason 2 review

Rerunning the graphs produces a similar result as follows:

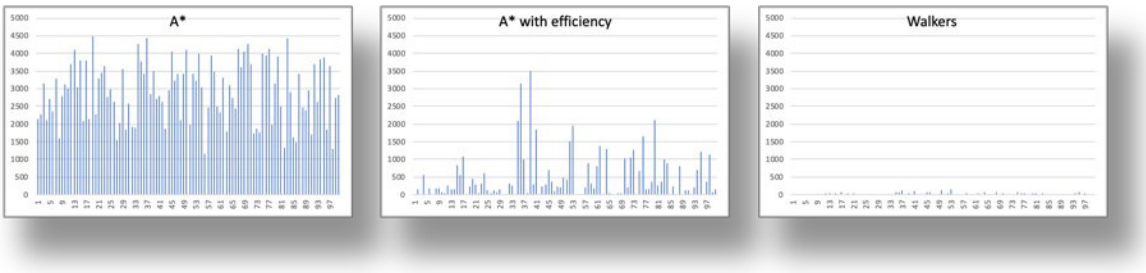


Figure 8.9. Rerun on Figure 8.3, Average number of end nodes tested for A* with efficiency wrapper for 20 end nodes

The analysis of Reason 2 in execution 1 remains correct. Figure 8.9 shows a rerun of the variation graphs for 20 end nodes. The variation for A* remains unchanged with a range from 3 to 3,510 (1,171 times bigger as before). For comparison, in Walkers the minimum is 2 and the maximum is 152 (76 times bigger). Therefore, the new algorithm still appears to be much more consistent for multiple end nodes.

Reason 3 review

Continuing the investigation as to whether results are skewed in favour of the new algorithm, Figure 8.10 is an updated version of Figure 8.5 but with both algorithms plotted on the same graph. The graphs look at one end node data only and are sorted by increasing total nodes and edges hit for A* and the corresponding nodes and edges hit for the new algorithm. The graph then shows a breakdown of nodes hit and edges hit for each test case. The updated results show that the new algorithm and A* are performing with similar efficiency for a single end node. This invalidates reason 3 and means the new algorithm is not more efficient than A* for 1 end node.

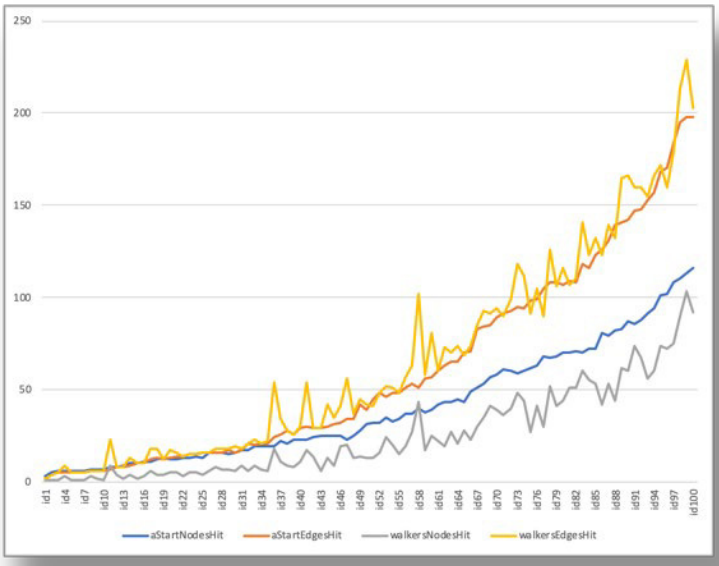


Figure 8.10. Visual comparison of A* versus the new algorithm

It is interesting that the lines on the graph in Figure 8.10 are curved. The graph contains the set of 100 test cases for one end node sorted in order of total number of nodes hit. Given that the test data is randomly selected, the reason for a curve rather than a straight line is not immediately obvious. However, this can be explained by the structure of the map. As the number of nodes hit is growing there is the potential for each hit node to impact on the number of edges and connected nodes and therefore the curve of this graph may be containing reflecting information about the number and regularity of edges within the map. This is referred to as the branching factor which influences the efficiency of the algorithm as described earlier in this report.

Reason 4 - The new algorithm uses less hits the more end nodes are selected

The new algorithm prioritises a search along paths in the direction of a segment containing all of the end nodes. The more end nodes there are, the greater the probability that there is an end node closer to the start node thereby reducing the number of nodes and edges that need to be searched. If this assumption holds true, then a graph showing the average distance to the nearest node for each number of end nodes should show this distance decreasing for each end node.

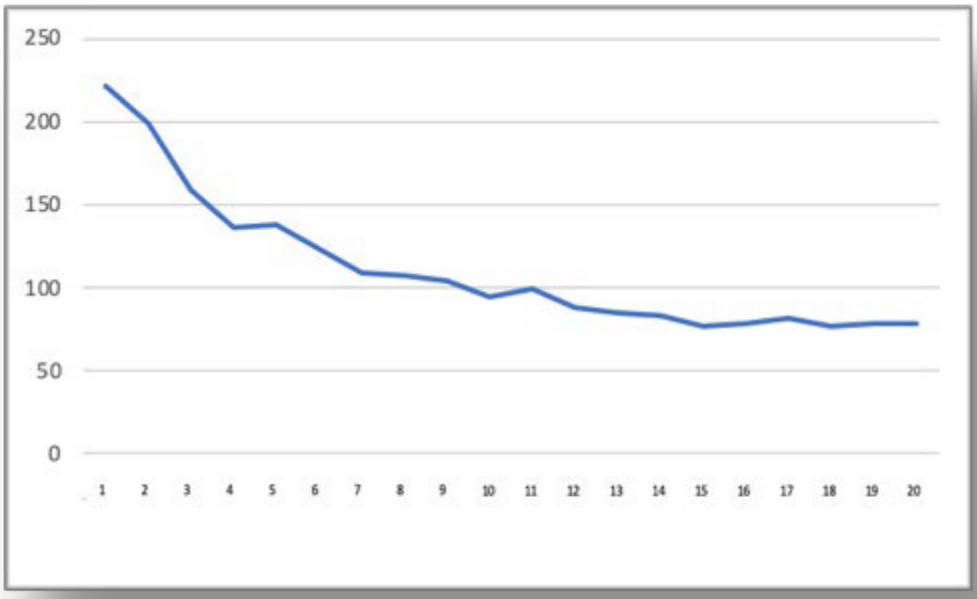


Figure 8.11. Visual comparison of A* versus the new algorithm

Figure 8.11 maps the number of end nodes on the x-axis against the average distance to the geographically closest end node on the y-axis and this confirms that the nearest end node is becoming closer to the start node as the number of end nodes increases. This seems to satisfy reason 4 and also highlights an issue with the creation of the test data. The random selection of end nodes in a directional segment is successfully testing the quality, but it does not seem to satisfy some of the use cases such as the nearest car park in a nearby city. In this use case the requester is likely to be distant from the city and the end nodes are likely to be more closely located in a cluster.

To attempt to better simulate the above use case, the existing data will be filtered to select only those test cases where the closest end node is in the furthest quartile and where the gap between the nearest and furthest end node is in the smallest quartile.

Only a small number of test cases fit this criteria as shown in Figure 8.12 graph below. In Figure 8.12 the x-axis is the number of end nodes and the y-axis is the number of test cases that meet the above filter.

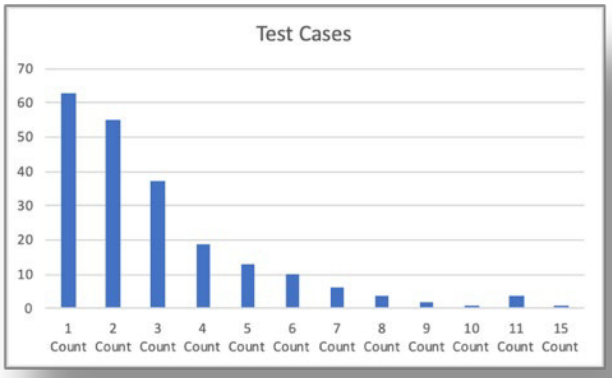
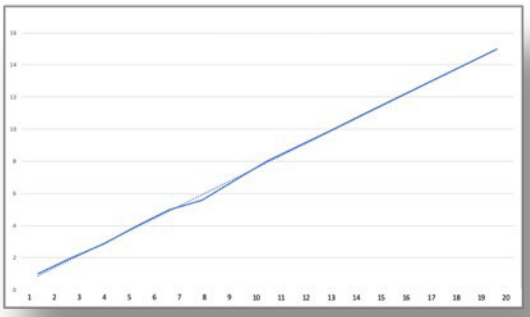


Figure 8.12. Visual comparison of A* versus the new algorithm

Figure 8.13, with trend lines added, shows a different story to the previous graphs for A*. Now that the end nodes are more clustered the graph on the left shows the number of nodes hit growing linearly with the number of end nodes selected and this is not plateauing as before. This removes any advantage in using the A* wrapper algorithm.

Additionally, the number of nodes hit reflects the number of end nodes being evaluated with A* and consequently also indicates a linear growth (see the right graph in Figure 8.13). However, the new algorithm continues to maintain a relatively low hit count that appears to remain constant regardless of the number of end nodes.



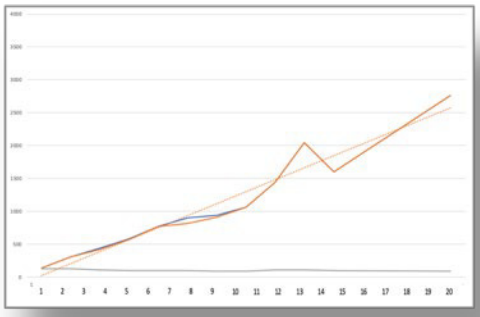


Figure 8.13. Visual comparison of A* versus the new algorithm

This indicates for a clustered set of end nodes (i.e. a set of end nodes relatively close together) the new algorithm is significantly more efficient than A* and A* with a wrapper and that this efficiency increases linearly based on the number of end nodes in the search.

10. Conclusions

The testing carried out has demonstrated that the new algorithm will find the shortest path to a single or multiple end nodes that is consistent with the results from A* and therefore it meets the quality measurement criterion.

In terms of efficiency the new algorithm appears to be significantly more efficient than A* and the A* wrapper algorithms in finding multiple end nodes. This efficiency increases in relation to the number of end nodes and the relative proximity of the end nodes with the greatest efficiency where the end nodes are closest together. In addition, the new algorithm has much lower variation in the least and greatest number of nodes hit so is more consistent. These findings were dependent on the test data and the efficiency scoring method. Given that the efficiency increase is so large it is likely any anomalies in the node and edge counting will not affect the conclusion that the new algorithm is more efficient than A* wrapper under these conditions. To confirm these findings, the efficiency scoring method should be reviewed and improved test data should be used based on one of the real-world use cases, such as finding the nearest electric car charging point in a city being visited.

11. Areas for Further Research and Development

During the development of this project the following areas have been identified as candidates for more development and / or research:

Add capability for priority queue

To improve the efficiency of the algorithm, a method for implementing a priority queue should be investigated. This is an efficient way of handling an ordered set of values such as the lowest or the highest can be selected for processing. For the new algorithm this is made more difficult as the values in the ordered list are recalculated at selection.

Finding the shortest path from multiple start nodes to a single end node

The simplest method to find the shortest path from multiple start nodes to a single end node would be to reverse the start and end nodes and therefore the end node will become the start node the multiple start nodes will be the multiple end nodes. However, this could become more difficult in a travel application where distances are directional and affected by travel conditions such as accidents and congestion. An alternative approach is to introduce the concept of a single virtual start node.

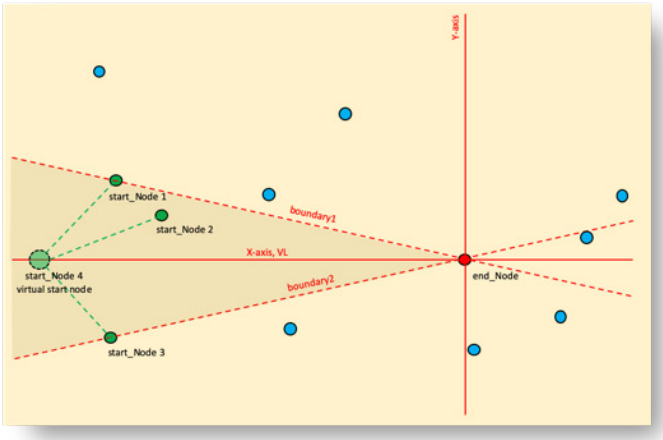


Figure 10.1. Approach for multiple start nodes to single end node

Figure 10.1 shows how the new algorithm might be adapted to find the shortest path from multiple start nodes to a single end node. A virtual start node (start_node 4) is mapped to the virtual line and this replaces the start node used in the algorithm. Virtual edges are created from the virtual start node to each start node. These edges are initialised to have zero slack and an effective distance of zero for calculation in the shortest path. The shortest path will have to pass through one of the start nodes as the virtual start node is only connected to these.

Finding the shortest path from multiple start nodes to multiple end nodes

Figure 10.2 develops the idea above further to enable the shortest path from multiple start nodes to multiple end nodes to be calculated.

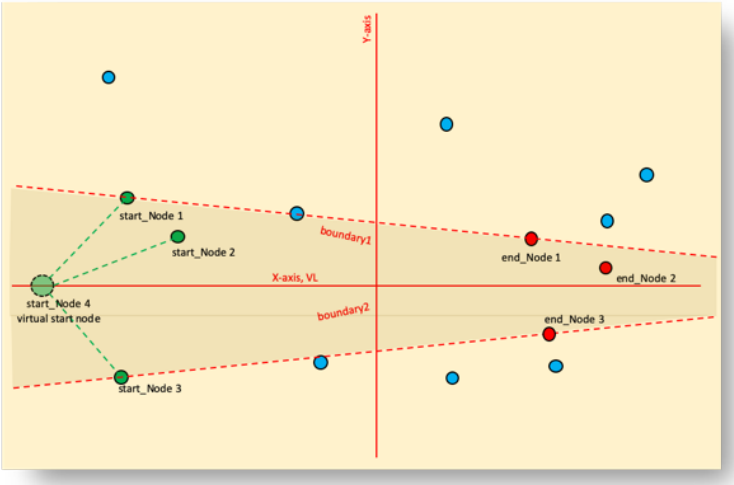


Figure 10.2. Approach for multiple start nodes to multiple end nodes

This continues with the concept of a virtual start node as a start point. The boundary lines become lines joining the outermost start and end nodes with the virtual line bisecting these as before. Finally the virtual line is rotated to the x-axis. Mapping to the virtual line should be researched further and will be based on the single start node to multiple end node approach.

Finding the shortest path through multiple clusters

Taking this idea even further, it may then be possible to find the shortest path from multiple start nodes through a cluster of multiple interim nodes to multiple end nodes where the shortest path is from any start node through any interim node to any end node.

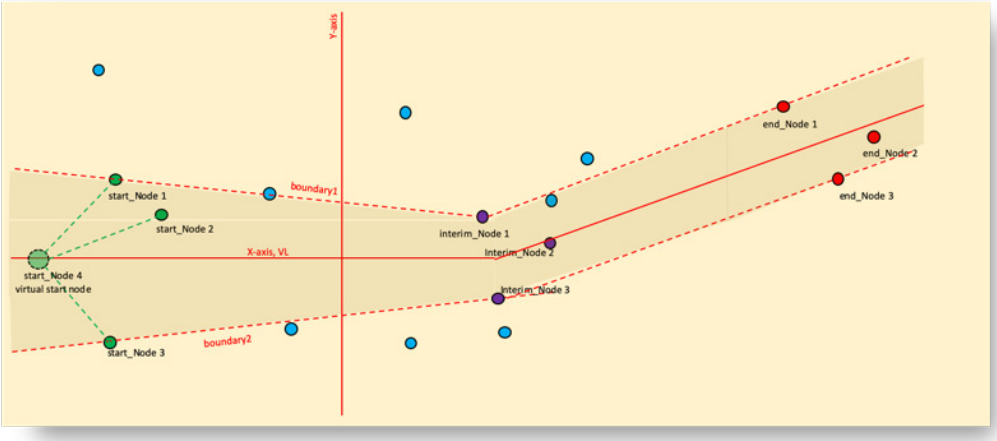


Figure 10.3. Approach for multiple start nodes to single end node

The above illustration shows how this might be developed. In this case the boundary lines change direction at the interim cluster. As previously mapping will be based on the original virtual line mapping, however mapping will be affected by the change in direction of the boundary and virtual lines and would need to handle cases where direction reversed. When an interim node is hit, each hit interim node will need to track a duplicate set of nodes and edges to allow for backtracking along edges should the end nodes be in the direction of the start nodes.

Review the scoring method for efficiency comparison

A more sophisticated scoring system that takes into account statements executed and ideally processor time executed should be explored. This could incorporate an error range that would allow the efficiency of the new algorithm to be better evidenced.

Incorporation of real-world test data

Openstreetmap can be used to integrate real world maps into testing. The data in openstreetmap is held as nodes and ways. Each way holds a path of connected nodes. Each node has a longitude and latitude. The geographical distance can be calculated by calculating the distance between each node in a way. Code can then be written to identify the nodes where ways connect. Using this data, openstreetmap data can be integrated into the test application for this project and used to provide bigger and more realistic map data for further testing.

Testing the regularity of the map

A regular map might be described by a set of regularly spaced nodes each with the same edges connecting the nodes to neighbours. Comparison of multiple tests to find the shortest path could lead to inferences about the graph by comparing results to expected results from a regular map. For example the number of times the A* wrapper algorithm needs to execute A* could indicate unconnected areas of the map. This could be used make inferences from maps which contains some unknown data.

Cluster analysis

The new algorithm could be used to find the minimum and maximum distances between clusters of nodes.

Shortest path in 3 dimensions

The new algorithm could be expanded to work on 3 dimensional maps.

Matrix transformations

Coordinates and transformations could be done using matrices. This could allow transformations to be combined and much faster execution time by making use of hardware optimised for matrix calculations.

Travelling salesman problem

The travelling salesman problem is one that describes the problem of a salesman trying to work out the shortest journey for him to visit multiple destinations. This is described as an NP-hard problem. The ability of the new algorithm to find shortest paths from multiple nodes to multiple nodes and passing through multiple nodes could lead to a new approach for finding solutions to the travelling salesman problem.

Reviewing the complexity (order) of the new algorithm

The success of the new algorithm will depend on its complexity and the number of nodes and edges it needs to examine to find a solution. Currently, this is exponential and more work is required to ensure the new algorithm is defined to perform at least as well as A*.

12. Reflection/Learning

The importance of modelling

The idea for this project formed during a lecture, however it was modelling on a whiteboard and Excel that gave it substance and indicated it might work in practice. After the whiteboard and Excel modelling I tried to replicate this in code, however I was unable to make this work until I had revisited Excel and created sample data allowing me to step through an example until a shortest path had been found. This reinforced to need to ensure I have spent sufficient time in completing and understanding the design before coding.

Tenacity

There were several times when it looked like the new algorithm wouldn't work. At these times I needed to step away from the problem and do something else. When I came back methodically re-tracing my steps enabled me to come up with a solution. This reinforced the need to persevere even when a solution does not appear to exist.

Avoid making assumptions

Although I had read through the Dijkstra and A* algorithms and thought I understood them, I didn't really understand them until I worked through the algorithms, developed my own code and ran through some examples. It takes strong self-discipline to take the time to work through existing knowledge prior to building something new and this is a lesson I will take forward in future research.

The importance of prototyping and unit testing

As soon as the design was understood I found it very valuable to write some code. This enabled examples to be executed quickly, however I was slowed by some code defects. To overcome these, I wrote mini test packs for the individual functions in use and then tested input data against output from Excel. This then gave confidence in integrating these functions together and enabled debugging to focus on only the new code. This worked very well for the initial mapping of coordinates to virtual positions and the different iterations of code versions and is an approach I will continue to use when developing software.

Analysis of output

As highlighted in the comparison report section above, when looking through the initial output I found an unexpected result in one of the graphs. Initially I thought it was easy to explain but I looked for further evidence to support this. The test example didn't support my initial assumption, and this led me to find a bug in my implementation of A*. I was really pleased I investigated further and reinforced to need to provide evidence for any inference made from the data even when this creates a lot more work.

Visualisation of the test data

From past experience and also working on modelling in Excel I knew that it would be really hard to understand what was happening during execution of the algorithm. Although it required a lot more work, this is why I developed an application with a user interface and the ability to step through execution with a visual display of progress along with detailed debugging output. This was invaluable in the test and analysis phase and reinforced the need to consider the test requirements in the initial development requirements.

Flexibility

The aim of the new algorithm was initially to rival A*. Although it does do this, it is unlikely to be as efficient for single node to single node shortest paths. After careful consideration though, I realised that the approach would lend itself to multiple node problems and therefore after discussion with my supervisor I updated to the project to include multiple end nodes. This created more design and development work but enabled me to explore new opportunities to test the algorithm's efficiency. This reinforced the need to be open to opportunities for use which may lead in a slightly different direction.

13. References

Schrijver, A. 2012. *On the History of the Shortest Path Problem.* University of Washington. *Available at: https://www.math.uni-bielefeld.de/documenta/vol-ismp/32_schrijver-alexander-sp.pdf*

Minty, GJ. 1957. *Letter to the Editor—A Comment on the Shortest-Route Problem. Operations Research 5(5):724-724. Available at: https://doi.org/10.1287/opre.5.5.724*

Dijkstra, EW. 1959. *A note on two problems in connexion with graphs.* Numerische Mathematik 1: 269-271

Hart, P et al. 1968. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics. **4** (2): 100–107.

Eggen, J. 2017. *Vector class source code.* [Source code]. Available at: https://codereview.stackexchange.com/questions/181384/making-math-vector-implementation-using-java [Accessed: 13 April 2020].

14. Appendices

Shortest String Path Proof

Theorem

This theorem concerns a geographical graph with 2 or more nodes and with 1 or more edges with a weight equal to the Euclidean distance between the connecting nodes. Given any start and end node where edges exist that connect the start and end node and which lie on a straight line from the start node to the end node and there are no overlapping edges, then the sum of the edge weights will be the shortest distance between the nodes.

Statement of Proof

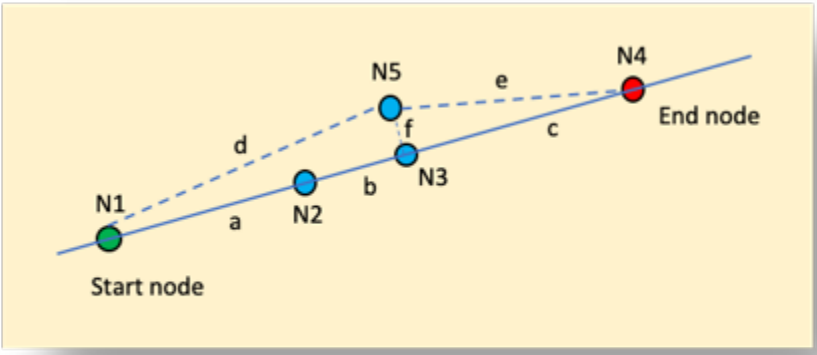


Figure A1. Shortest String Path Proof

Consider a path that lies on the straight line connecting the start node to the end node and the path consists of 0 or many interim nodes. For example, in Figure A1, this path is N1 -> N2 -> N3 -> N4. Suppose that this path is the shortest path, then the distance of this path will be a+b+c.

Suppose that there is a shorter path that does not lie along the straight line from the start node to the end node. In Figure 1, this is represented by the path N1 -> N5 -> N4. Using Pythagorus's theorem the length d can also be calculated as the square root of $(a+b)^2 + f^2$. Similarly the length e can be calculated as the square root of $c^2 + f^2$.

Since N1 -> N5 -> N4 is the shortest path then $\sqrt{((a+b)^2 + f^2)} + \sqrt{(c^2 + f^2)}$ must be less than a + b + c

$\sqrt{((a+b)^2 + f^2)}$ can be rewritten as follows:

$\sqrt{((a+b)^2 + f^2)} = \sqrt{((a+b)^2(1 + (f^2/(a+b)^2))} = \sqrt{((a+b)^2)} * \sqrt{(1 + f^2/(a+b)^2))}$

$= (a+b) * \sqrt{(1 + f^2/(a+b)^2))}$

$\sqrt{(1 + f^2/(a+b)^2))}$ must be >= 1 since $\sqrt{(1 + x)} >= 1$ where x >= 0

Similarly, $\sqrt{(c^2 + f^2)}$ can be rewritten as follows:

$\sqrt{(c^2 + f^2)} = \sqrt{(c^2(1 + f^2/c^2))}$

$= c * c^2(1 + f^2/c^2)$

The length N1 -> N5 ->N4 can be rewritten as

((a+b) * x) + (c * y) where x>=1 and y >= 1 - *Statement 1*

QED

Given statement 1, ((a+b) * x) + (c * y) >= a + b + c and therefore the length N1 -> N5 ->N4 cannot be shorter than the length N1 -> N2 -> N3 -> N4 (note however, if f=0, then this distance N1 -> N5 ->N4 could equal the shortest path)

Definition of slack

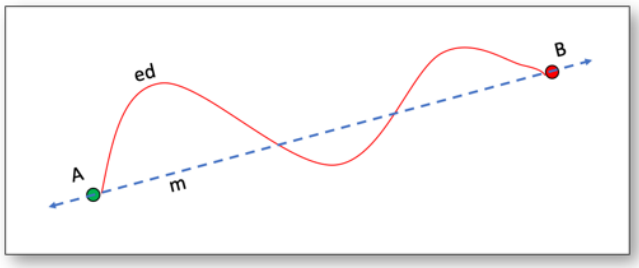


Figure A2. Definition of slack

Consider Figure A2.

There are two nodes, A and B.
Let the Euclidian distance between node A and B is m.
Let the distance along the edge A -> B (marked in red) be ed.
Let AB be the straight line passing through nodes A and B.
Let f(A, B, Z, ed) be a function that translates the point A in the following manner:

- Node A is translated a distance of Z units along AB such that the Euclidean distance, m, between the two points increases by m
- If following the translation, m is greater than the edge distance between node A and node B, ed, then node A will be moved to a point on AB such that m is equal to ed.

The slack is the maximum distance that node A can move along AB without m being greater than ed.

Therefore, the slack is ed – m

If the above function is called repeatedly with a small amount for Z eventually m will be equal to ed.

Rather than call the translation function repeatedly, the translation may be executed with one function call by initially calculating the slack as ed-m and then passing this value in a parameter Z. This will result in node A moving along the line AB, ed-m units as shown in Figure A3.

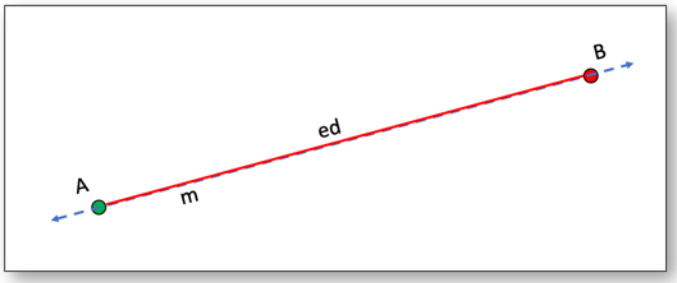


Figure A3. Translation by the slack amount

Given that the distance from node A to B will always result in a value ed, then the start position of A along the line AB will not affect the end position and therefore the start position is arbitrary providing m <= ed.

Key source code

Mapping to the virtual line

```
public void walkersSetupVirtualContext () {

        // New code caters for multiple end nodes
        // First, create a translation to map the start node to the origin

        mycd.tn = new Translate(new Coord(mycd.nodeArray.get(mycd.startNode).myNodeInit.x,
        mycd.nodeArray.get(mycd.startNode).myNodeInit.y) , new Coord(0.0, 0.0));

        // Translate all points using the above translation

        Coord oldCoord, newCoord;
        for (int i=0; i<=mycd.nodeArray.size()-1; i++) {
                oldCoord = new Coord(mycd.nodeArray.get(i).myNodeInit.x,
        mycd.nodeArray.get(i).myNodeInit.y);
                newCoord = mycd.tn.translate(oldCoord);
                mycd.nodeArray.get(i).myNodeInitVL = new NodeInit(mycd.nodeArray.get(i).myNodeInit.name,
        newCoord.x, newCoord.y);
        }

        // Using selected end nodes, calculate Virtual Line, boundary lines and rotation to place virtual line
        along the +ve x-axis

        ArrayList<Coord> endNodeAL;
        endNodeAL = new ArrayList<Coord>();

        for (int i=0; i<=mycd.endNodeAL.size()-1; i++) {
                endNodeAL.add(mycd.nodeArray.get(mycd.endNodeAL.get(i)).myNodeInitVL.getCoord());
        }

        Line.findVLAndBoundaries(mycd, endNodeAL, (9.0/10.0 * Math.PI));

        // Loop through all points and map to virtual line coordinates.  This is done up front in this example
        code,
        // but to improve efficiency the virtual line coordinates only need to be calculated once when a node is
        evaluated.
        // Coordinates are copied to an intermediate array to allow for reuse by test methods

        ArrayList<Coord> nodeAL;
        nodeAL = new ArrayList<Coord>();

        for (int i=0; i<=mycd.nodeArray.size()-1; i++) {
                nodeAL.add(mycd.nodeArray.get(i).myNodeInitVL.getCoord());
        }

        Coord myCoord, myCoordRotated;
        boolean iwb;

        for (int i=0; i<=nodeAL.size()-1; i++) {
                myCoord = nodeAL.get(i);
                myCoordRotated = myCoord.rotate(mycd.rotVlAngle);
                iwb = myCoordRotated.coordWithinBoundaries(mycd, true);
                newCoord = myCoordRotated.mapCoordToVl(mycd, iwb);
                mycd.nodeArray.get(i).myNodeInitVL.setCoord(newCoord);
        }

        // To simplify the search (as a test), map all of the end points to the same virtul position
        // First find the lowest x value (i.e. the closest to the startnode)
        double lowestEndNodeXValue = Double.MAX_VALUE;
        for (int i=0; i<=mycd.endNodeAL.size()-1; i++) {
                if (mycd.nodeArray.get(mycd.endNodeAL.get(i)).myNodeInitVL.x < lowestEndNodeXValue) {
                        lowestEndNodeXValue =
        mycd.nodeArray.get(mycd.endNodeAL.get(i)).myNodeInitVL.x;
```

```
                }
        }
        // Next update all of the endNode x values to the found low value
        for (int i=0; i<=mycd.endNodeAL.size()-1; i++) {
                mycd.nodeArray.get(mycd.endNodeAL.get(i)).myNodeInitVL.x = lowestEndNodeXValue;
        }

}

public static void findVLAndBoundaries (ControlData inCD, ArrayList<Coord> inCoordAL, double maxAngle) {

        // This method will calculate the maximum boundary line, the minimum boundary line and
        // VL (the Virtual Line) which bisects them.  The angle between the maximum and the minimum
        // boundary line must be within the input maxAngle.  All angles are in radians.

        double maxGapAllowed = (Math.PI*2.0) - maxAngle;

        if (maxAngle > Math.PI) {
                System.out.print("Line.findVL :: ERROR - maxAngle exceeds PI (180 degrees) and is invalid");
                System.exit(0);
        }

        ArrayList<Line> outLineAL;
        ArrayList<Double> angleAL;

        outLineAL = new ArrayList<Line>();
        angleAL = new ArrayList<Double>();
        Coord myCoord;
        Line myLine;
        Vector vectorYAxis = new Vector (0,1,0);
        Vector myVector;
        double angle;

        // First find the angles between x-axis and lines going through the origin and input Coordinates
        for (int i=0; i<= inCoordAL.size()-1; i++) {

                myCoord = inCoordAL.get(i);
                if (myCoord.x == 0 && myCoord.y == 0) { // If at the origin, ignore
                        angleAL.add(null);
                }
                else {
                        myVector = new Vector(myCoord.x, myCoord.y, 0);
                        angle = vectorYAxis.angleClockwiseFrom0(myVector);
                        angleAL.add(angle);
                }

        }

        // Next work out the minimum segment angle from the origin that will include all coordinates
        // This is done by looking for a gap between angles greater than 2*PI (a full circle) less than maxAngle
        // Start from the lowest angle and then look for a gap between each successive angle.  Start by adding
        the first
        //  angle to the end of the arraylist  with the addition of 2*PI to ensure that the gap crossing the start
        point
        //  is also considered.
        ShortestPathUtility.sort(angleAL);
        angleAL.add(angleAL.get(0) + (Math.PI*2.0));

        double angleVL = 0, angleBoundary1 = 0, angleBoundary2 = 0, index1 = 0, index2 = 0;
        boolean valid = true;
        int maxIndex = -1;
        double maxGapFound = -1;

        for (int i=0; i<=angleAL.size()-2 && valid; i++) {
                if ( (angleAL.get(i+1) - angleAL.get(i)) > maxGapFound) {
                        maxGapFound = (angleAL.get(i+1) - angleAL.get(i));
                        maxIndex = i;
```

```
            }
        }

        if ( maxGapFound < maxGapAllowed) {
            // Correct gap found so calculate lines
            valid = false;
            System.out.println("INVALID, gap: "+maxGapFound+" < "+maxGapAllowed);
            inCD.multiEndNodeValid = false;
            System.out.print("Line.findVL :: ERROR - maxAngle exceeded");
            System.exit(0);
        }

        if (valid) {

            index1 = angleAL.get(maxIndex);
            index2 = angleAL.get(maxIndex+1);
            if (index2 >= (Math.PI*2.0)) {
                index2 = index2 - (Math.PI*2.0);
            }
            if (index1 <= index2) {
                angleBoundary1 = index1;
                angleBoundary2 = index2;
            }
            else {
                angleBoundary2 = index1;
                angleBoundary1 = index2;
            }

            // Check if there is a shorter bisection as it crosses 0 degrees
            double tmpAngleBoundary1 = angleBoundary1;
            if ((angleBoundary2 - tmpAngleBoundary1) > Math.PI) {
                tmpAngleBoundary1 = tmpAngleBoundary1 + (Math.PI*2.0);
            }

            angleVL = tmpAngleBoundary1 + ((angleBoundary2 - tmpAngleBoundary1) / 2);
            if (angleVL >= (Math.PI*2.0)) {
                angleVL = angleVL - (Math.PI*2.0);
            }

            inCD.boundary1Angle = angleBoundary1;
            inCD.boundary2Angle = angleBoundary2;
            inCD.vlAngle = angleVL;
            inCD.boundary1Line = Line.findLineFromAngle(angleBoundary1);
            inCD.boundary2Line = Line.findLineFromAngle(angleBoundary2);

            // Calculate the angle to rotate vlAngle to positive x-axis
            inCD.rotVlAngle = ( (2.0*Math.PI) + (Math.PI/2.0) - inCD.vlAngle);
            if (inCD.rotVlAngle >= (2.0*Math.PI)) {
                inCD.rotVlAngle = inCD.rotVlAngle - (2.0*Math.PI);
            }

            inCD.boundary1AngleRotated = inCD.boundary1Angle + inCD.rotVlAngle;
            if (inCD.boundary1AngleRotated >= (Math.PI*2.0)) {
                inCD.boundary1AngleRotated = inCD.boundary1AngleRotated - (Math.PI*2.0);
            }

            inCD.boundary2AngleRotated = inCD.boundary2Angle + inCD.rotVlAngle;
            if (inCD.boundary2AngleRotated >= (Math.PI*2.0)) {
                inCD.boundary2AngleRotated = inCD.boundary2AngleRotated - (Math.PI*2.0);
            }

        }

    }
```

The new algorithm

```
public void walkers () {

// This function will find the shortest path from a single start node to one or multiple end nodes

ThreadMXBean threadMXBean;
long cpuStartTime, cpuEndTime;

threadMXBean = ManagementFactory.getThreadMXBean();
cpuStartTime = threadMXBean.getCurrentThreadUserTime();

Gson myGson = new Gson();
String gsonString;
DecimalFormat df = new DecimalFormat("####0");

boolean found;
this.hitNodes = new ArrayList<Integer>();
this.walkersSetupVirtualContext();

// Block 0
this.hitNodes.add(mycd.startNode);
mycd.nodeArray.get(mycd.startNode).moves = (double) 0;

// Block 1
boolean conditionHit;

// Find the next edge to process
this.walkersFindNextEdge();

while (this.nextEdge != null) { // loop until no more edges are found

	conditionHit = false;

	// if first time in or a shorter path is found to the next node then the condition is hit
		if (this.nextNode.moves == null) {
			conditionHit = true;
		}
		else if ( (this.thisNode.moves +
				(this.nextEdge.myEdgeInit.distance + this.walkersCalcVDist(this.thisNode,
		this.nextNode)))
				< this.nextNode.moves) {
			conditionHit = true;
		}

		// If condition hit, update the next node details
		if (conditionHit) {
			this.nextNode.moves = (this.thisNode.moves +
				(this.nextEdge.myEdgeInit.distance + this.walkersCalcVDist(this.thisNode,
		this.nextNode)));
			this.nextNode.pathDistance = this.thisNode.pathDistance + this.nextEdge.myEdgeInit.distance;
			this.nextNode.fromNode = this.thisNode.nodeIndex;

			// Make all connected edges active (i.e. available for selection)
			for (int j=0; j<=this.thisNode.myEdgeAL.size()-1; j++) {
				this.thisNode.myEdgeAL.get(j).active = true;
				this.thisNode.myEdgeAL.get(j).edgeColour = Color.BLUE;
				this.thisNode.myEdgeAL.get(j).visited = true;
			}

			this.nextEdge.active = false;
			this.nextEdge.edgeColour = Color.RED;

			// Determine if nextNode is an endNode
			boolean nextNodeIsEndNode = false;
			for (int enLoop=0; enLoop<=mycd.endNodeAL.size()-1; enLoop++) {
				if (this.nextNode.nodeIndex.equals(mycd.endNodeAL.get(enLoop))) {
```

```
                                nextNodeIsEndNode = true;
                        }
                }

                // If not an end node add it to the nodes hit
                if (!nextNodeIsEndNode) {
                        this.hitNodes.add(this.nextNode.nodeIndex);
                }
                else {
                        shortestMovesToEndNode = nextNode.moves;
                        mycd.endNodeFound = this.nextNode.nodeIndex;
//                      System.out.println("End node found: node=
"+mycd.nodeArray.get(this.nextNode.nodeIndex).myNodeInit.name+", moves= "+nextNode.moves);
                }

                this.thisNode.nodeColour = Color.BLUE;
                this.thisNode.visited = true;

        }
        else {
                // if not hit deactivate the edge
                this.nextEdge.active = false;
                this.nextEdge.edgeColour = Color.RED;
        }

        if (myPlayControl.showDetails) {

                        myPlayControl.addInfo(true, "- Processing node: "+this.thisNode.myNodeInit.name);
                        myPlayControl.addInfo(false, "- Virtual line position x:
"+this.walkersCalcVPosition(this.thisNode).x);
                        myPlayControl.addInfo(false, "- Virtual line position y:
"+this.walkersCalcVPosition(this.thisNode).y);
                        myPlayControl.addInfo(false, "- moves: "+this.thisNode.moves);
                gsonString = myGson.toJson(mycd);
                myPlayControl.strControlDataAL.add(gsonString);
                }

        this.walkersFindNextEdge(); // look for the next edge

        if (myPlayControl.showDetails) {
                        if (this.nextNode != null) {
                                myPlayControl.addInfo(false, "- Next node:
"+this.nextNode.myNodeInit.name);
                                myPlayControl.addInfo(false, "- moves: "+this.nextNode.moves);
                                myPlayControl.addInfo(false, "- Slack: "+(this.nextEdge.myEdgeInit.distance
+
                                                this.walkersCalcVDist(this.thisNode, this.nextNode)));
                                myPlayControl.addInfo(false, "- This to next node vDist: "+
                                                this.walkersCalcVDist(this.thisNode, this.nextNode));
                                this.nextEdge.edgeAdditionalText =
"("+df.format(this.nextEdge.myEdgeInit.distance +
                                                this.walkersCalcVDist(this.thisNode,
this.nextNode))+")";
                        }
                        this.thisNode.additionalNodeText = "(" +
df.format(this.walkersCalcVPosition(this.thisNode).x) + "," + df.format(thisNode.moves) + ")";
                        myPlayControl.addInfo(false, "- Number in hit nodes: "+this.hitNodes.size());
                        String hitNodesStr = "";
                        for (int f=0; f<=this.hitNodes.size()-1; f++) {
                                hitNodesStr =
hitNodesStr.concat(mycd.nodeArray.get(this.hitNodes.get(f)).myNodeInit.name+" ");
                        }
                        myPlayControl.addInfo(false, "- Hit nodes: "+hitNodesStr);

        }

//      ii++;
```

```
//                      if (ii>50) this.nextEdge = null;

    } // while (this.nextEdge != null)

                        myPlayControl.algorithmThreadRunning = false;


            }

            public void walkersFindNextEdge () {

                        // This method finds the next edge for processing in Walkers algorithm

                        double movesAndSlack;
                        boolean found = false;
                        Node myNode, neigbourNode;
                        Edge myEdge = null;
                        boolean activeEdgeFound;
                        ArrayList<Node> removeHitNodes = new ArrayList<Node>();

                        this.leastMovesAndSlack = null;
    this.nextNode = null;
    this.nextEdge = null;

    // loop through all of the hit nodes
                        for (int i=0; i<=this.hitNodes.size()-1; i++) {

                                myNode = mycd.nodeArray.get(this.hitNodes.get(i));
                                myNode.visited = true;

                                activeEdgeFound = false;
                                // loop through the edges connected to the hit node
                                for (int j=0; j<=myNode.myEdgeAL.size()-1; j++) {

                                        myEdge = myNode.myEdgeAL.get(j);
                                        myEdge.visited = true;

                                        if (myEdge.active) {

                                                neigbourNode =
                        mycd.nodeArray.get(myEdge.getConnectedNode(myNode.nodeIndex));

                                                // Calculate the moves and slack and look for a new edge with least slack
                                                movesAndSlack = myNode.moves + (myEdge.myEdgeInit.distance +
                                                        this.walkersCalcVDist(myNode, neigbourNode));

                                                if (shortestMovesToEndNode!= null && movesAndSlack >=
                        shortestMovesToEndNode) {
                                                        myEdge.active = false;
                                                        myEdge.edgeColour = Color.RED;
                                                }
                                                else {
                                                        found = false;

                                                        if (this.leastMovesAndSlack == null) {
                                                                found = true;
                                                        }
                                                        else if ( movesAndSlack < this.leastMovesAndSlack ) {
                                                                found = true;
                                                        }

                                                        if (found) { // Set the stored objects to objects in arrays so
                        reference won't change
                                                                this.leastMovesAndSlack = movesAndSlack;
                                                                this.thisNode =
                        mycd.nodeArray.get(this.hitNodes.get(i));
                                                                this.nextEdge = this.thisNode.myEdgeAL.get(j);
```

```
                                                this.nextNode =
mycd.nodeArray.get(myEdge.getConnectedNode(this.thisNode.nodeIndex));;
                                                }
                                        }

                                        activeEdgeFound = true;

                                } // if (myEdge.active)

                        } // for (int j=0; j<=myNode.myEdgeAL.size()-1; j++)

                        if (!activeEdgeFound) {
                                removeHitNodes.add(myNode);
                        }

                } // for (int i=0; i<=this.hitNodes.size()-1; i++)

                if (this.nextEdge != null) {
                        this.nextEdge.active = false;
                }

                // Remove any hitNodes which don't have an active edge
                for (int i=0; i<=removeHitNodes.size()-1; i++) {

                        this.hitNodes.remove(removeHitNodes.get(i).nodeIndex);

                }

}
```

```
// 2. Execute for A*Plus

ArrayList<AStarPlusEndNodes> aspenAL;
aspenAL = new ArrayList<AStarPlusEndNodes>();
AStarPlusEndNodes aspen;

double xDist, yDist;

Coord sn = new
Coord(this.myControlData.nodeArray.get(this.myControlData.startNode).myNodeInit.x,this.myControlData.
nodeArray.get(this.myControlData.startNode).myNodeInit.y);

// For each end node store in an array and calculate the distance from the start node
for (int i=0; i<=this.myControlData.endNodeAL.size()-1; i++) {
        aspen = new AStarPlusEndNodes();
        aspen.endNode = this.myControlData.endNodeAL.get(i);

        Coord cen = new
Coord(this.myControlData.nodeArray.get(this.myControlData.endNodeAL.get(i)).myNodeInit.x,this.myContr
olData.nodeArray.get(this.myControlData.endNodeAL.get(i)).myNodeInit.y);
        xDist = cen.x - sn.x;
        yDist = cen.y - sn.y;
        aspen.dist = Math.sqrt((xDist*xDist) + (yDist*yDist));

        aspenAL.add(aspen);
}

// Sort the end nodes into order of closest end nodes first
ShortestPathUtility.sortAspen(aspenAL);

double firstElementDistance = Double.MAX_VALUE;
shortestDistance = Double.MAX_VALUE;

aggregatedAlgorithmOutput = new AlgorithmOutput();

aggregatedAlgorithmOutput.edgesHit = 0;
aggregatedAlgorithmOutput.nodesHit = 0;
aggregatedAlgorithmOutput.totalHits = 0;
aggregatedAlgorithmOutput.aStarPlusNodesHit = 0;

// loop through the end nodes
for (int i=0; i<=aspenAL.size()-1; i++) {

        // only execute this block if the distance is less than the first element distance
        if (aspenAL.get(i).dist < firstElementDistance) {

                this.reset();
                this.myControlData.endNode = aspenAL.get(i).endNode;
                myAlgorithmOutput = thisWindow.executeAlgorithm("A*", false);

                if (i==0) {
                        firstElementDistance = myAlgorithmOutput.distance;
                }

                // check if a new shortest path has been found
                if (myAlgorithmOutput.distance < shortestDistance) {
                        aggregatedAlgorithmOutput.closestNode = myAlgorithmOutput.closestNode;
                        aggregatedAlgorithmOutput.distance = myAlgorithmOutput.distance;
                        shortestDistance = myAlgorithmOutput.distance;
                }

                aggregatedAlgorithmOutput.edgesHit = aggregatedAlgorithmOutput.edgesHit +
myAlgorithmOutput.edgesHit;
```

```
                    aggregatedAlgorithmOutput.nodesHit = aggregatedAlgorithmOutput.nodesHit +
myAlgorithmOutput.nodesHit;
                    aggregatedAlgorithmOutput.totalHits = aggregatedAlgorithmOutput.totalHits +
myAlgorithmOutput.totalHits;
                    aggregatedAlgorithmOutput.aStarPlusNodesHit ++;
//                  System.out.println("A*Plus" + myAlgorithmOutput.writeCsv());

            }

}
```

Creation of test data

```
public static void createCompareTestData() {

        ControlData mycd = new ControlData();
        mycd.reset();

        // SET UP DATA
        int maxNumEndNodes = 20; // The maximum number of end nodes in test data i.e. 1..n where n is
maxNumEndNodes
        int numberCases = 100; // The number of test cases for each number of end nodes e.g. 100 cases for 10
end nodes
        String graph = "France";
        double boundaryAngleAllowed = (Math.PI / (double) 2.0) * ( (double) 1.0 / (double) 2.0);

        // Read in base node data and populate node and edge arrays

        mycd.nodeArray = Node.getNodeAL(graph);
        mycd.edgeArray = Edge.getEdgeAL(graph, mycd.nodeArray, false);

        // Create connections in node array to edges

        Node.createNodeConnections(mycd.nodeArray, mycd.edgeArray);

        // Generate test data
        int numNodes = mycd.nodeArray.size();

        // loop until numberCases found

        int numCasesFound;
        int numCasesTried;

        TestData td = new TestData();
        TestDataInput tdi;
        td.maxNumEndNodes = maxNumEndNodes;
        td.numberCases = numberCases;

        for (int numEndNode=1; numEndNode<= maxNumEndNodes; numEndNode++) {

                numCasesFound = 0;
                numCasesTried = 0;
                tdi = new TestDataInput();

                do {

                        tdi = new TestDataInput();

                        // 1. Randomly select a start node
                        int startNode = ThreadLocalRandom.current().nextInt(0, numNodes);

                        Coord fromCoord = new
Coord(mycd.nodeArray.get(startNode).myNodeInit.x,mycd.nodeArray.get(startNode).myNodeInit.y);
                        Coord toCoord = new Coord (0,0);

                        Translate tn = new Translate (fromCoord, toCoord);

                        // 2. Randomly select a vl direction
                        int vlDirectionInt = ThreadLocalRandom.current().nextInt(0, 361);
                        double vlDirection = ((double) vlDirectionInt / (double) 360.0) * ((double) 2.0 *
Math.PI);

                        // 3. Determine upper boundary and lower boundary

                        double vlRotation;
                        if (vlDirection < (Math.PI / (double) 2.0)) {
                                vlRotation = (Math.PI / (double) 2.0) - vlDirection;
                        }
```

```
		else {
			vlRotation = (Math.PI * (double) 2.0) + (Math.PI / (double) 2.0) - vlDirection;
		}

		mycd.boundary1AngleRotated = (Math.PI / (double) 2.0) - boundaryAngleAllowed;
		mycd.boundary2AngleRotated = (Math.PI / (double) 2.0) + boundaryAngleAllowed;

		// 4. Loop and randomly select nodes and select any within the boundaries until criteria
met

		int numEndNodesFound = 0;
		int numTried = 0;
		int randomNode;
		Coord myCoord, myCoord2, myCoord3;
		boolean iwb;
		boolean alreadySelected;

		do {

			randomNode = ThreadLocalRandom.current().nextInt(0, numNodes);
			myCoord = new
Coord(mycd.nodeArray.get(randomNode).myNodeInit.x,mycd.nodeArray.get(randomNode).myNodeInit.y);

			myCoord2 = tn.translate(myCoord);
			myCoord3 = myCoord2.rotate(vlRotation);
			iwb = myCoord3.coordWithinBoundaries(mycd, false);

			// check random node selected is not the start node and also not already
been selected
			alreadySelected = false;
			for (int ii=0; ii<=tdi.endNodeAL.size()-1; ii++) {
				if (tdi.endNodeAL.get(ii) == randomNode) {
					alreadySelected = true;
				}
			}

			if (iwb && randomNode != startNode && !alreadySelected) {
//
"+mycd.nodeArray.get(startNode).myNodeInit.name+" -> "+mycd.nodeArray.get(randomNode).myNodeInit.name);
				System.out.println("start node:
				numEndNodesFound++;
				tdi.endNodeAL.add(randomNode);
			}
			numTried ++;

		} while (numEndNodesFound < numEndNode && numTried <= (numEndNode*1000));

		if (numEndNodesFound == numEndNode) { // Found a test case
			numCasesFound ++;
			tdi.startNode = startNode;

			// calculate nearest and farthest nodes
			double minDist = Double.MAX_VALUE;
			double maxDist = Double.MIN_VALUE;
			double dist;
			double xDist, yDist;

			Coord sn = new
Coord(mycd.nodeArray.get(startNode).myNodeInit.x,mycd.nodeArray.get(startNode).myNodeInit.y);

			for (int endNodeALloop=0; endNodeALloop<=tdi.endNodeAL.size()-1;
endNodeALloop++) {
				Coord cen = new
Coord(mycd.nodeArray.get(tdi.endNodeAL.get(endNodeALloop)).myNodeInit.x,mycd.nodeArray.get(tdi.endNodeAL.get(endNo
deALloop)).myNodeInit.y);
				xDist = cen.x - sn.x;
				yDist = cen.y - sn.y;
				dist = Math.sqrt((xDist*xDist) + (yDist*yDist));
```

```
                                        if (dist < minDist) {
                                                minDist = dist;
                                        }
                                        if (dist > maxDist) {
                                                maxDist = dist;
                                        }
                                }

                                tdi.maxDist = maxDist;
                                tdi.minDist = minDist;
                                tdi.medianDist = ((maxDist - minDist) / (double) 2.0) + minDist;

                                td.testCase.add(tdi);

                        }

                        numCasesTried ++;

//                      System.out.println("---------------");
//                      System.out.println("maxNumEndNodes: "+numEndNode);
//                      System.out.println("numFound: "+numEndNodesFound);
//                      System.out.println("numTried: "+numTried);
//                      System.out.println("numCasesFound: "+numCasesFound);

                } while (numCasesFound < numberCases);

//              td.testCase.add(tdi);

                System.out.println("****************");
                System.out.println("maxNumEndNodes: "+numEndNode);
                System.out.println("numCasesFound: "+numCasesFound);
                System.out.println("numCasesTried: "+numCasesTried);

        } // for numEndNode loop

        Gson myGson = new Gson();
        String myJson = myGson.toJson(td);

        try (PrintWriter out = new PrintWriter("testData.json")) {
            out.println(myJson);
        } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                System.out.println("ERROR - can't create json file");
        }

    }
```