Computing Reachability Graphs for Street Networks and Investigating Approximation of Reachability

CM3203 – One Semester Project (40 Credits)
5th May 2017

Author: Winston Ellis
Supervisor: Dr. Padraig Corcoran
Moderator: Dr. Frank C Langbein

Abstract

The project described in this report investigated Dijkstra's algorithm and how it can be used to compute reachability graph of street networks. It also describes a novel approximation procedure to compute reachability graphs, with scalability as a goal for large graphs which would take excessive time using Dijkstra's method.

Background information has been given on the techniques used for computing the reachability graphs. Analysis of methods to be used in the project have been provided and results have been presented to justify their use.

Acknowledgements

The author would like to express his sincere thanks to his supervisor Dr. Padraig Corcoran for his guidance and encouragement during this project.

Table of Contents

Table of Figures

1 – Introduction

1.1 – Electric vehicles and the future

Electric vehicles are becoming more established, efficient and more reliable in the current commercial market offering. The popularity is growing and the technologies behind the batteries and their travel distance are becoming more attractive to all levels of consumer purchasing power. In addition to this, the constant rise in price of fossil fuels due to the growing scarcity and taxation is also pushing the need for the growth of the use of electric vehicles. To establish a greater market share from the fossil fuel vehicle counterpart, a well organised and accessible network of electric charging stations should be readily available to further promote the viability of electric vehicles to potential and current users.

1.2 – Necessity of well-placed Electric Charging stations and difficulties

Like petrol stations, the accessibility for electric vehicles to charging stations must be adequate to be reached from most locations depending on the mileage of the vehicles. Currently electric vehicles do require charging more often than fossil fuel powered vehicles, which poses the challenge of requiring the placement of the charging stations to be more frequent on a road network compared to a petrol station. In addition to this, the electric charging stations will require the vehicles to be parked for some time as charging is not as instant as filling up a tank of fuel. Consumer level electric vehicles in 2017 have a range of 70 miles to 300 miles, depending on the cost of the model [1].

Some important characteristics of the locations of the charging stations:

- Frequently placed due to any potential requirement to have an emergency charge, also to accommodate overflow (see last point below)
- Accessible in a dense urban area
- Accessible for long journeys such as on a motorway
- To be placed in locations such that it can be accessible from no build zones e.g. housing areas or conservation areas
- Spacious charging stations to accommodate enough vehicles

1.3 – Two solutions in selecting Electric Charging station locations

To solve this problem, I will break it down by representing the street network as a graph of connected nodes. Nodes represent intersections of roads and the end of a dead-end road. The edges will represent the roads, where the edge weight is the road length. The reachability of each node will be calculated, this creates new edges from the node being evaluated to all the nodes reachable given a distance. Once this graph has been created, the dominating set can be calculated using this newly created graph to give a set of nodes that are either a dominating node or a neighbour to a dominating node.

Computing the reachability graph will require a path finding algorithm to evaluate which nodes are reachable or not. This can be simply achieved by utilising a breadth first search which will exhaustively traverse through all neighbouring nodes until the nodes are beyond the given distance limit set. This is an accurate way of calculating the graph, however it is computationally expensive the larger the graph and the greater the distance limit set. The number of the nodes evaluated will grow exponentially the greater this distance limit. The details on how the placement of the electric charging stations on a street network is derived, are described in section 2.3.1.

To overcome this problem in the project, I decided to try to exploit a combination of depth first search and the use of spatial tools such as KD Trees to approximate the reachability graph. A KD Tree is a data structure which orders data points by their coordinates which can be quickly queried to find nearest neighbours through depth first search. The goal is to reduce computation time and or computation resources so that the algorithm has scalability.

In conclusion, the two methods that will be developed are:

1. Utilising a modified Dijkstra's algorithm to search from the initial node and find all nodes reachable given a max distance
2. Utilising a KD Tree to find all nodes reachable given a distance and then use a depth first search in different directions from the centre to evaluate which nodes falling under the KD Tree query should be kept

2 – Background

2.1 – Existing Solutions

The problem for finding the reachability of nodes is a common problem for different contexts such as finding the reachability of locations for public transport where time is the constraint instead of distance, depending on the time of day which determines the traffic [2]. The problem for this project is simplified due to the constraint being distance which is not affected by time. However, this could be a consideration for an extension to the project. This referenced report mentions use of Dijkstra's algorithm and use of A* search. I will be incorporating the style of A* search in the approximation attempt in computing the reachability graph, see section 2.6.

This problem is also similar to routing internet traffic for networks to determine which is the shortest path from A to B to reduce communications latency. Once the reachability graph has been created, it can be used as a reference to plan efficient routes of packets. For the project, instead of using the reachability graph as a reference to route packets, it will be used as a reference to create the dominating set. The dominating set can also be used in this context of routing traffic so that the dominating set suggests where upgrades for the network can be placed, such as more capable routers to increase the throughput of packets.

2.2 – Street Networks represented with Graph Theory

For this project, the street networks used will be in a graph form. This is for the abstraction of the street network so that path finding algorithms can be used to calculate the reachability graph.

Graph: A weighted graph is denoted as $G = (N, E, w)$ where N is a finite set of nodes, E is a finite set of edges ($E \subseteq V \times V$) and w is the weight of an edge. Nodes are also referred to as vertices, and will be referred to as nodes in this report. An edge contains a head and a tail, where in the tuple $edge \in (u, v)$ the term u is the head and v is the tail. The graphs used in this report will be a undirected graph such that an edge from u to v implied that there is an edge from v to u [3]. Therefore, one way streets will not be considered in the calculations. This will simplify the problem for the current implementation but could be added in future work. The graphs will all have positive weights, these will be used as the length of a road in the street networks. The map locations that have been tested have minimal one way road systems, so this will not have a big impact on the final reachability graph.

Street Networks: The street network data is retrieved from OpenStreetMaps which contains lots of geographical information that can be utilised by algorithms. For example, the roads will be given properties such as defining if it is a one-way road or if it a pedestrian only road. The algorithms implemented for this project will only utilise the street length as the weight of the edges, therefore the graphs that are extracted from OpenStreetMaps will be simplified to reduce file size and algorithm run time by pruning unnecessary data.

The following figures illustrate the differences of the a street network represented as a graph and represented as a reachability graph.

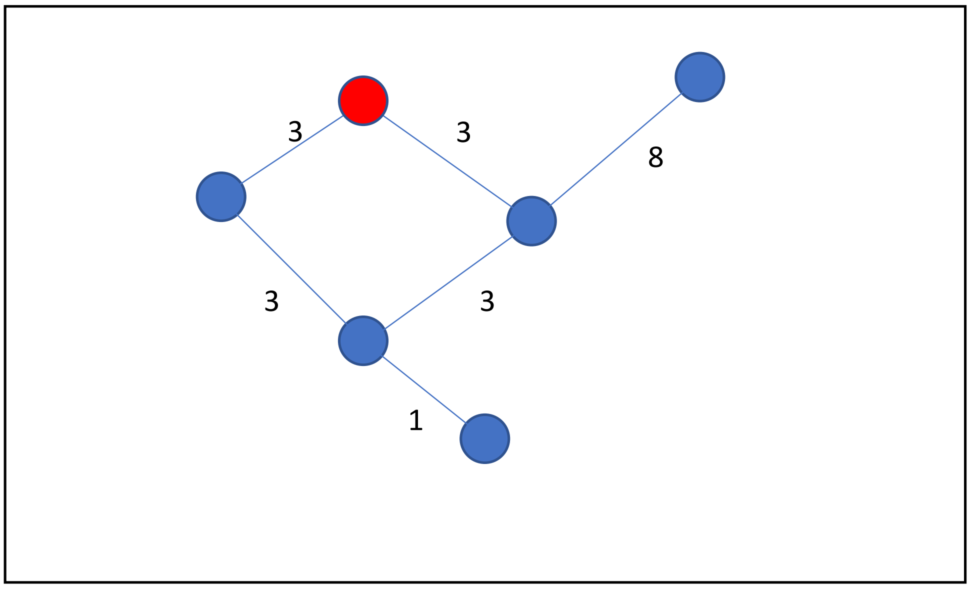

Figure 1 - Street Network Represented as a Graph

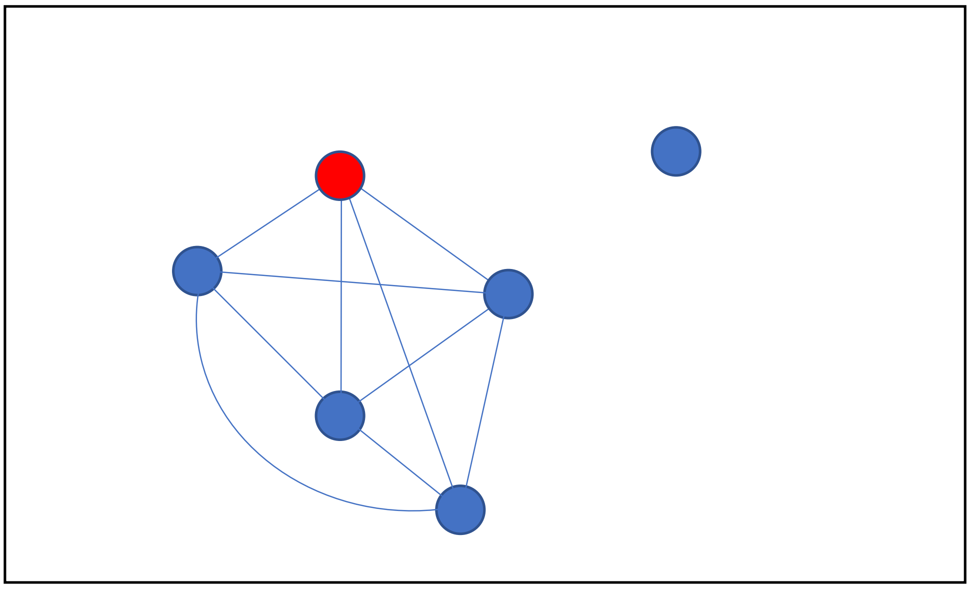

Figure 2 - Street Network Reachability Graph

Figure 1 contains a weighted graph of 6 nodes. If this were a street network, the weight would be the street length. If a reachability algorithm were to be applied to that network with a reachability distance of 7 starting at the red node, the resulting graph is shown in Figure 2. The graph becomes disconnected as the node on the right is not reachable.

2.3 – Reachability of nodes in a graph

The definition of the reachability states that a node B is reachable from A if:

- B is the same node as A

- There is an edge from node B to A
- There exists node C such that node C is reachable from A and B is reachable from C

This is used to determine if another node can be reached in the graph or how many steps it takes [4].

The reachability of nodes in a graph is found by giving the searching algorithm some criteria of maximum distance that can be travelled. The algorithm should check as it is traversing neighbouring nodes in its list of nodes that it has not visited to see if the current node distance plus the cost of traversing the edge will pass over the max distance threshold. Every node that is visited and is under the max distance threshold will have a new edge added from the initial node to the node visited. This connects the initial node directly to all the visited nodes within the max distance threshold. The new connectivity is necessary for calculating a dominating set. See Figure 1 and Figure 2 for a visual description.

2.3.1 – Dominating Set

From source [5] **Definition 7.1:** (Dominating Set). Given an undirected graph $G = (V, E)$, a dominating set is a subset $S \subseteq V$ of its nodes such that for all nodes $v \in V$, either $v \in S$ or a neighbor u of v is in S.

Computing the smallest dominating set is not straight forward to get the perfect set of dominating nodes. The computation of this problem is defined as NP-hard. This means that to have an acceptable runtime to solve this problem, approximation algorithms can be used. These algorithms are said to be optimal to a certain factor [5]. The downside to this implementation is that the dominating set can differ for each run depending on the sequence of the nodes that are chosen.

For this project, the dominating set will be computed after all the nodes in a given street network have their reachability graph computed and combined into one graph. This highly connected graph will then have the dominating set computed, which will be the suggestion of the electric vehicle charging stations.

2.4 – Dijkstra’s Algorithm

Dijkstra’s algorithm is a shortest path algorithm that maintains a priority queue of active nodes with tentative distances. This priority queue orders the active nodes with the smallest tentative distance at the front with ascending tentative distances to the end. Initially only the initial node is the active node and its neighbours are added to the queue. The tentative distance of the initial node is set to 0 and all other nodes are set to positive infinity. As the neighbours are checked, their distance from the initial node replaces the positive infinity distance and are ordered in the priority queue. The algorithm then iterates through the priority queue evaluating each node at the front of the queue by checking if it is the goal node or adding its neighbours to the priority queue. The algorithm can adjust the distances of each node if there is a shorter path to the node being evaluated by relaxing them. The algorithm will stop when it reaches the goal node.

The list of visited nodes as the algorithm iterates grows in a circle around the initial node, essentially as a circle until the goal node is reached. The algorithm does not know in which direction to search so it searches in all possible directions. This is a property of breadth first search. At a given time during searching, since the algorithm iterates though all closest neighbours in turn, the shape of the search space will resemble a circle.

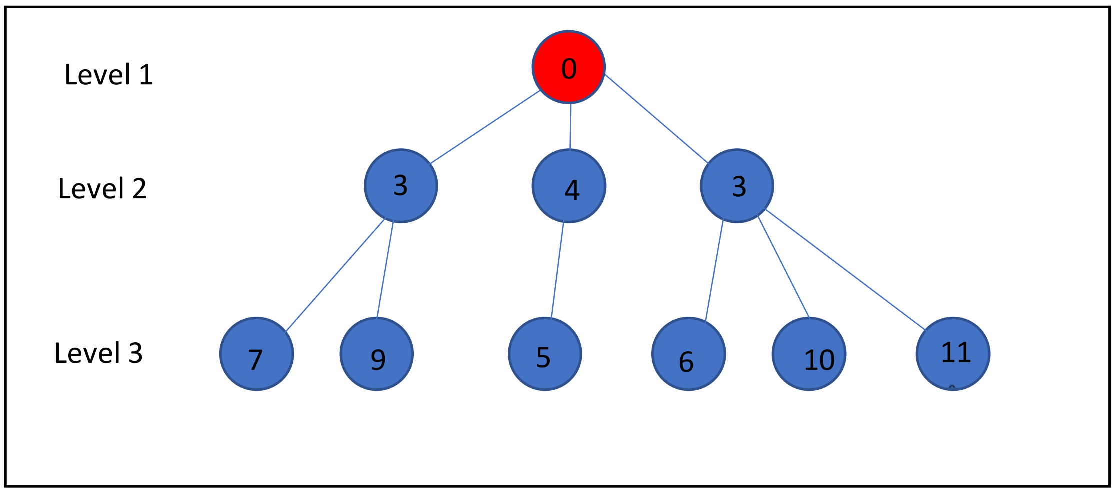

Figure 3 - Dijkstra's Algorithm Search Space Represented as a Tree

In Figure 3, the tree of an example Dijkstra's algorithm search shows the neighboring nodes of the red node, which is the initial node. Level 2 contains all the neighbors to the initial node and level 3 contains all the neighbors to the nodes in level 2. The numbers inside the nodes represent the cost to get to the node from the initial node.

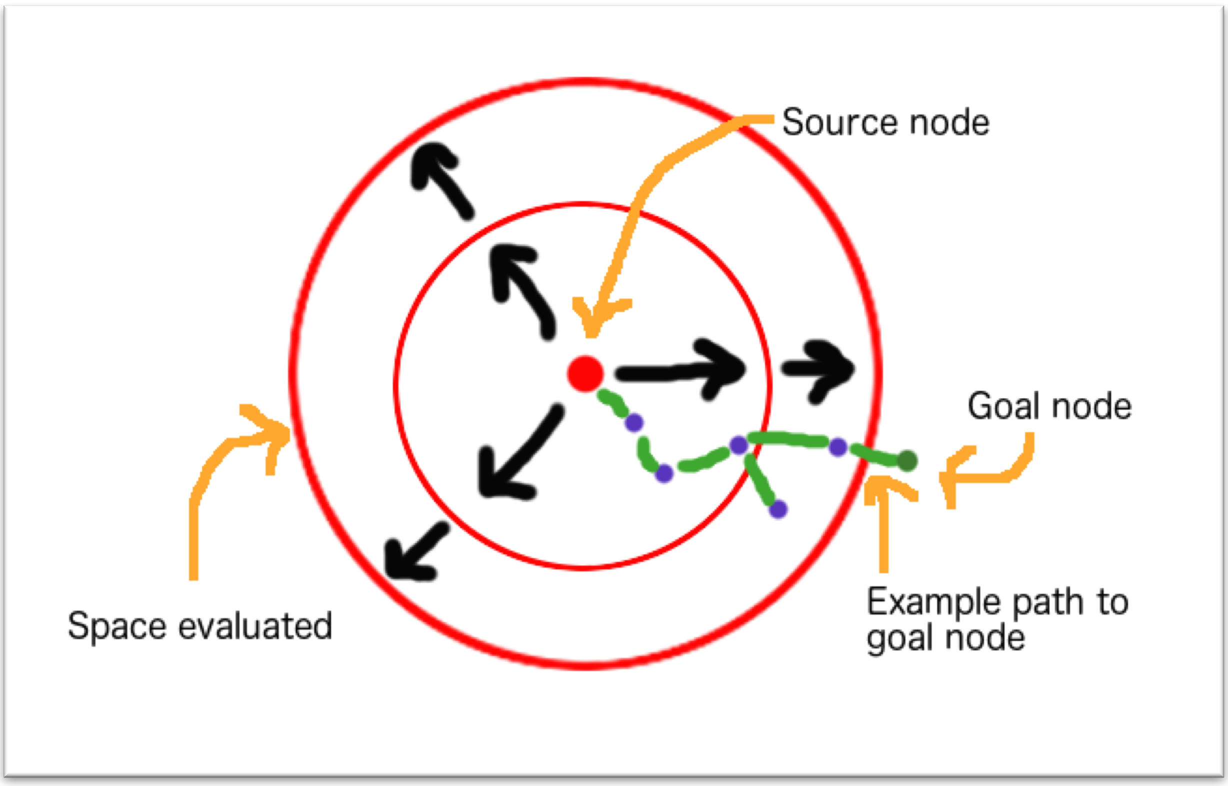

Figure 4 - Dijkstra's Algorithm Spatially Represented

Figure 4 shows how the space evaluated grows outward as it evaluates the paths in all directions until it eventually reaches the goal node, giving the shortest path to the goal node. A potential path has been shown with green edges and purple nodes. The branch that does not lead to the goal node would have had a shorter tentative distance initially but as the algorithm iterates it would have evaluated the other node which is a neighbour to the goal node.

Dijkstra's Algorithm Find the shortest path [6]   input: (graph G, initial node v)

```
For each node in G:
        distance[node] = positive infinity
        previous[node] = null
distance[v] = 0
Q = set of all nodes in G                              //priority queue
while Q is not empty:
        u = Q.pop(node in Q with smallest distance)
        for each neighbour n of u:
                T = distance[u] + distanceBetween[v + u]
                if T < distance[v]:                    //relaxation
                        distance[v] = T
                        previous[v] = u
```

output: previous (connected graph of distances between all nodes searched)

Project specific modifications: The property of Dijkstra's algorithm where it evaluates nodes spatially in all directions from the initial node is quite useful in computing the reachability graph. Changing the criteria of stopping from reaching a goal node to just evaluating the tentative distance, and keeping a set of the nodes that have a tentative distance smaller than the max distance threshold. The necessity of keeping an adjacency list that records the actual path to the goal node is not required. This greatly simplifies the implementation of the algorithm.

See section 3.2 for the pseudocode for the modified Dijkstra's algorithm.

Dijkstra's algorithm is a breadth first search algorithm, it will expand all nodes at a level before traversing to the next level of nodes. This property is useful as it will ensure that all nodes are visited and none are skipped, essential for computing the reachability graph. This property comes at a cost, the algorithm will require a lot of memory resources as the search level increases. The scalability of this algorithm is quite poor when searching through large graphs.

2.5 – A* Shortest Path Finding Algorithm

Dijkstra's and A* search are both mentioned in the report referenced in [2] to be used in computing the reachability graph. A* search uses a heuristic to direct the search spatially, this means it should evaluate less nodes compared to Dijkstra's algorithm using more of a depth first search. The A* algorithm is useful when trying to find the shortest path without searching in areas which obviously do not lead you directly to the goal node and are misleading to search algorithms.

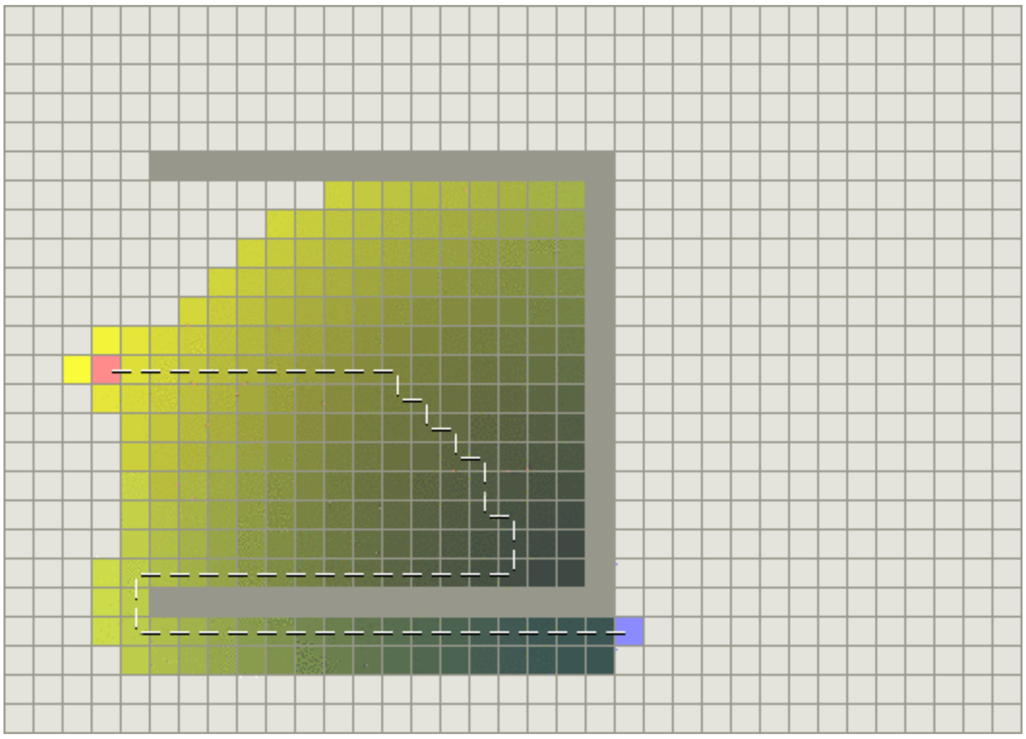

Figure 5 Greedy Search Diagram [7]

As can be seen from this picture, a concave obstacle will fool some search algorithms and cause it to take a long path to the goal node. This is an example of a greedy algorithm.

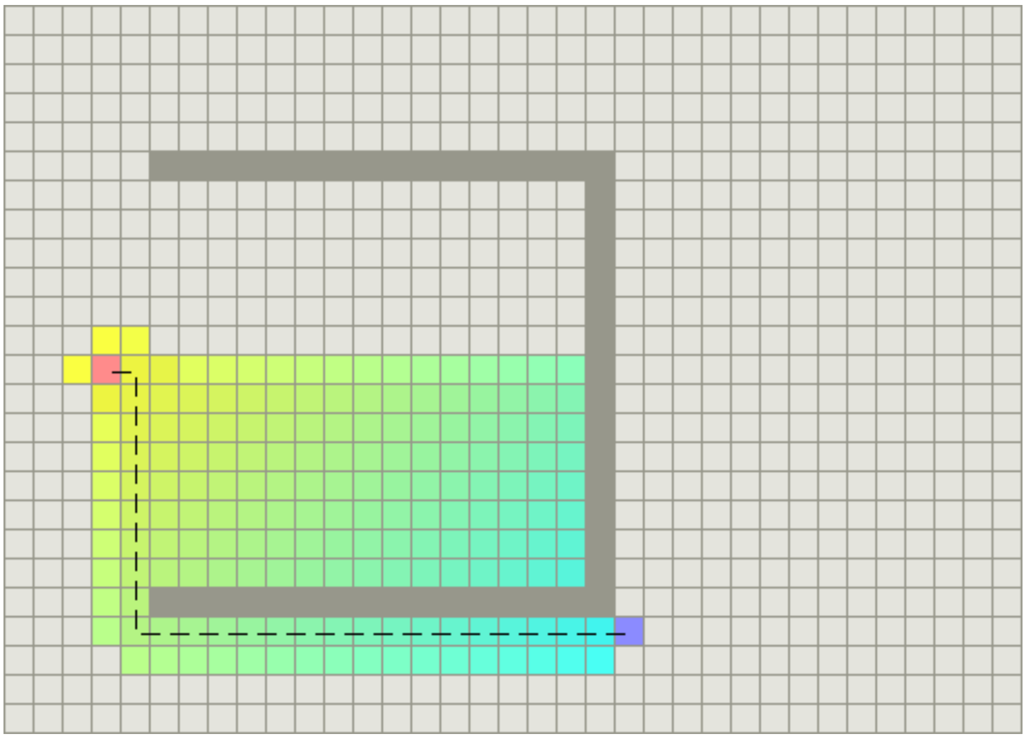

Figure 6 A Search Diagram [7]*

As can be seen here, the A* algorithm can work out that there is an impassable area and will still give the shortest path.

A* search computes its search path at each node by computing the value for each neighbouring node using the equation where N is the node being checked:

$$f(N) = g(N) + h(N).$$

In this equation $g(N)$ is the current cost of the node and $h(N)$ is the heuristic cost. The algorithm will choose the neighbouring node from the current active node with the smallest cost given by $f(N)$. The heuristic cost is calculated prior to running the algorithm. This can be performed by calculating the actual cost for every pair of nodes or if that is unfeasible for large graphs, then an approximation approach can be taken. The approximation approach can utilise waypoints, which can be nodes with high connectivity that the search is likely to traverse many times during different searches. A heuristic does not need to be precomputed if the graph does not have many obstacles, however the street networks used for this project will have such obstacles as geographic obstacles mentioned above [7].

Project Specific Modifications (Greedy Heuristic Search): Precomputing the heuristics for the street networks used would not be suitable as this process alone would take time, whereas one of the aims of this project is sensitive to timing of the computation of the reachability graphs. In addition to this, precomputation is only applicable to individual street networks and the reachability algorithm is applied to different graphs.

To avoid precomputation, the algorithm will use the Euclidian distance to the goal node as the heuristic. As the algorithm checks the neighbouring nodes to see which path it should take, it will choose the node that has the smallest Euclidian distance to the goal node. This heuristic directs the search in the direction of the goal node in a depth first search manner. Termination of this algorithm is shared with A*, where if the goal node is reached then it stops. Another termination check is added, if the distance travelled as it traverses nodes reaches the reachability distance e.g. 3Km, then it shall stop.

Greedy Heuristic Search Find goal node using Euclidian distance as heuristic
input: (graph G, initial node v, goal node u, reachability distance **maxDistance**)

```
frontier = heapQueue( neighbours($v$) )
visited = set(∅)
closestNode = v
furthestDistance = maxDistance
while frontier is not empty
    currentNode = frontier.pop()
    if currentNode in visited
        continue
    if currentNode <= maxDistance
        for each neighbour n of currentNode
            heuristic = Euclidian distance of current node to goal node u
            if heuristic < furthestDistance
                furthestDistance = heuristic
                closestNode = n
            frontier.push(n)
```

output: closest node to goal ***closestNode***
distance of node which had smallest euclidian distance to goal node ***furthestDistance***

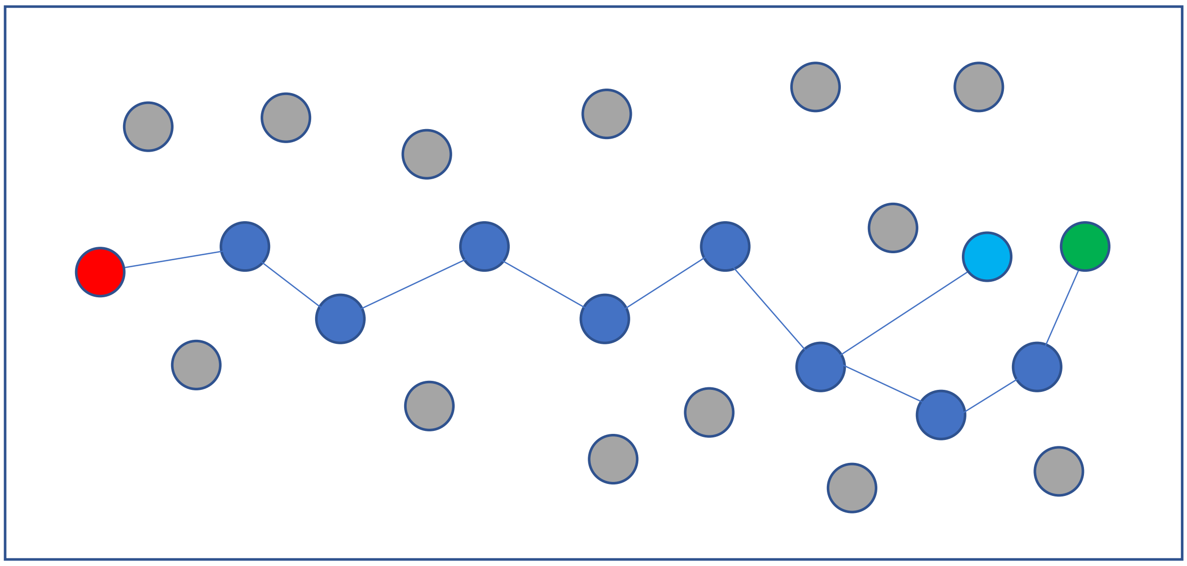

Figure 7 - Greedy Heuristic Search Example

In Figure 7 the path from the red initial node to the green goal node is shown. The Algorithm traverses through the dark blue nodes until it reaches the goal. The light blue node is an example of the heuristic leading the path finding greedily to the goal node, however the light blue node is not connected to the goal node. The algorithm will then pic the next closest node and continue until the termination criteria is met.

2.6 – KD Trees

For the approximation algorithm, a method to find nodes without evaluating them individually is required. A KD Tree query works in the same style as Dijkstra's algorithm spatially where from the node being queried, the search will look out in a direction up to a specified distance in a circle. See Figure 4 in section 2.5.1 for a visual representation. This data structure is used to reduce computation time.

A KD Tree works by splitting the list of coordinates alternating by the x-coordinate and y-coordinate for a 2 dimensional KD Tree, therefore alternating on the dimension attributes of the dataset. This will therefore on the first split, put half of the x-coordinates on the left and the other half on the right then following the same principle with the y-coordinates. Simply, it must look at all the values for the x coordinates in the data given and split it at the median putting the lower half on the left and the other half on the right. The next split will be on the y coordinates, finding the median for the left half and then splitting that in half and then finding the median of the right half and splitting that in half. By doing this, a tree is created which can be more efficiently queried compared to individually searching for the nodes that fit the search criteria, explained below in the **Nearest Neighbour Search** section. The tree created will have not have a depth greater than Log_2N where N is the number of data points.

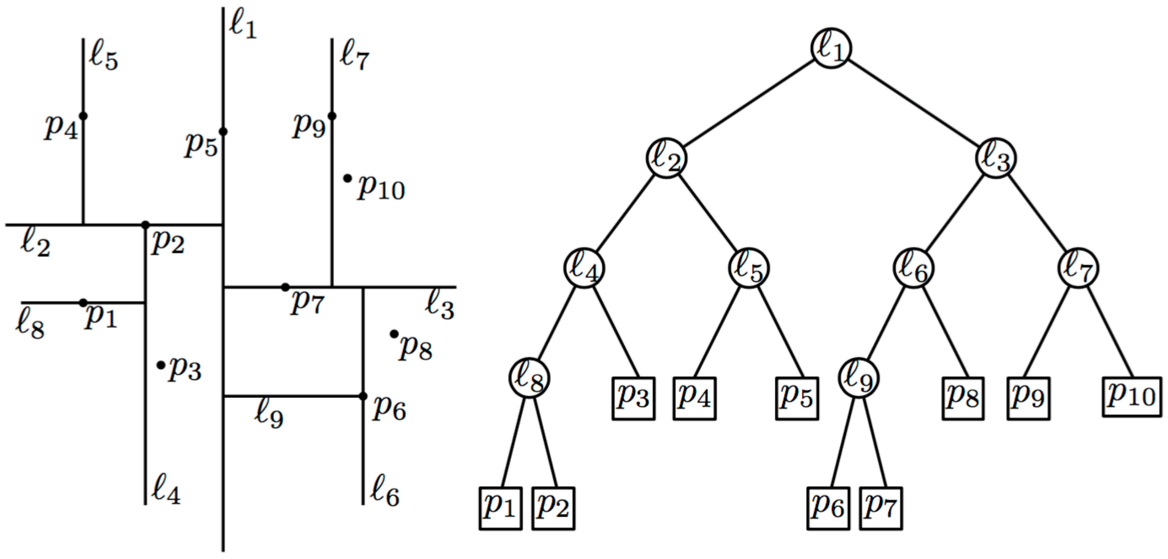

Figure 8 - KD Tree x,y-coordinate split on the left and tree representation on the right [8]

Figure 8 illustrates spatially how the KD Tree splits up coordinates. The lines denoted by L are the level splits which split the coordinate by half in turn and the points denoted by P show the coordinates that are at the end of the nodes where the splitting occurs. The first level is split on point P_5 which has the median x coordinate value for the dataset. On the second level L_2, it is split on P_2 which is the median y coordinate and for L_3 which is on the same level as L_2, is split on P_7.

Nearest Neighbour Search: On the right-hand side of Figure 8, the tree shows the points P as the leaf nodes. When a query for nearest neighbour search is used on this data structure, the point will be compared to the values on each level to determine how the query will traverse the tree. When it reaches the leaf node(s), the value given to the query will then compare to these value(s) to see which is the nearest neighbour. This tree structure provides logarithmic searching speeds, $Log\ N$ [9].

However, there is one downside to this data structure, the query can sometimes not find the actual nearest neighbour. When the search traverses the tree and reaches the lowest level containing the potential nodes, the actual nearest neighbour may be in a neighbouring branch.

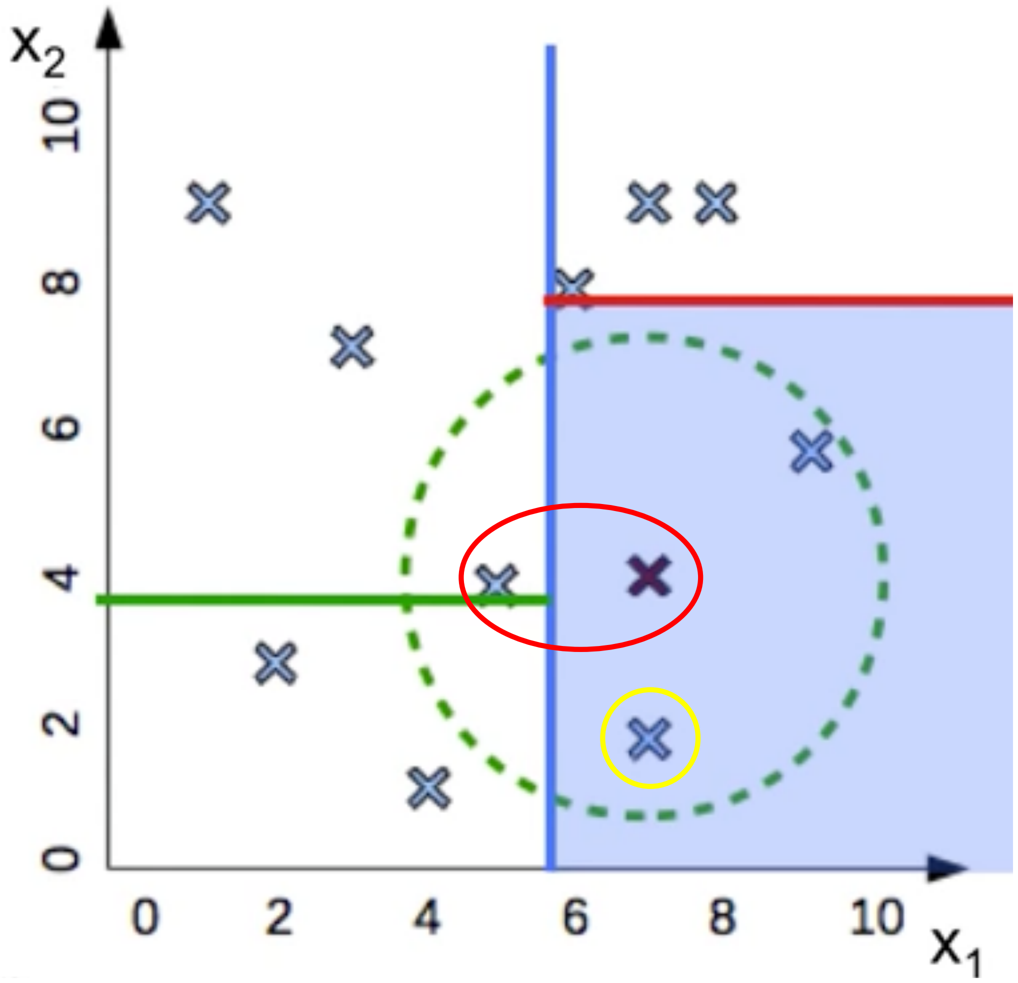

Figure 9 - KD Tree failing to find correct nearest neighbor [10]

In Figure 9, the red circle shows the actual nearest neighbor to the black X which is the point being queried to find the nearest neighbor. The KD Tree will traverse down the tree and end at the level highlighted in blue. The nearest neighbor to the query point is the point circled in yellow. The amount of nodes and their spatial density for the data points used in this project, this shortcoming of nearest neighbor search will not affect the results of the approximation results. If the nearest neighbor search finds the wrong closest neighbor, the node is likely to be very close and for the purpose this search is used for is not necessary to find the exact closest neighbor, see section 3.3 for the use of nearest neighbor search at the arc midpoint.

Nearest Neighbour Range Search: To find all nearest neighbours in a radius around the centre point, an r-Nearest Neighbour query needs to be computed on the KD Tree. This requires the search algorithm to traverse down the tree to find the left most node in the tree which is at the border of the range specified. It must also find the rightmost node at which is at the border of the range specified. All nodes in between these two will be returned in the query.

2.7 – Projections and conversions

The coordinate system for the OpenStreetMaps use latitude and longitude, which is referenced as a projection with the code WGS84 [11]. Projections are used to model the surface of the Earth, many projections are required as the surface of the earth is not perfectly spherical, it is an oblate spheroid. In addition to this, the surface of the earth is not consistent as the surface height compared to sea level is different for every area. To more accurately model areas closely, there needs to be projections for different areas of the Earth. The projection EPSG:27700 [12], which models the United Kingdom will be used. This projection preserves Euclidian distance, which will be used extensively in the approximation of the reachability graph [13].

Converting the street network from latitude and longitude coordinates before computing the reachability graph is required for the KD tree data structure. As the poles are approached, the meridians of the longitude become closer together eventually converging at the poles. See Figure 10, the red arrow pointing at the meridians show them converging at the pole. So for two coordinates of latitude and longitude that share the same latitude value and differ only in longitude, the actual distance between the two coordinates would be different if measured at the equator compared to close to the poles.

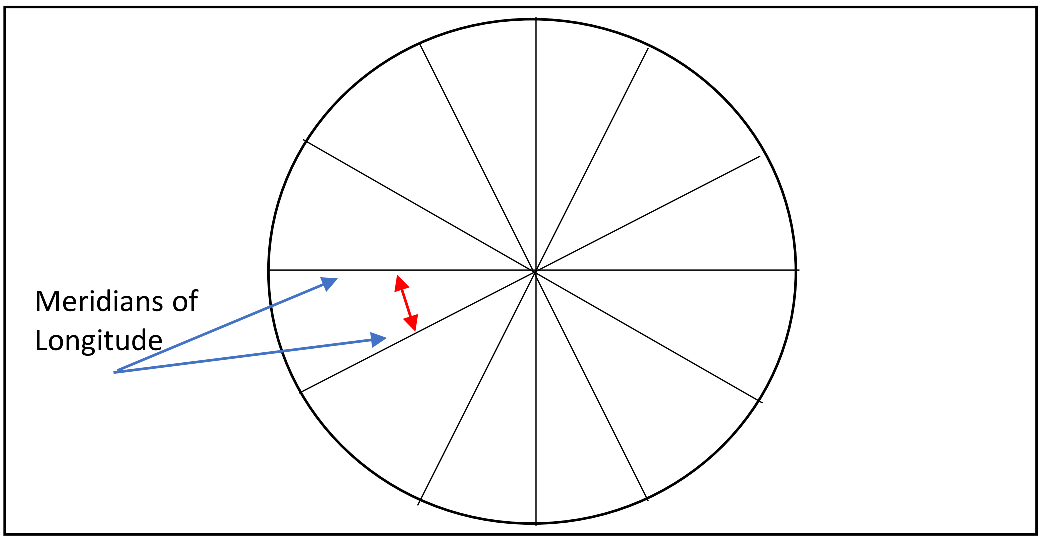

Figure 10 – View of Earth above a Pole demonstrating converging Longitude

This property would affect the accuracy of the KD tree, therefore the conversion before the algorithm starts will be necessary. In addition to this, converting from latitude and longitude during the computation of the reachability graph as necessary would slow down the algorithm. This conversion at the start is a set constant.

2.8 – Complexity of Algorithms and Big O Notation

Complexity of an algorithm, the measure of how an algorithm scales as the problem gets larger, is an important consideration for this project. Complexity is expressed in Big-O notation as a function O of the problem size N. Some examples of Big-O notation:

- a constant-time method is "order 1": $O(1)$
- a linear-time method is "order N": $O(N)$
- a quadratic-time method is "order N squared": $O(N^2)$

Big-O notation do not have constants or low order term as these will not affect the time as much as the higher order terms do [14].

2.9 – Aims

In this project, several goals have been identified. The goals cover computing the reachability graph and the use of mathematical and technical approaches to achieve approximation and optimisation. The main aims listed below are in order of ascending complexity:

1. Compute a reachability graph for a street network using a modified Dijkstra's algorithm

2. Compute a reachability graph through approximation using a KD Tree

3. Compute a reachability graph through approximation using a KD Tree with a combination of algorithms and mathematical theory to more accurately define the graph and with optimisation of the solution

3 – Design

3.1 – Programming Language and libraries used

Python has a large range of scientific libraries that can be used to implement this project. They provide most of the code required. However, it has been necessary to recode some of these algorithms to meet the project modification specifications where necessary.

1. NetworkX
    - Provides tools to create and manipulate graphs and has a method to work out the dominating set
2. OSMnx
    - Provides a lookup to OpenStreetMaps to extract a street network and save it in GraphML format for NetworkX
3. Heapq
    - Provides the heap queue algorithm to create, push and pop elements
4. SciPy
    - Contains the KD Tree data structure used implemented in C
5. CartoPy
    - Reads Shapefiles to be used in conjunction with matplotlib to visualise the algorithms implemented for testing and to visualise the dominating set
        - Shapefiles for Cardiff and London obtained from *mapcruzin.com*
6. PyProj
    - Provides conversion between projections used
7. Numpy
    - Used for trigonometric methods
8. Matplotlib
    - Provides plotting tools to visualise the outputs of the algorithms

3.2 – Solution 1 - Modified Dijkstra’s Algorithm

Modifying Dijkstra’s algorithm for this project is relatively simple. The requirement for the algorithm is to keep track of all nodes visited that are under the max distance threshold given. There is no need to keep track of the path to each node visited, only the set of active nodes is required. The set of active nodes is kept by the priority queue which this algorithm will keep querying for the element at the front of the queue.

A priority queue is an abstract data structure, this data structure can be implemented with the required properties by using a Heap Queue. A heap queue is different to a normal queue by instead of ordering elements by “first in first out” it orders the elements by priority. For this project, the priority is going to be the smallest tentative distance from initial node to the active node being evaluated (see 2.5.1).

In the search, there will be nodes that share neighbouring nodes. This means that these nodes will be added to the heap queue multiple times with only the tentative distance differentiating them. Since we only need to see if a node has been visited once, duplicates will not matter and do not need to be removed from the heap queue. This would have been computationally expensive since the heap queue would have had to be remade each time there was a duplicate removed.

This algorithm is run inside a for loop which will evaluate all the nodes in the given street network to calculate the reachability of each node. In this loop, the modified Dijkstra’s algorithm is given the initial node and the street network as a graph and returns the set of reachable nodes. A nested loop will go through this set and add edges from the initial node to these nodes to connect them all.

Once the first loop has iterated through all the nodes in the street network, the reachability graph is constructed and the dominating set can be calculated. At this stage, the first aim has been achieved.

Modified Dijkstra's Find all reachable nodes within a set distance (initial node v)

```
frontier = heapQueue( neighbours($v$) )
reachable = set(Ø)
while frontier is not empty
    currentNode = frontier.pop()
    if currentNode in reachable
        continue
    if currentNode <= maxDistance
        reachable.add(currentNode)
        for all neighbours of currentNode frontier.push(neighbouring nodes)
    else
        break
return reachable
```

3.3 – Solution 2 - Greedy Heuristic Path Finding Algorithm using KD Tree

This algorithm is more complex than the modified Dijkstra's algorithm, all the steps and justifications for each method used to approximate the reachability graph are discussed below.

A graph must be created containing the nodes in the street network with coordinates for the projection used converted from latitude and longitude. After this a KD Tree containing all these needs to be initialised. This KD Tree will be queried many times by the main loop looking for all potential reachable nodes from an initial node to then be processed to remove the nodes that the algorithm does not suggest that is reachable.

The algorithm will now loop through all the nodes in the street network to approximate the reachable nodes. For each node in this loop, a range query is used to find all nearest neighbours in a certain distance. This distance is set just above the average distance that the modified Dijkstra's algorithm finds. So, if the algorithm was given 3km as a max distance through roads, in Euclidian distance the furthest reachable nodes were found to be around 2.2km. This is a fair assumption to make because due to the bends in roads in street networks the outermost nodes will not also have the same Euclidian distance. This assumption is used to reduce the number of nodes that need to be evaluated later in the algorithm that are checked if they fit inside the polygon constructed which approximates the reachable nodes.

Up to this point in this algorithm, the second aim of the project has been achieved. The next section incorporates the extra techniques to give a more accurate approximation. After comparing the reachability for a small sample of nodes using the modified Dijkstra's method and the tree that the KD Tree approximation produced, there were some interesting results. The geography of the area around the initial node will cause the shape of reachable nodes for the modified Dijkstra's method to not be a circle. Geography such as rivers and parks that have no street network connectivity through them means that nodes beyond this geographical obstacle will not be reachable, but the KD Tree approximation method still suggests such nodes as reachable.

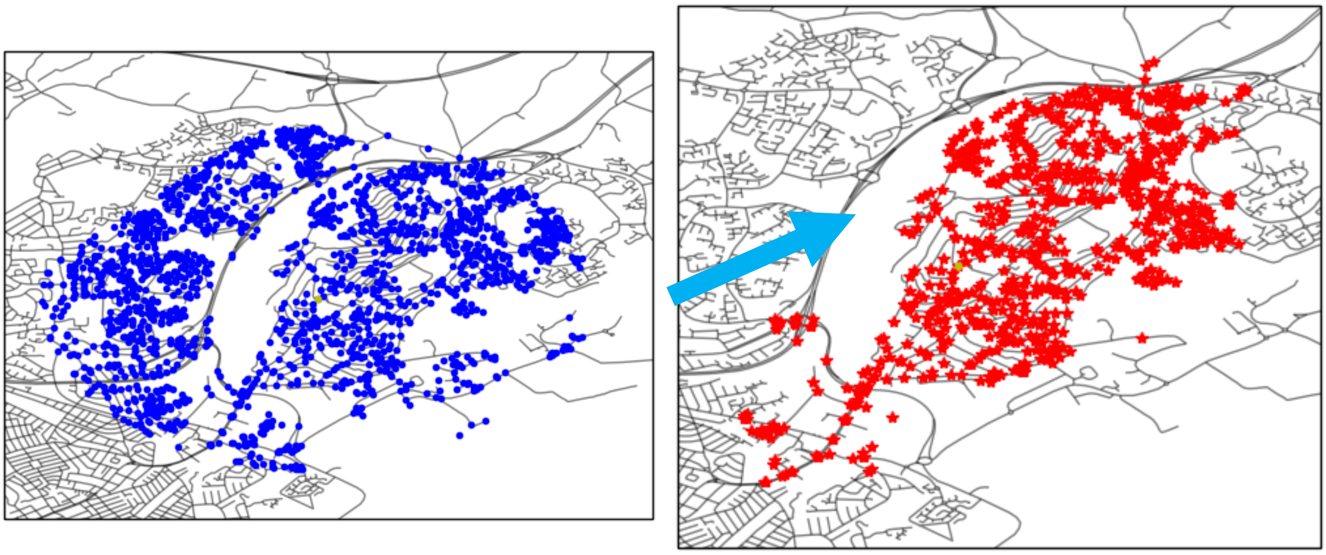

Figure 11 - Geographic Obstacle Example 1

In Figure 11 on the right is the reachable nodes of a sample node next to a geographic obstacle (see blue arrow) produced by the modified Dijkstra's algorithm. On the left, shows the reachable set of nodes produced by a KD tree.

To tackle the problem, the nodes that the KD Tree return as reachable must be validated. This will be done by splitting the circle shape of nodes returned into 8 sectors and individually checking how far from the initial node the radius of the sector should be. This can be done by using the Greedy Heuristic Search from the initial node, traversing nodes until it reaches the outermost node in the theoretical sector or if it reaches the max distance that the modified Dijkstra's algorithm uses, e.g. 3km. This outermost node is defined as the nearest neighbour to the midpoint of the arc in each sector.

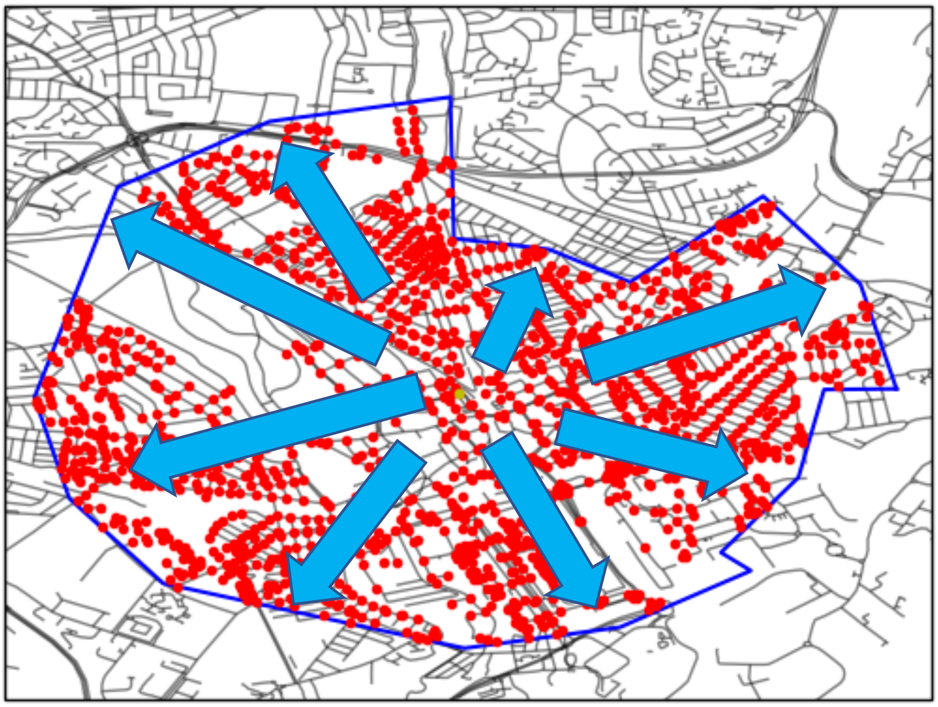

Figure 12 - Sector Validation Example

In Figure 12, the blue arrows show the different radii of the 8 sectors from the center node.

After considering on how to use A* search for this validation section, it was decided that computing heuristics would be too computationally expensive for this method since the street network is so complex. Simplifying A* search to just evaluate the Euclidian distance to the goal node had to suffice, the simplification became the Greedy Heuristic Search algorithm. See below for the algorithm pseudocode and see section 2.6 for the modifications that took place to derive this algorithm. Using this greedy algorithm one the other hand, means that it can be subject to the problems mentioned in section 2.6 of impassable areas. However, this can be justified as acceptable because the purpose of this search algorithm is not to find the shortest path to the goal node, but to see how close the search gets to the goal node.

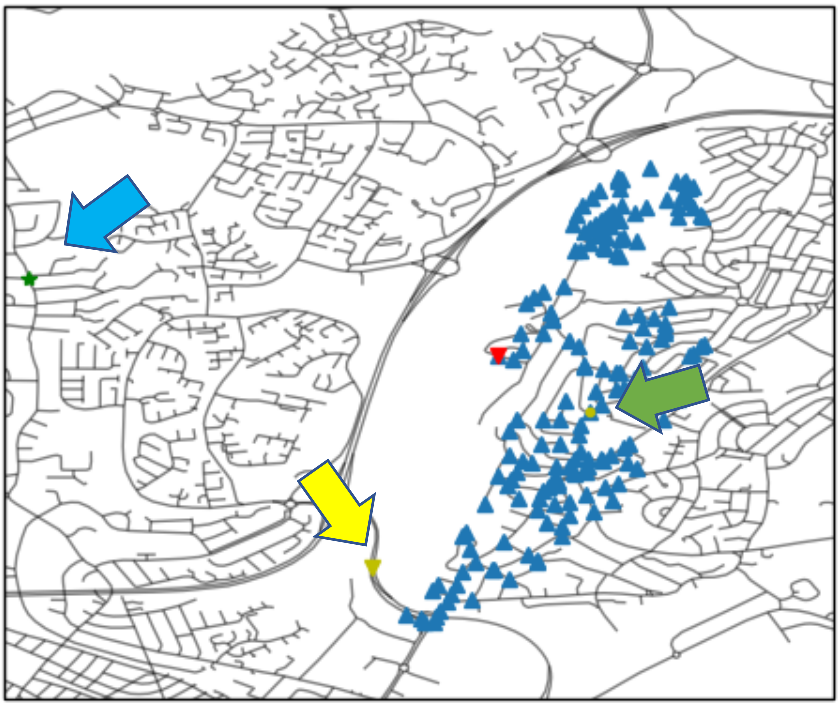

Figure 13 - Geographic Obstacle Example 2

In Figure 13, the Greedy Heuristic search is shown to get stuck in the geographic obstacle. The nodes in blue are the nodes that it has checked to reach the goal node. The red node shows the node that was closest to the goal node, which is the green star on the left of the diagram (see blue arrow). The yellow triangle represents the node where the algorithm terminated when it achieved its termination criteria of max distance traversed (see yellow arrow).

If this depth first search is successful in reaching the outer most node, a sector has been successfully approximated without having to traverse through all nodes in a depth first search manner saving computation time. If the algorithm does not reach the outermost node, two node locations are saved. One is of the node that has the closest Euclidian distance to the outer goal node. The second is the last node evaluated before the search hits the max distance threshold. The algorithm will then check which of the two is closer to the goal node and whether this node is inside the sector being evaluated (see Figure 13).

The closest distance of the two nodes saved will be used to calculate the new radius for the sector from this node to the initial node. Further to this, the check to see if the nodes end up inside the sector will help determine if the furthest node ended up being close to the goal node but only by going through another sector. This will change the shape of the sector, shown below using the closest and last node. In Figure 13, the distance to the initial node (see green arrow) to the node which had the smallest Euclidian distance to the goal node is quite small. If this distance was used as a radius, it would correctly approximate the nodes in that sector. However if the Euclidian distance between the initial node and the node which was at the end of the longest graph traversal (see yellow arrow) was to be used as a radius for the sector, it would approximate the nodes behind the geographical obstacle which would be incorrect.

Once each sector is evaluated, a polygon is created using the coordinates of the arcs of the sectors (the two ends and the midpoint) showing the bounds of the approximated area that is reachable. The nodes that lie inside should be added as reachable to the initial node. A point in polygon algorithm is used for this, see section 3.5.1. Before doing this, a polygon smoothing algorithm is utilised (see section 3.5.2) to remove the sharp edges that are created by the arc coordinates when neighbouring sectors have greatly different radii. This has been added because the shape of Dijkstra's algorithm as it grows spatially, seen in the figure below, gives the reachability graph has a smooth profile. The areas covered by the sectors after they have been validated do not share this smooth profile characteristic. The objective is to emulate Dijkstra's algorithm, so by smoothing the polygon there will be sets of nodes that will be included as reachable and sets that will not be included.

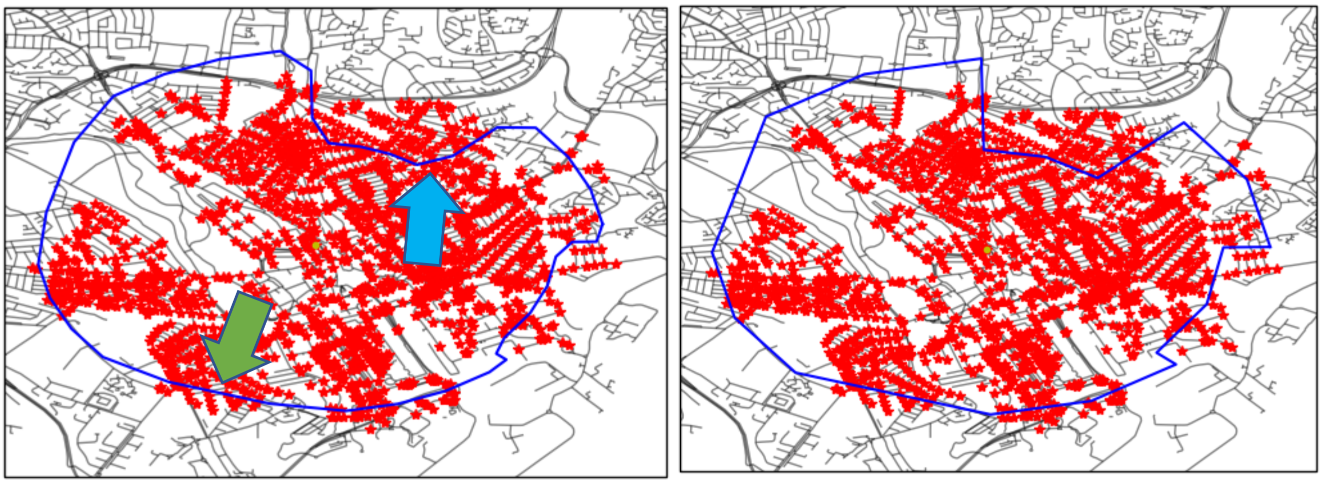

Figure 14 - Polygon Smoothing Example

In Figure 14, there are two cases where the smoothing algorithm is a positive addition to the approximation algorithm. The blue arrow points to a location where before smoothing is a obtuse angle, but after smoothing the polygon will now include extra node and improve the accuracy of the approximation. The green arrow points to a line which would have been straight, but the smoothing allows the shape of the polygon to be more curved. This would add a few additional nodes.

The nodes which lie under the corners cut which are pointing outwards from the centre will not be included. The nodes which lie next to the corners pointing inwards, will be added.

Now the approximated reachable nodes are connected to the initial node and the algorithm repeats this process for all the nodes in the street network being evaluated.

In approximating the reachability graph, there are several stages where performance can be lost due to the extra steps in refining the approximation. The polygon smoothing is a luxury but can improve the results.

3.4 – Dominating Set

The calculation of the dominating set uses a function from the library NetworkX. The functionality of the algorithm is described in section 2.3.1.

3.5 – Auxiliary algorithms

3.5.1 – Point in Polygon

Once the final polygon is constructed from the validation process of each sector, the nodes that lie inside this polygon from the KD Tree ball query are the approximated reachable nodes. This polygon is used as a mask over the nodes which is processed by matplotlib.contains_point to return True or False for each node. If True, then the node is added to a set of reachable nodes which can then be used to connect edges from the initial node to these nodes.

The code implementation for this is provided by matplotlib.Path library which has an underlying C implementation. This helps keep the computation time down as C is much quicker than Python.

3.5.2 – Polygon Smoothing

To remove the sharp corners and make the mask smoother and similar to the area that Dijkstra's algorithm would cover as discussed in section 3.3, a smoothing algorithm has been used. The algorithm was found on the internet called Chaikin's Corner-cutting Algorithm [15]. This algorithm is used as it is simple to keep computation time down.

It simply takes the coordinates of the neighbouring coordinates in the order that the algorithm is given assuming that they are in the order to make a closed set of edges. The coordinates of the polygon given to this algorithm are passed in order so this is not an issue. Before Chaikin's algorithm is called, the first coordinate is added to the end of the list to ensure there is a closed set of edges.

Given a control polygon $\{P_0, P_1, \ldots, P_n\}$, refinement takes place by deriving a new sequence of points $\{Q_0,\ R_0, Q_1, R_1, \ldots, Q_{n-1},\ R_{n-1}\}$ that are of a fixed ratio from the control points P. These points are defined as:

$$Q_i = \frac{3}{4}P_i + \frac{1}{4}P_{i+1}$$
$$R_i = \frac{1}{4}P_i + \frac{3}{4}P_{i+1}$$

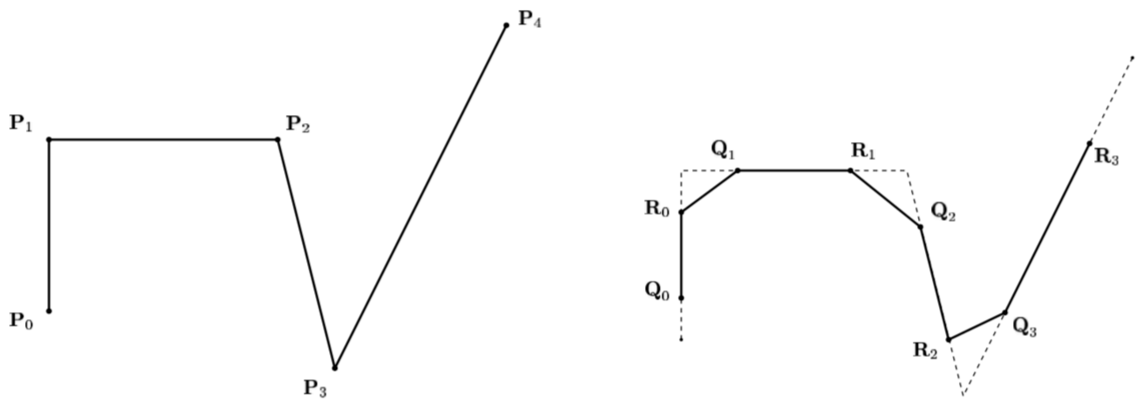

Figure 15 - Chaikin's Polygon Smoothing Diagram [15]

Figure 15 illustrates points Q and their relative location between the control points Q and R.

These new coordinates will be the new smoothed polygon that is used by the point in polygon algorithm used in section 3.4.1. Optionally, this algorithm can be applied repeatedly on the output polygon further smoothing the polygon. For smoothing in this project, the algorithm has only been applied once.

3.6 – Evaluating the results of the two algorithmic implementations

The reachability graphs will be computed for the target street networks, followed by the dominating sets that will suggest the nodes which are suitable locations to place electric charging stations. The distance that will be used for reachability needs to suit the criteria mentioned in section 1.2 where the stations need to be frequent. This means that the reachability distance cannot be 70 miles for example, as this would make the stations infrequent and assume that the use of the vehicles would be for long distance. This also includes the problem of congestion in the stations due to the cars requiring a minimum amount of time to charge so there might not be enough charge points in one station. Therefore, a small reachability distance needs to be used. In this project, the reachability graphs will be computed for a reachability distance range of 3km to 7km in 500m steps.

The reachability graphs for the modified Dijkstra's method will have their dominating sets examined to see how they change as the reachability distance increases. Dijkstra's algorithm is a known best solution to compute the reachability of a graph, the results do not need to be compared to another source. On the other hand, to evaluate the results from the approximation method, the reachability for individual nodes need to be compared to the modified Dijkstra's method. This is simply a comparison of the nodes that the modified Dijkstra's algorithm has computed to be reachable for a given node, then see if the approximation method has also selected the same nodes. The set of nodes chosen for the approximation method can then be checked to see how many nodes it has missed if it under approximates and to see how many extra nodes it has approximated. The average difference in the two sets can then be measured to see how accurate the approximation method is.

There is one downside to this method of measuring accuracy, since it only counts the difference in nodes as a number. This does not evaluate the accuracy of the approximation algorithm around geographic obstacles and how well the validation step has dealt with such obstacles (see section 3.3).

3.7 – Evaluating the performance of the two algorithmic implementations

The complexity of the two algorithms will be calculated and then compared to real test results. The timing of how long each solution takes to produce the reachability graph will be recorded for the two different variables that can be changed:

- The size of the graph (number of nodes)
- The reachability distance

Calculating the complexity of the two algorithms will be specific to the variable that is being changed above. The following text will be an evaluation of the code implemented and how it will affect the respective complexity.

The algorithms will be tested on two locations, Cardiff and an area of London. The area of London chosen has a dense street network. This means that spatially there are lots of nodes in a given area. Compared to Cardiff, for the same area there are less nodes. Since Cardiff is a small city, the graph quickly turns less dense as you leave the centre with more housing and green nature areas away from Cardiff city centre. Node count for the different graphs used:

- London 16km^2 nodes: 18593
- London 32km^2 nodes: 62840
- Cardiff 16km^2 nodes: 9566

This allows the testing of how well the two algorithms perform with different reachability distances depending on the node density. It is expected that the number of nodes increases exponentially as the reachability distance evaluated increases. The complexities of the algorithms are predicted below, to further test these predicted complexities I have included the only a larger London map (32km^2) and not one for Cardiff as the (32km^2) size graph did not have comparable increase in nodes.

Algorithm Complexity for Graph Size: To evaluate the complexity when changing the size of the graph, the reachability distance must be kept constant. The complexity of both algorithms should be $O(N)$ because the algorithms will always traverse through a fixed size subset of the graph up to the reachability distance, the only change is how many times it performs this. Therefore, for this section the complexity predicted for each algorithm are:

- Modified Dijkstra's Algorithm: $O(N)$
- Approximation method: $O(N)$

Algorithm Complexity for Reachability Distance: To evaluate the complexity when changing the reachability distance, the graphs used must be kept the same. The changing factor that will determine the complexity of these algorithms is how far and how long the algorithms have to traverse a graph.

Firstly, the modified Dijkstra's algorithm which is a breadth first search will exhaustively traverse all the levels in a tree. This means that the number of nodes it traverses will grow exponentially as the reachability distance increases. In the algorithm implementation, all the code that is required to compute the reachability graph to produce a complexity term in Big-O notation has been evaluated. After removing all the constants, the complexity remains as $O(N^2)$.

Secondly, the approximation method was designed to behave like a depth first search. There are steps in this algorithm which will add a base constant time to the algorithm, these should be removed from Big-O notation but is quite apparent in the results, see section 4.4.

In the code implementation of this algorithm, there are many sections of code that will always run N amount of times independent of the reachability distance. The sections of code that do depend on the reachability distance are the Greedy Heuristic Search and the Point in Polygon algorithms. The Greedy Heuristic Search will be performed $8N$ times per initial node and the Point in Polygon algorithm will also be performed $8N$ times. This results in $O(8N + 8N)$, simplified in steps:

- $O(8N + 8N)$
- $O(N + N)$
- $O(2N)$
- $O(N)$

Therefore, for this section the complexity predicted for each algorithm are:

- Modified Dijkstra's Algorithm: $O(N^2)$
- Approximation method: $O(N)$

The approximation method is expected to outperform the modified Dijkstra's method. However, as mentioned previously the approximation method has a base constant runtime. Due to this, the approximation method is expected to be slower in less dense graphs or with a small reachability distance.

4 – Experimental Results and Evaluation

During testing of the algorithm to obtain results, the algorithms were giving inconsistent timings that ranged in up to 2 hours to compute the reachability graph for the street network of London with a bounding box of 16km^2 with a reachability distance of 3km and up for both algorithm implementations.

To find the problem, the algorithms were first checked to see where they were slowing down. Initially, the algorithms were performing fast then slowing down when computing the reachability of a single node. The nodes where the algorithms slowed down were checked to see if they caused any loops in the logic, which was not the case. Individually, the reachability for these nodes was computed and it was found that it was not slow. This also lead to the discovery that the nodes which were problematic were not always the same for every run. Checking the memory usage of python lead to the discovery that Python was using many gigabytes of memory, up to 50GB. The computer being used to test the code only has 8GB of memory. To make sure that it was not inefficient code or any bugs, the code was extensively debugged and variables cleared such as the list of active nodes for each search once completed, were deleted to free up memory. This did not resolve the problem, so knowing that there are millions of edges being added to the graph, the creation of edges onto the graph was disabled. The memory usage was then smaller and compatible with the computer's physical memory, which therefore allowed the algorithms to run very quickly without memory bottlenecking the performance. The problem lies in the implementation of NetworkX due to it using memory heavy data structures to load graphs onto memory.

To work out how much memory the algorithm needs, the current implementation file sizes were investigated. The *.graphml* file for the 16km^2 area of London is 21.4mb with around 47,000 edges. The algorithm when it finished, added around 120 million edges so working out how many megabytes it takes to save one edge and multiplying it by 120 million will mean the graph requires 50,000MB or 50GB of storage to save the reachability graph as text. This value can be compared to the memory requirements of NetworkX, since it stores most the graph data as a *String* datatype inside *Dictionaries* which requires much more memory than if they were stored as integers. The IDs for the nodes are around 9 characters, if these are saves as Strings then each ID will take up a byte per character so 9 bytes in bits is 72bits. If these were represented as numbers, then 9 digits should take up 30 bits if it was 999999999. In addition to this, NetworkX also stores metadata such as length and directionality which also adds to the space required to store each edge. Assuming the savings in moving from a *String* to numbers applies to the metadata then the graph would take much less space, however the size would still be large and would still cause problems when trying to compute even large reachability graphs. The size of the graph being used is 16km^2 and the whole of London is 1,572 km^2. In the future work section, methods on how to solve this storage requirement problem will be discussed.

The following test results for performance only measure how long it takes for the algorithms to find all the reachable nodes, not adding the edges to the graph additionally. After investigating the memory usage for NetworkX, it seems that this is a limitation of the implementation of NetworkX as it stores lots of information using dictionaries. Improvements to this memory usage shortcoming will be discussed later.

4.1 – Reachability Graph Computation Evaluation

The following analysis contains figures with red stars. The red stars are nodes that are part of the dominating set.

Modified Dijkstra's – London – 16km^2: Firstly, the reachability graphs computed by the modified Dijkstra's algorithm as mentioned in section 3.6 will produce accurate results as Dijkstra's algorithm is the known best solution. The dominating sets produced by the reachability graphs will be visually examined. As the reachability distance increases, it is expected that the distance between the dominating nodes will increase and the number of these dominating nodes will decrease.

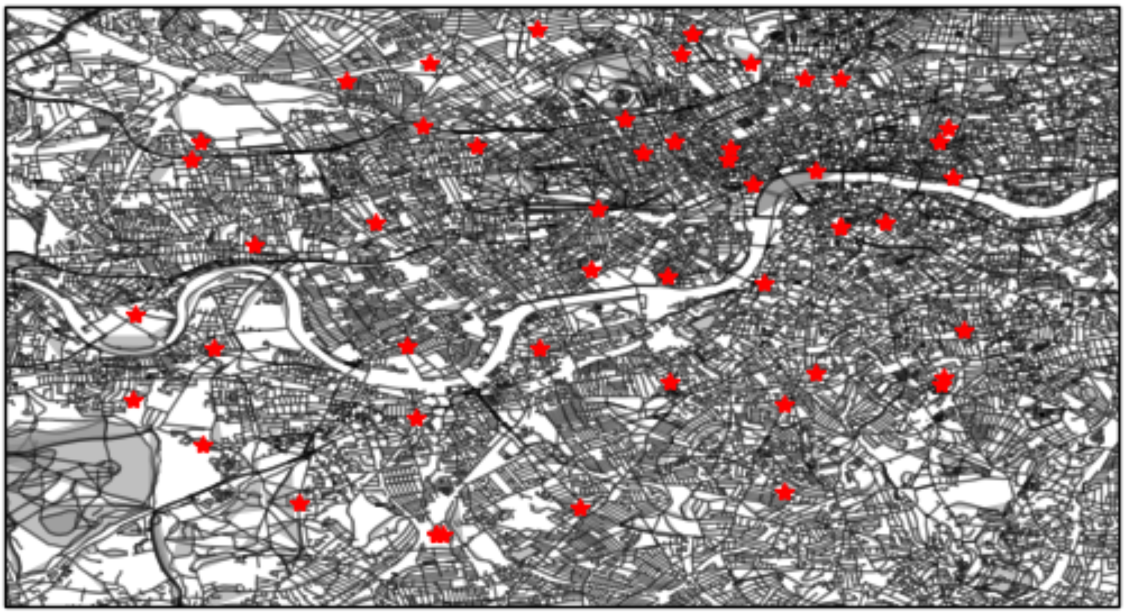

Figure 16 - Dominating Set for London - Dijkstra's - Reachability 3km

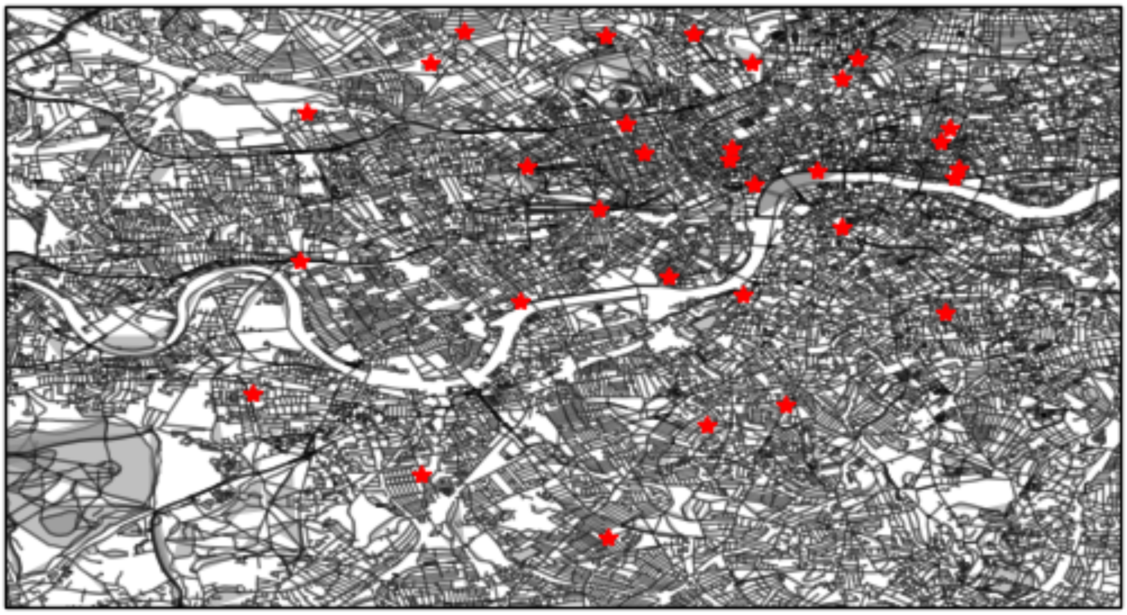

Figure 17 - Dominating Set for London - Dijkstra's - Reachability 5km

Comparing Figure 16 and Figure 17, there are a few similarities such as the location of some dominating nodes. Some of these are in the same place in both figures. These nodes are clustered on the top right, where it also seems to be a denser street network area. Another observation is that in the center, the distance between the dominating nodes is bigger in Figure 17, confirming that the nodes had more reachability. In addition to this, there are less nodes on the perimeter of the map.

The nodes in the street network which lie on the boundary for the map will only have their reachability calculated towards nodes in the center. This is a shortcoming of calculating the reachability for all nodes in the map. This means the nodes at the top of the graph will only have half it's reachability computed. This goes for all nodes near the edge of the street network. Using a bigger street network or only evaluating the nodes in the street network which have a Euclidian distance to the edge of the map that is greater than the reachability distance.

Approximation Method – London – 16km^2: For the same map as modified Dijkstra's method, the reachability graph has been computed using the approximation method. The dominating sets will be visually examined to see if they are similar to those chosen by the dominating set of modified Dijkstra's method. See below in the section (**Reachability Graph Comparison Using a Sample Node**) to evaluate the accuracy of approximation method.

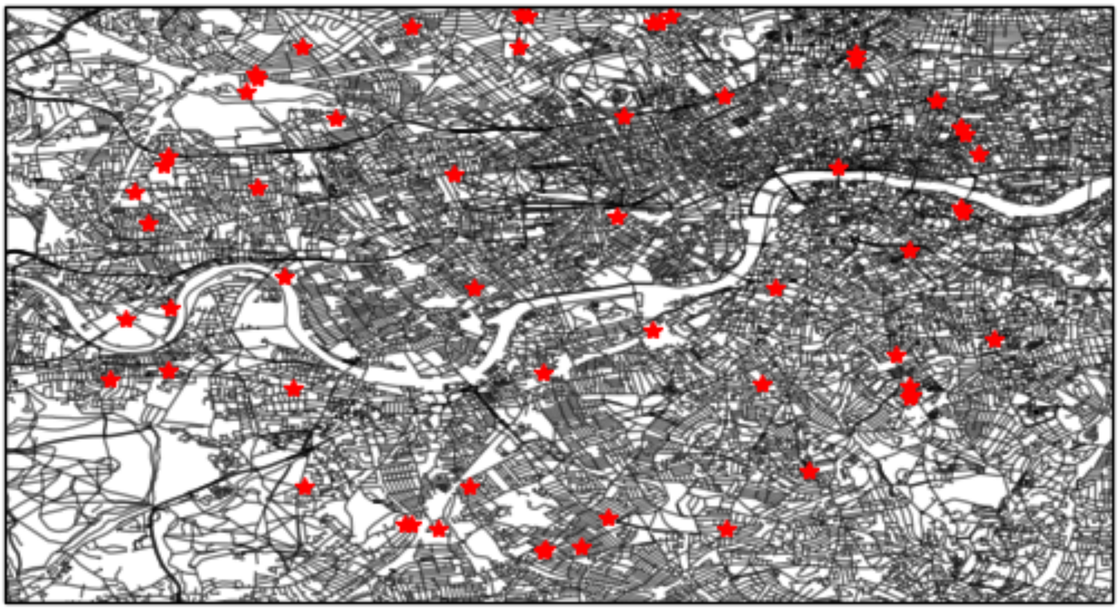

Figure 18- Dominating Set for London - Approximation - Reachability 3km

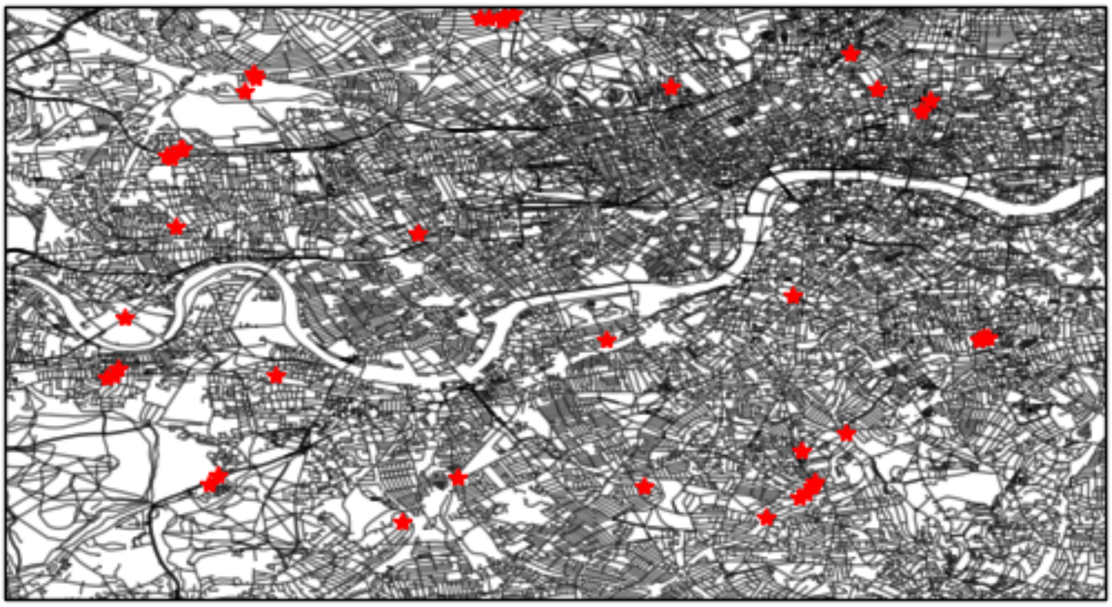

Figure 19 - Dominating Set for London - Approximation - Reachability 5km

The first observation that can be made when comparing Figure 18 and Figure 19 is that for the bigger reachability distance, the distance between the dominating nodes in the middle is very large. There are only 4 in the middle for 5km reachability whereas there are around 10 for the reachability of 3km. Comparing these to the modified Dijkstra's method shows that the dominating sets computed are quite different. However, they both share one property of having bunches of dominating nodes on the outside. This is due to the nature of the computation of the dominating set. Referring to the dominating set algorithm, the nodes are taken randomly and then nodes are added and removed as fit until the dominating set criteria has been met. The algorithm does not check to stop bunching of nodes in the dominating set.

Modified Dijkstra's – Cardiff – 16km^2: The following two figures illustrate the dominating sets for the street network of Cardiff. The obvious difference of the two networks can be seen visually as Cardiff is much smaller. In section 4.2, the performance of the search algorithms are analysed as the network size increases but shows that the increase in runtime decreases. The figures below show how the density of the nodes decreases around Cardiff.

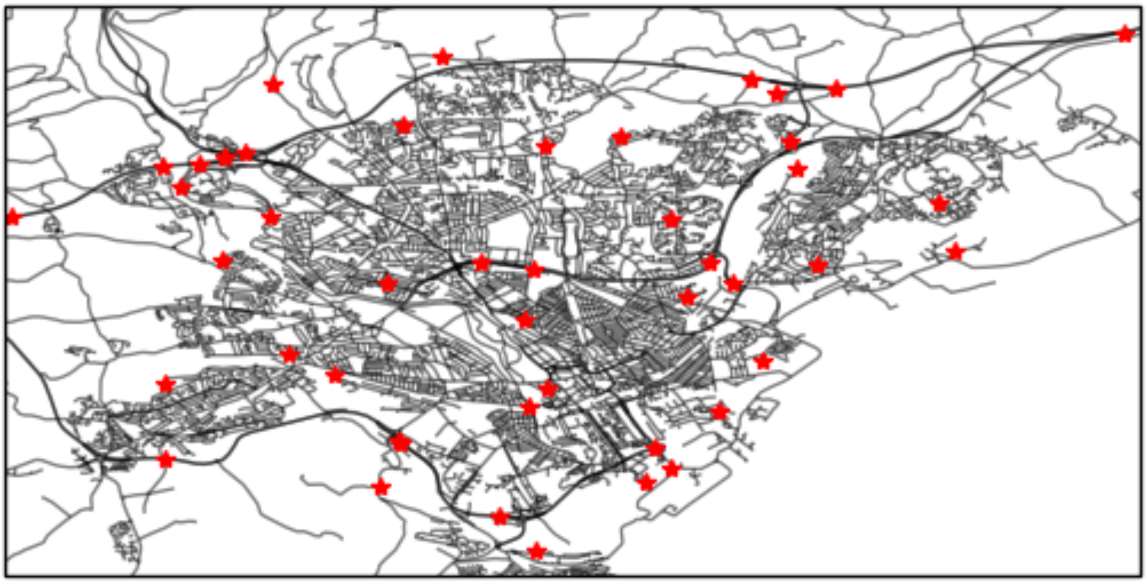

Figure 20 - Dominating Set for Cardiff - Dijkstra's - Reachability 3km

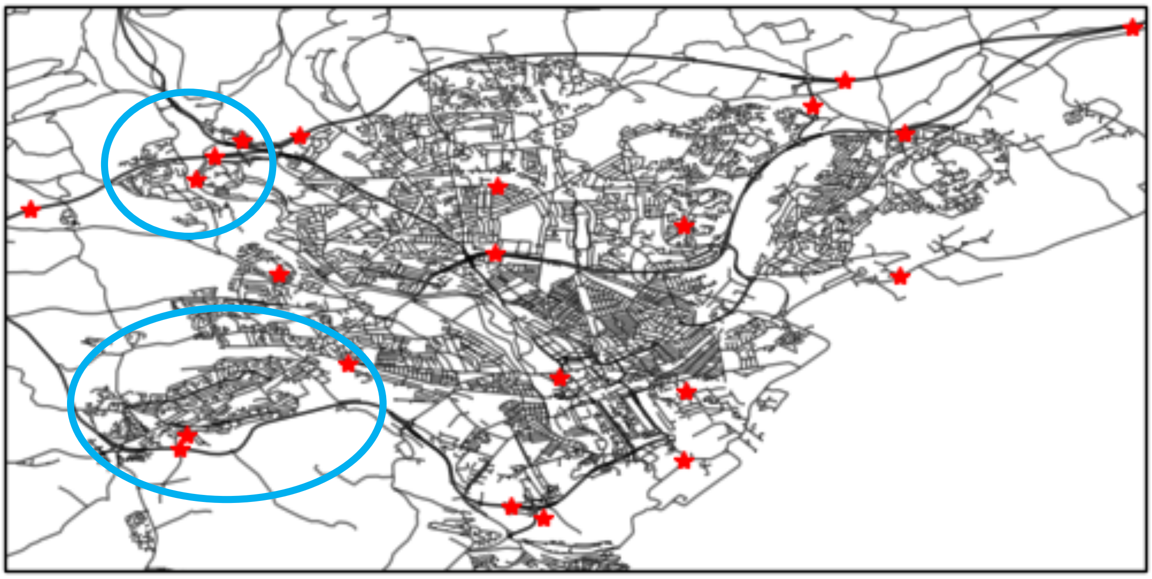

Figure 21 - Dominating Set for Cardiff - Dijkstra's - Reachability 5km

The nodes chosen to be the dominating nodes for 3km reachability are generally closer together when compared to the nodes for 5km reachability. There is some clustering of the nodes in both Figure 20 and Figure 21, due to the approach of the dominating set algorithm. The clustering in the map of Cardiff seems to be greater. This might be because Cardiff has several subgraphs that are not very connected to the rest of the graph(circled in blue). These subgraphs only have a few nodes that connect them to the rest of the graph.

Approximation Method – Cardiff – 16km^2: The final two figures show the dominating set computed for Cardiff using the approximation method.

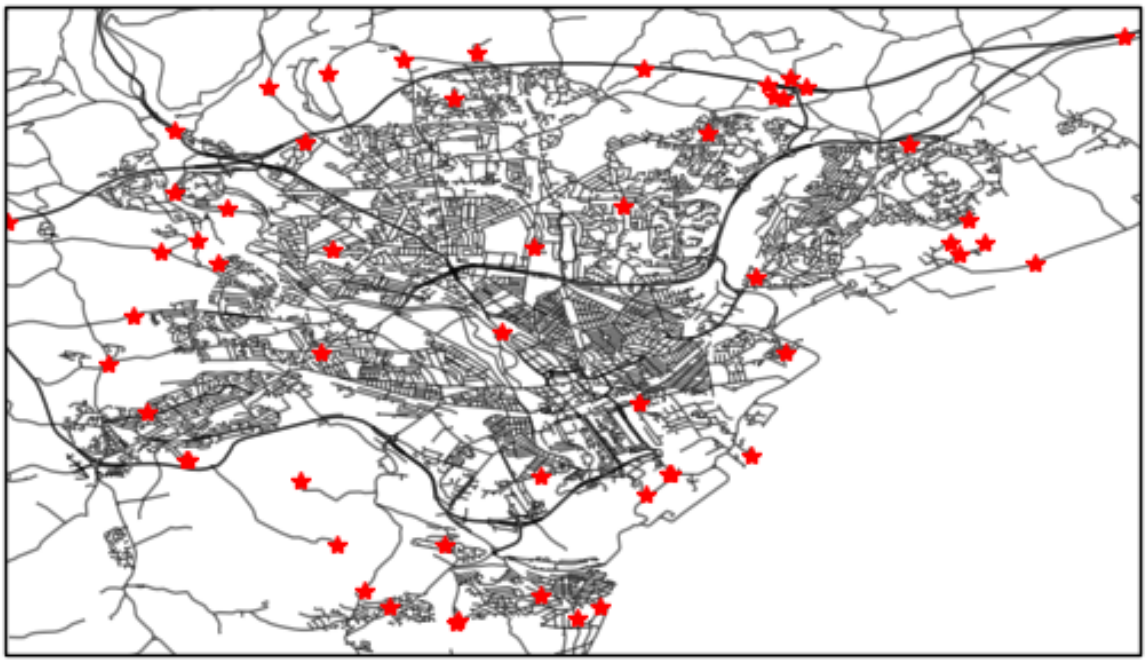

Figure 22 - Dominating Set for Cardiff - Approximation - Reachability 3km

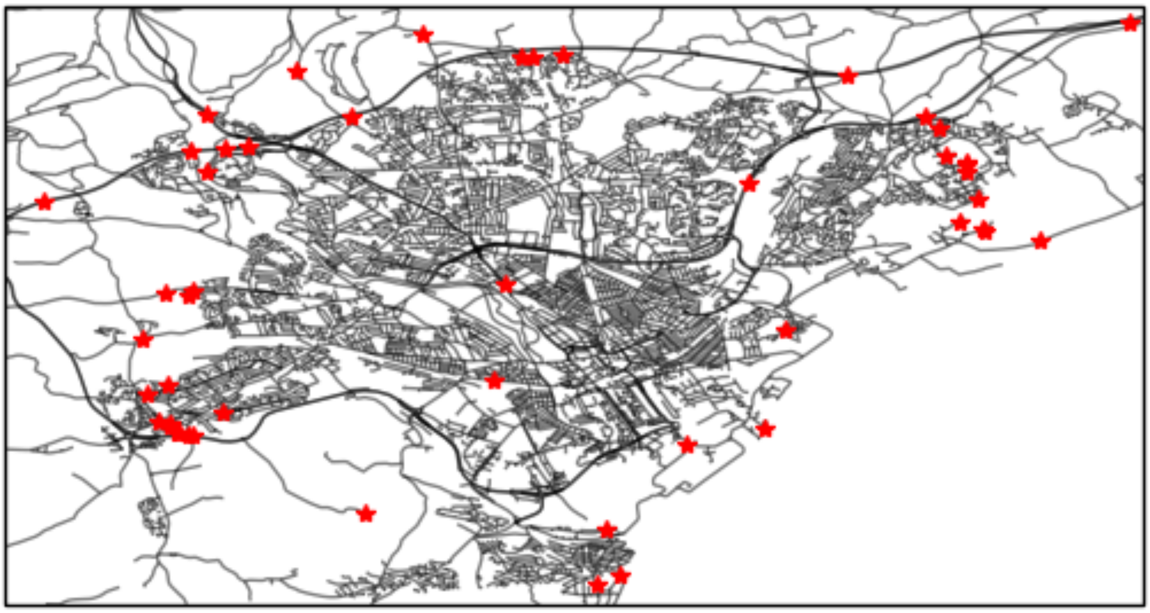

Figure 23 - Dominating Set for Cardiff - Approximation - Reachability 5km

Similar to Figure 19, there is a fair bit of bunching on the outskirts of the map. Clustering is not as prevalent in the smaller reachability distance of 3km. It seems that this clustering is an unfortunate drawback of the approximation method. This clustering may be caused by the approximation algorithm not working correctly at the borders of the map. However, in the center of these two figures show that changing the reachability does change the distance between the dominating nodes.

Reachability Graph Comparison Using a Sample Node: A random sample node has been chosen to illustrate the different reachability graphs produced by both algorithms. The node was taken from the Cardiff street network.

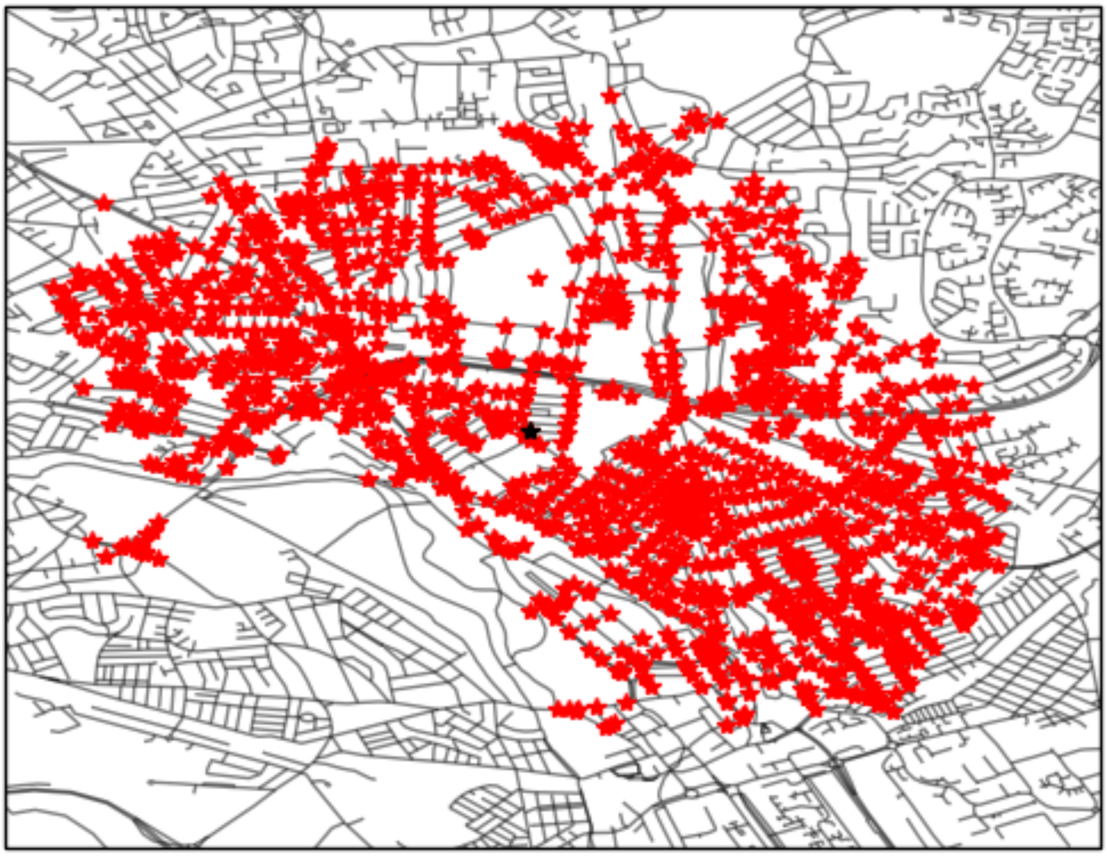

Figure 24 - Reachability Graph For Sample Node - Modified Dijkstra's

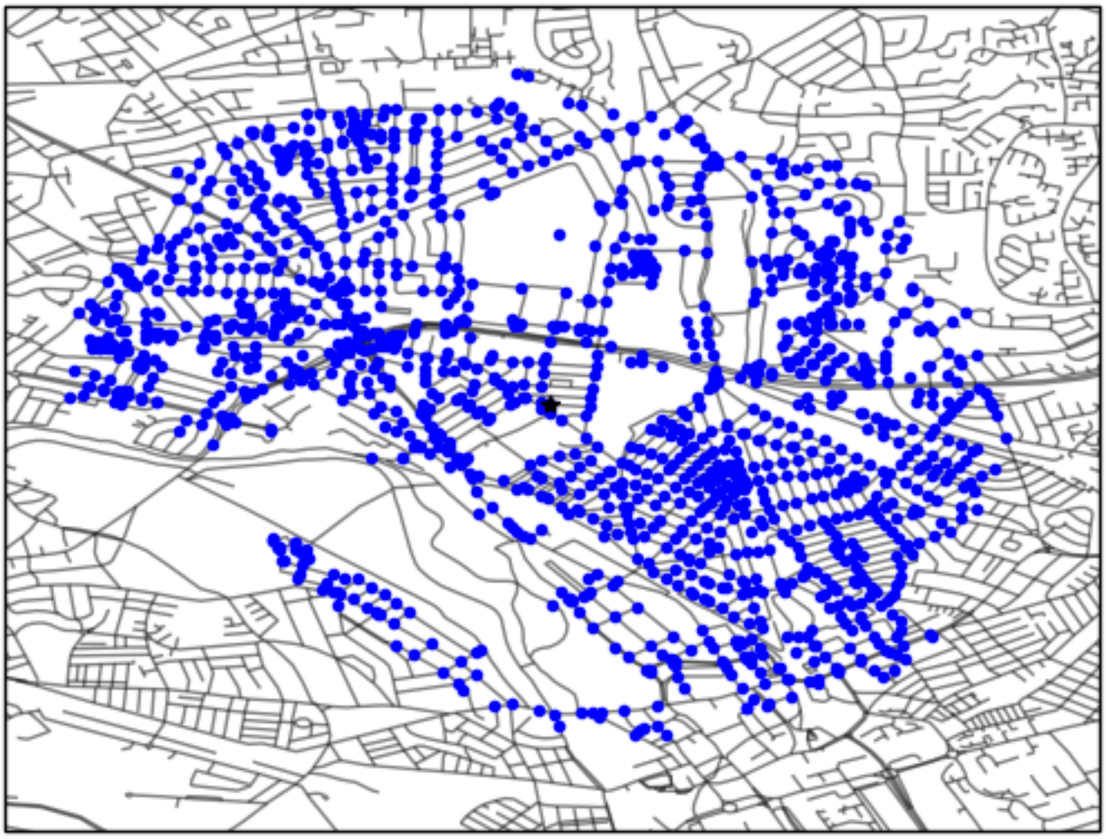

Figure 25 - Reachability Graph For Sample Node – Approximation

Figure 24 shows the nodes that the modified Dijkstra's algorithm marked as reachable as red stars. The black star in the middle is the center node where the algorithm started its search. Figure 25 shows the equivalent nodes approximated to be reachable from the same initial node. These figures show an example of where there is a geographical obstacle blocking the path to the nodes beyond this obstacle, preventing them being reachable.

For this sample node, modified Dijkstra's method created a reachability graph with 1715 nodes. The approximation method created a reachability graph with 1495 nodes. The algorithm has under approximated the actual reachable nodes. The approximation method added 94 nodes that the modified Dijkstra's method did not add. The approximation method did not add 314 nodes that the modified Dijkstra's method added (shown as the red nodes on Figure 26 around the blue nodes). Therefore, for the nodes that were correct, it was only missing 5.48% of the nodes and added an extra 18.31% of nodes.

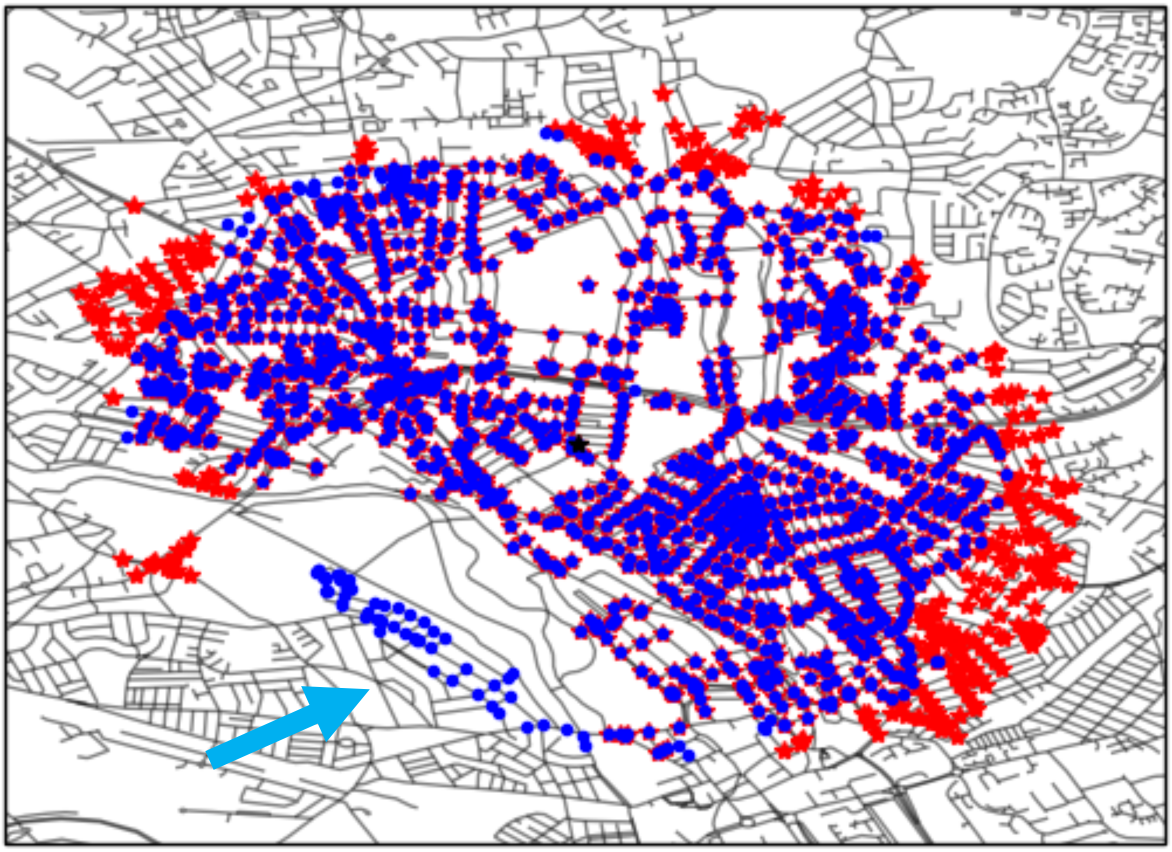

Figure 26 - Reachability Graph For Sample Node - Approximation and Modified Dijkstra's Superimposed

Figure 26 shows the reachability graphs superimposed to see how the approximation method has performed. At first sight it seems that a lot of nodes have been missed. However, the accuracy of the approximation method is acceptable because if most the nodes are correctly marked as reachable, combined with the reachability graphs of the other nodes in the graph, the total reachability graph of all nodes combined (the single highly connected graph) will still produce similar dominating sets. Therefore, the inaccuracy is not as apparent in a large scale.

The blue arrow on Figure 26 points to a cluster of nodes that has been included by the approximation method that are not included by the modified Dijkstra's method. Note that the validation process has been mostly successful in not adding too many nodes that should not have been added, overall the approximation covers the same shape as the modified Dijkstra's method.

The following data are some statistics for the street network of Cardiff on how much the approximation method over and under approximated the reachability graphs. (Note: Over approximated is the measure of extra nodes that aren't in the correct set, under approximated is the measure of number of nodes that have not been added but should have been).

- Average over approximated nodes: 12.6%
- Average under approximated nodes: 27%

These average over and under approximation show that the algorithm is more likely to not mark nodes, making the reachability graph smaller. In the larger scale when considering all the reachability graphs put together, the dominating set computed will have more nodes. Having more dominating nodes means more suggested electric charging stations, which would be preferable over not having enough charging stations. However, 27% means that the algorithm still needs improvements in its approximation.

In summary, the modified Dijkstra's method has been successfully applied to compute reachability graphs. In comparison, the approximation method however has been relatively successful in generating a reachability graph of the same shape as that of the modified Dijkstra's method, including most of the correct nodes. Improvements can be made to the algorithm at a cost in run time, the next section will see how much faster the approximation method performs which will then show if there is much scope to add extra features to improve the approximation.

4.2 – Performance Evaluation

Changing the reachability distance for the two algorithms has shown their expected behaviours on complexity. Changing the street network used has also shown the expected behaviour. Comments will be made on the data produced in the timings and review the predictions made in section 3.7. The data recorded has had its R squared values placed on the graph to see the fit of the values and how closely they compare to the function of complexity through their trend lines.

Changing the Size of the Graph: The results for the modified Dijkstra's algorithm are different between the two different geographic locations.

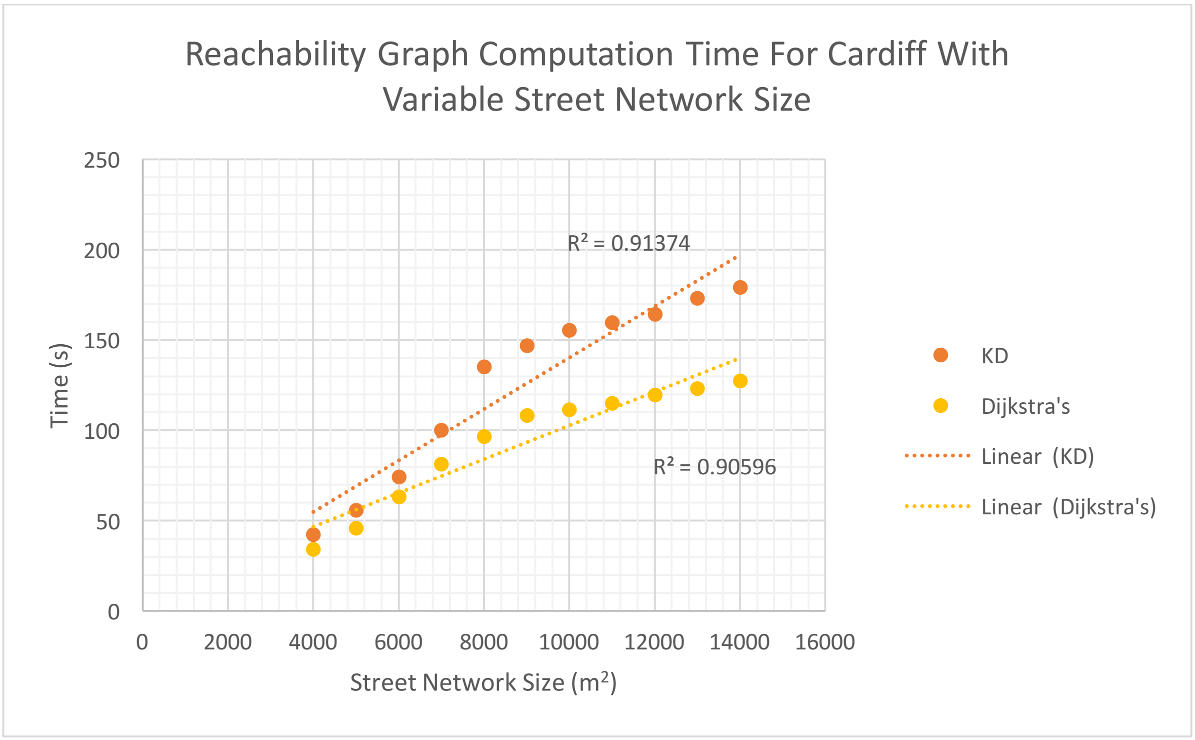

Figure 27 - Reachability Graph Computation Time For Cardiff With Variable Street Network Size

For Cardiff, the computation time did not increase linearly as the size of the graph increased. This is because as the maps became larger, the node count did not increase at the same rate. When the algorithm uses the nodes on the countryside around Cardiff, the tree that it must traverse is much smaller (map of Cardiff illustrated in Figure 20 and Figure 21). Therefore, the runtime for the algorithm decreases as you move away from the centre. This is similarly shown in Figure 27, at street network size 9km the increase in the algorithm runtime slows down with increasing street network size.

An extra note on the street network of Cardiff: since the city is on the coast there will not be any nodes in the body of water. Therefore, this also causes the node count for the larger graphs to not increase proportionately as there is more of the sea included as the graph size increases.

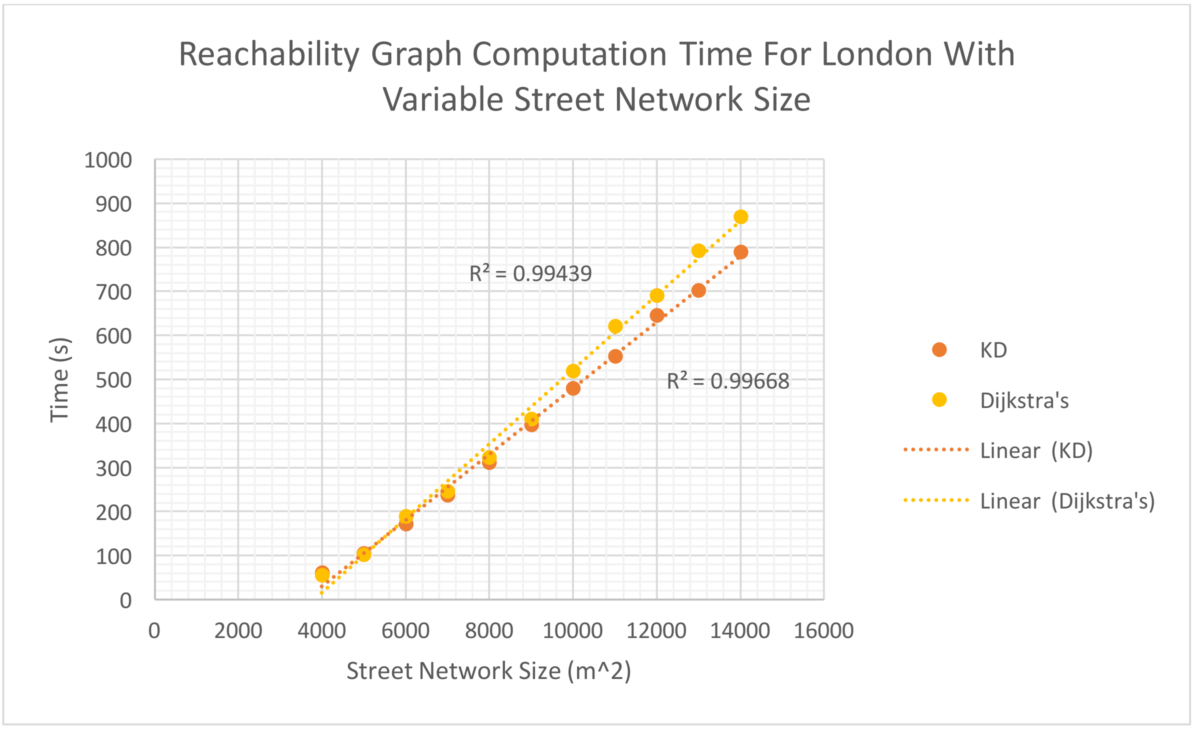

Figure 28 Reachability Graph Computation Time For London With Variable Street Network Size

For London, the computation time increased linearly as expected. The London map has a consistent increase in number of nodes as the map became larger. This shows that the algorithm computation time was essentially the same for each node in the graph.

Figure 30 and Figure 31 below show the linear relationship of changing the size of the graph on the complexity. Comparing the time to compute the reachability between the 16km^2 London network and the 32km^2 for all points shows that by increasing the geographical area by 4 times, the time to compute the reachability also increases 4 times. For example, the modified Dijkstra's method takes around 2000 seconds for the 16km^2 network and 8000 seconds for the 32km^2 network with a reachability distance of 7km.

Changing the Reachability Distance: The data very clearly shows how and when the approximation method takes over in being the more optimal algorithm for computing reachability in Cardiff.

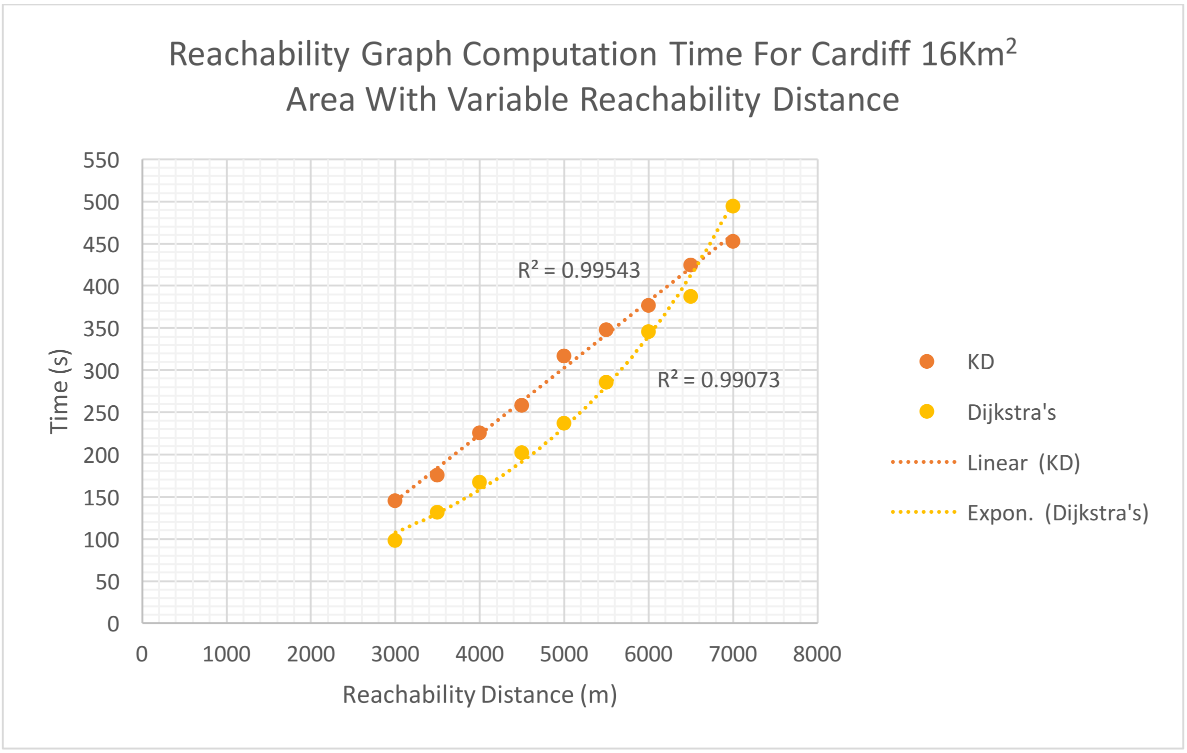

Figure 29 - Reachability Graph Computation Time for Cardiff 16km^2 Area With Variable Reachability Distance

In Figure 29 the reachability distance between 3km and 6km can be more quickly computed using the modified Dijkstra's method. For the approximation method, the higher base time shows that there are not enough nodes in the reachability distance to make using the approximation algorithm worthwhile. At 6.5km however, the time that the modified Dijkstra's method requires to run for this reachability distance starts to show it's $O(N^2)$ complexity. The trend lines cross at this point and now the approximation method becomes the more optimal method in computing the reachability.

Comparing the change in time for the modified Dijkstra's method, the change in time stays consistent. This is because the approximation method is not dependant on how many nodes there are to traverse in a tree like Dijkstra's algorithm. Since it does not perform a breadth first search, the time is used in computing the validation process of the sectors. The validation time is constant since there are a predetermined number of steps that need to be carried out, including the short depth first search by the Greedy Heuristic algorithm. This will cause an insignificant change to the total time even though it's runtime is proportional to the reachability distance.

The following two Figures illustrate the timings for London.

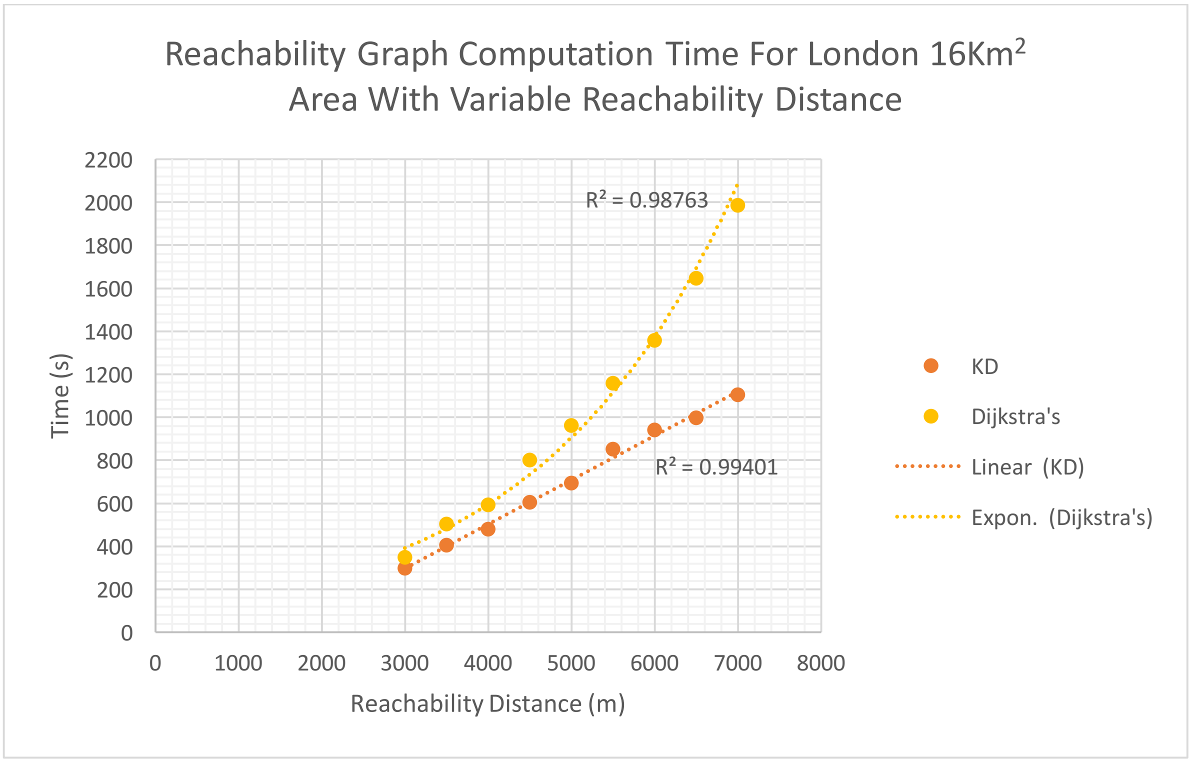

Figure 30 - Reachability Graph Computation Time For London 16km^2 Area With Variable Reachability Distance

Figure 30 shows the reachability times for the same size street network in distance but denser in nodes. The first observation shows that the modified Dijkstra's method from the start has a higher computation time. This shows that the modified Dijkstra's method is traversing more nodes for the same reachability distance when compared to Cardiff's 16km^2 street network.

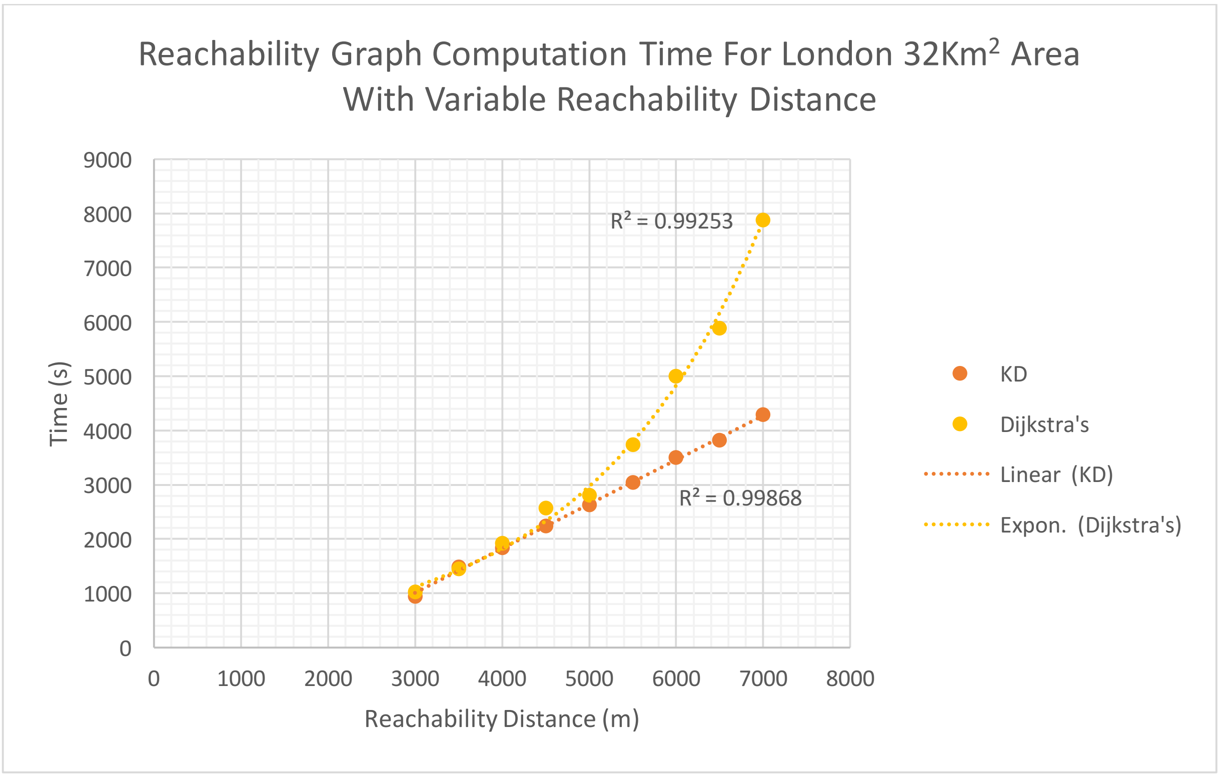

Figure 31 - Reachability Graph Computation Time For London 32km^2 Area With Variable Reachability Distance

Figure 30 and Figure 31 both show the same relationship between the two algorithms on how they scale with reachability. The approximation method is the same speed between reachability distances 3km and 4km but the $O(N^2)$ complexity shows beyond these reachability distances. For reachability 4.5km, it is slightly anomalous but the R squared value is still close to 1 so this deviation is not too great.

Calculating when the Approximation method is faster:
The modified Dijkstra's method will have a cut off where it becomes slower. In Figure 31 the trend line for the modified Dijkstra's method starts to separate from the trend line of the approximation method at around 4.5km reachability distance. The average node count for 4.5km reachability distance was found to be 4206 nodes. Therefore, reachability graphs bigger than this size, the approximation method becomes more efficient.

In summary, it shows that the modified Dijkstra's method starts to slow down and show its $O(N^2)$ with a large enough reachability distance. This comes down to breadth first search not being able to scale well when the number of nodes it needs to traverse becomes too large. Even though the approximation method is faster after a certain reachability distance, it has some accuracy limitations. Making the approximation more accurate would slow down the algorithm by adding extra steps of validation. However, if the reachability distance was required to be much larger than currently being used, then the modified Dijkstra's method would be unfeasible to use due to the size of the list of active nodes in its search.

In Figure 31, if you extrapolate the trend line for the modified Dijkstra's method to a reachability distance of 8km then the time to compute the reachability graph would become excessive. Further to adding a more accurate approximation, if the trade-off in getting a very good approximation equalled to a constant 1000 seconds being added to the time for all reachability distances, the approximation method would still become more efficient at around 5.5km reachability distance.

5 – Future Work

The scope of the project can be increased in many ways, taking into consideration other factors that impact the computation of reachability or further improve the accuracy of reachability. The factors that influence the accuracy of the reachability computed can be static and dynamic data.

To improve the accuracy of the reachability computed, dynamic data could be taken into consideration such as traffic. Traffic is time dependent on street networks, this means that a reachability graph could be calculated for different hours of the day and different days of the week. Considering traffic is already a factor using in efficient route planning [2] for bus routes as the timetable for busses changes throughout the day which is important when users need to plan their routes on bus times. This would change reachability from being a distance to a distance that can be travelled in a set time. Considering traffic is an important factor because electric vehicles will lose charge even when stationary, they will not consume a consistent amount of power when repeatedly starting and stopping in congested traffic. This style of driving is not efficient and would decrease the range a vehicle could drive.

Static data can also be considered, an example of this would be to consider height maps of a street network to determine the slopes of roads. Electric vehicles can optimise their use of battery charge by using their reduced power consumption when driving downhill. There is also the possibility to use the energy gained from acceleration downhill to charge the battery. However, this also works the other way where driving uphill requires more energy. Another example of static data could include no build areas or area that require less charging stations due to factors such as urban areas that do not have space or cannot have stations built, affluent areas that would not want charging stations due to traffic increase or potential negative attention and Greenfield sites that should not be built upon. Further to Greenfield sites, Brownfield sites should be considered more as reusing already built upon land would conserve Greenfield areas and even reduce build costs.

In section 4.1, the graphs illustrating the dominating sets for the approximation method show that the algorithm might not work well at the border of the street network. The algorithm seemed to not have a problem in computing reachability in the middle of the graph. A potential fix for this problem would be to use a larger graph but only compute the reachability for the nodes in the centre, where the Euclidian distance of the node to the border was equal to or more than the reachability distance.

To improve the computation of the reachability graph, there are many ways to approach the speed of computation. The main issue in this project is an inefficient data structure. This problem also suggests that the language Python might not be a suitable language for computing reachability graphs when the scale becomes large. Some of the algorithms implemented by libraries used are coded in C with a Python wrapper which I access. Taking the same approach of implementing the data structure and algorithm used in C, but using Python to handle the simple tasks could greatly improve runtime. An example of where C would provide improvements would be in using binary arrays for all data, which would utilise the smallest amount of memory space.

Another technique that could be used to improve the speed of computing the reachability graphs would be to use a distributed system to parallelise the computation. The computation of the reachability graph for each node is independent of other nodes, so there is no need for complex parallelisation techniques.

6 – Conclusion

In this project, reachability graphs have been computed for two different geographically located street networks using two different algorithms to suggest suitable locations for electric vehicle public charging stations.

A modification to Dijkstra's algorithm was used initially to compute the reachability graphs and it was found to be accurate but did not scale well with network size, resulting in slow computing time. A solution was developed using approximation methods, creating algorithms which assisted the computation of reachability that scale better with network size. The solution involved the KD Tree data structure to quickly find nodes in 2-dimensional space. It also involved a greedy heuristic path finding algorithm to be used with the output of the KD Tree to determine an approximate area for the reachability graph. The greedy algorithm was used to keep the complexity of the problem in linear time to allow the solution to scale better with network size.

This solution was successful for the following situations:

- When the reachability distance covers many nodes
- A highly connected street network is used
- Large graphs (large street network)
- When a lower accuracy of results is acceptable e.g. start of a project which only needs a general reachability and dominating set (feasibility)

However, the solution was less successful in the following situations:

- Small graph (small street network)
- Lots of small subgraphs with low connectivity
- Accuracy of reachability is a necessity

One major issue was the data structure implementation of a library used, which involved a very large and inefficient use of computer memory. It was necessary to test the algorithms without saving the reachability graphs. When the reachability graphs were saved, the algorithms ran considerably slower due to memory inefficiency.

Bibliography

[1] Car Buyer, "Best electric cars," [Online]. Available: http://www.carbuyer.co.uk/reviews/recommended/best-electric-cars.

[2] B. Tesfaye and N. Augsten, "Reachability Queries in Public Transport Networks," [Online]. Available: http://ceur-ws.org/Vol-1594/paper20.pdf. [Accessed 19 4 2017].

[3] University of Maryland, "Graph Definitions," [Online]. Available: https://www.csee.umbc.edu/portal/help/theory/graph_def.shtml.

[4] Grinnel College, "Outline of Class 49: Reachability and Shortest Path Algorithms," [Online]. Available: http://www.math.grin.edu/~rebelsky/Courses/152/98S/Outlines/outline.49.html.

[5] University of Lugano, "Network Algorithms (Spring 2011)," [Online]. Available: http://www.inf.usi.ch/faculty/kuhn/teaching/netalg/lectures/chapter7.pdf. [Accessed 25 01 2017].

[6] Geographic Information Technology Training Alliance, "Dijkstra Algorithm: Short terms and Pseudocode," [Online]. Available: http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html.

[7] A. Patel, "Heuristics," [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[8] Utrecht University, "Geometric Algorithms," [Online]. Available: http://www.cs.uu.nl/docs/vakken/ga/slides5a.pdf.

[9] Clemson University, "Lecture 13: Nearest Neighbour Search," [Online]. Available: http://andrewd.ces.clemson.edu/courses/cpsc805/references/nearest_search.pdf.

[10] V. Lavrenko, "kNN.15 K-d tree algorithm," [Online]. Available: https://www.youtube.com/watch?v=Y4ZgLlDfKDg.

[11] OpenStreetMaps, "Projections/Spatial reference systems," [Online]. Available: http://openstreetmapdata.com/info/projections.

[12] Spatial Reference, "EPSG:27700," [Online]. Available: http://spatialreference.org/ref/epsg/osgb-1936-british-national-grid/.

[13] Geokov, "Map Projections - types and distortion patterns," [Online]. Available: http://geokov.com/education/map-projection.aspx.

[14] University of Wisconsin Madison, "Complexity and Big O Notation," [Online]. Available: http://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html.

[15] University of California Davis, "On-Line Geometric Modeling Notes," [Online]. Available: http://graphics.cs.ucdavis.edu/education/CAGDNotes/Chaikins-Algorithm.pdf.

[16] H. Bast, "Efficient Route Planning," [Online]. Available: http://ad-wiki.informatik.uni-freiburg.de/teaching/EfficientRoutePlanningSS2012. [Accessed 10 2 2017].