Executive Summary

The aim of the project was to research and evaluate a new algorithm for finding the shortest path between a start and a target node. To achieve this, we combined two speed-up techniques; Shortcuts + Arc-Flags (SHARC) and Reach values that exist at present separately. We managed to create an improved algorithm that provides a speed-up of up to 18 over existing implementations. The implementations against which the new speed-up algorithm was tested were Dijkstra with Binary and Fibonacci heap. Moreover, we tested it against the above implementations speeded-up by SHARC and the result had the same excellent speed-up rate of up to 18.

For the evaluation of the algorithm we used real-world data sets. These data sets are road networks of different cities. The column of edge weights was generated by us because the initial files contained only the start and the end node of the edges.

In every scenario, Dijkstra with Fibonacci Heap speeded-up with the new algorithm (SHARC + Reach values) gives us the best results with the smallest pre-processing and query time.

Testing of the new speed-up algorithm gave us very encouraging results for future research. There is more that can be done to improve the speed of the algorithm by approaching the implementation from a different angle. This angle could be the way in which the graph is partitioned. Furthermore, the implementation can be extended from static to time-dependant graphs.

Acknowledgements

I would like to thank Dr. Kirsten Cater for supervising this project and offering her support and knowledge in all the stages whilst allowing me to work on my own way.

Finally, I would like to thank my parents, friends and colleagues for supporting me throughout my studies at the University of Bristol.

Table of Contents

Chapter 1

Introduction

During the last years, there has been a rapid development in journey planning systems. These systems help the user move from one point to another, while keeping track of his preferences and choices. They also provide information about different means of transportation that the user must use in order to reach her destination. Most of the existing journey planners make use of routing and search algorithms in order to provide the optimal path from a source *s* to a target *t*. However, there are still areas for improvement in computing these optimal routes in a transportation network [11,13,14].

In large transportation networks the number of queries, regarding finding the shortest path between two or more points, can become huge. Many problems arise when, in road networks the shortest path has to be calculated in order to get the optimal routing as it happens in drive guiding systems. As more and more parameters change during the journey, such as traffic conditions, a large number of queries have to be answered in order to get a solution, for example, the fastest route. Therefore, the need for designing efficient algorithms is very crucial. For that reason, many different combinations and speed-up techniques for routing algorithms have been developed and are already used in some applications. Most developed techniques require the network to be static or accept only a small number of updates [1]. Nevertheless, in the real world products have to accept dynamic updates because travel time depends on real time data e.g. railway or bus delays. This means that routing models should be based on time-dependent networks. The same happens in road networks where travel times are affected by traffic congestions etc. In addition, in commercial navigation systems the common approach is not to look at less important routes unless the user is close to the source or target [12].

In chapter 2 we state the aim and the objectives of the project. In chapter 3 we focus on the latest products developed in this field, commonly used algorithms that solve search and routing problems and combined speed-up techniques that have been developed in order to make searching and routing in transportation networks faster than just making use of simple routing approaches individually. Finally, we conclude our findings on what has been developed till today and we propose a new approach that handles search and routing queries in an efficient manner. In chapter 4 we explain the design of the three implementations of Dijkstra that we use; simple Dijkstra, Dijkstra with Binary heap, Dijkstra with Fibonacci heap. Moreover, the two speed-ups (SHARC + Reach values) that will be combined in the new approach are analysed. In chapter 5 we present the most important parts of the code from the new algorithmic implementation. Chapter 6 presents detailed analysis of our new approach compared to present algorithms and chapter 7 summarizes our new speed-up technique and discusses methods for future work.

1.1 The Shortest path problem

Every road network can be represented as a graph, that is to say a collection of nodes V and edges E. Each edge connects two nodes and has a weight which is the length of the road for static networks (road networks, Figure 1.1), or an approximation of the time

needed to travel between two nodes for time-dependent networks (bus, rail networks). In graph theory the computation of the shortest path between and a start and a target node is a classical problem. This problem can be separated in [33]:

- *point-to-point:* compute the shortest path length from a source node s ∈ V to a target node t ∈ V
- *single-source:* for a source node s ∈ V, compute the shortest path lengths to all nodes v ∈ V
- *many-to-many:* for given node sets S, T ⊆ V, compute the shortest path length for each node pair (s, t) ∈ S × T
- *all-pairs:* a special case of the many-to-many variant with S := T := V

In this project we focus on the point-to-point variant. The shortest path problem was first solved by Dijkstra in 1959, who gave an algorithm that solves the single-source shortest path problem using O(m+n) priority queue operations for a graph G = (V,E) with n nodes and m edges [33].

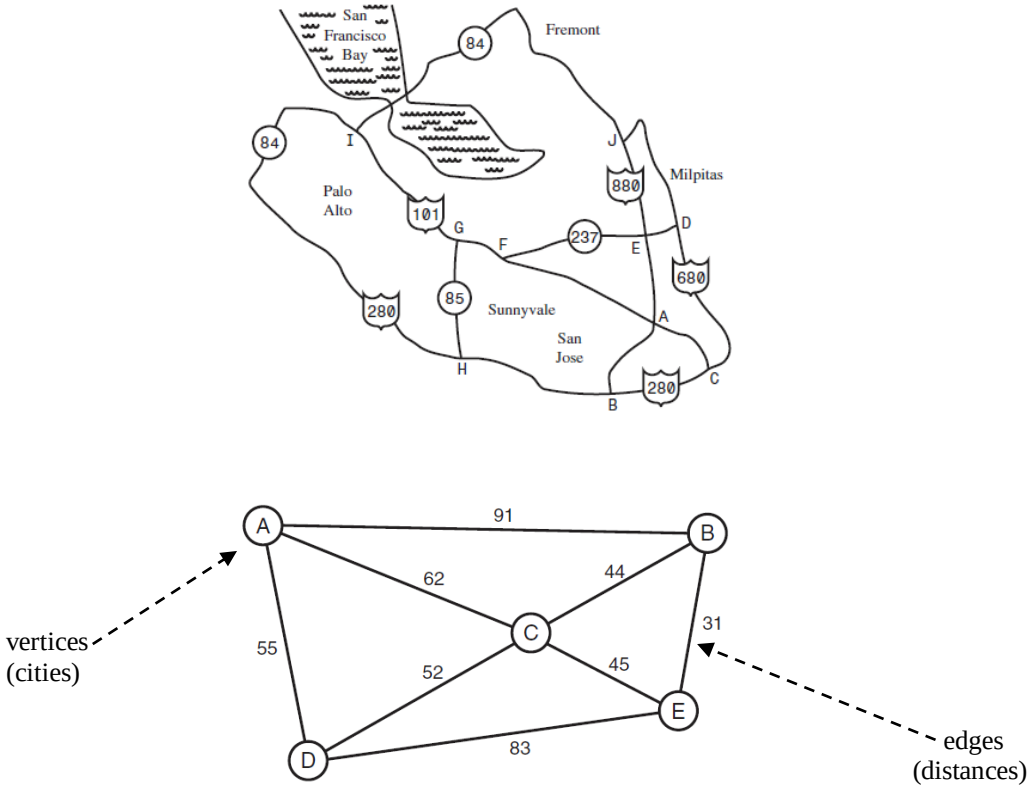


Figure 1.1: A roadmap and a graph, Source: Ref. [42]

1.2 Speed-ups

When we want to compute a shortest path in very large road networks and the point-to-point problem, Dijkstra's algorithm does not give us satisfying running times. There are many approaches that suggest improvements on the timing issue:

A) Dijkstra's algorithm computes the shortest path from a given source node s to all nodes in the graph and not only to the given target node t. For the point-to-point variant this can be altered by stopping the algorithm as soon as target node *t* is found (Figure 1.2) but still the shortest paths from *s* to *v* that are closer to *s* than *t* are determined [33].

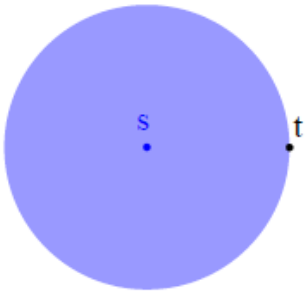


Figure 1.2: Dijkstra's search space, Source: Ref. [33]

B) In many cases we have to compute many point-to-point queries on one road network. Therefore, it is more interesting to pay attention on the pre-processing time that generates "cached" data which can be used later on to improve the running time of the queries [33].

C) Also, we can very easily come to the conclusion that road networks are very sparse and almost planar. Moreover their layout is usually given as each node can be identified by its geographic coordinates. Another attribute of road networks is the hierarchical properties they have such as roads (edges) that are "more important" and others that are "less important" [33].

All the above can lead to the design of speed-up techniques that can perform better query times that those of Dijkstra's algorithm. The requirements that a speed-up technique must fulfil are:

- Query times should be as fast as possible
- The result should be accurate, that is to say the optimal path should be computed.
- The running time of the computation of the shortest path should not differ a lot between a small and a large graph.
- The pre-processing time should be adequately small so that we can apply the speed-up in very large road networks.
- Updating edge weights should be possible.

The challenge however, is to find a good speed-up techniques combination so that the requirements do not conflict with each other.

Chapter 2

Aim & objectives

The *aim* of this project is to research and evaluate a new algorithm for finding the shortest path between a start and a target node.

To do this two speed-up techniques Shortcuts + Arc-Flags (SHARC) and Reach values, algorithms that exist at present separately, were investigated to see whether combining them would create an improved algorithm.

To evaluate whether this new combination is faster or slower than the already developed algorithms, the following were also implemented so that an accurate comparison with our new algorithm could be carried out:

- The Simple Dijkstra
- Dijkstra with Binary Heap / Priority queue
- Dijkstra with Fibonacci Heap (F-Heap)

The *objectives* of this project are:

- Implement a simple Dijkstra algorithm.
- Implement a Dijkstra algorithm using a binary heap.
- Implement a Dijkstra algorithm using a Fibonacci heap.
- Develop the Arc-Flags and the Contraction (shortcuts addition) technique for the above approaches apart from the simple Dijkstra. The combination of Arc-Flags with Contraction will give us the SHARC speed-up technique.
- Develop the new algorithm that takes the Reach values technique and adds it in SHARC.
- Finally, embed the new algorithm in the Dijkstra binary heap and the Dijkstra Fibonacci heap implementation and test the running times.

Chapter 3

Background and Context

In this chapter we will focus on commercial products already in use in the field of routing and search algorithms. Moreover, we will explain in detail the speed-up techniques that have already been developed on some route planning algorithms.

3.1 Products in use

Some well-known commercial products providing navigation and journey planning capabilities along with other functions are presented in this section. Furthermore, there is a reference in functionalities that each product lacks.

Automobile Association in the UK: Has an online route planner for the United Kingdom, Ireland and Europe. Users have the option to choose between roads with or without tolls, congestion charging areas and motorways. It also provides a better responsive context-aware routing system that uses a high sensitivity GPS receiver that can re-route in case of traffic congestion for example. The drawback of the AA system is that it only provides a single-value based routing and this means that context changes that should affect the routing are not considered [12].

ACE-INPUTS: It is a location-based application that retrieves bus routes at a low wireless communication cost, Cannot be used for private driving as it only uses fixed scheduled routes and main roads (less complexity) [15].

ROSE: A mobile application which offers navigation for pedestrians with public transport support. It calculates alternative routes in case of delays in the public transport system. For route planning there are no complex speed-up techniques combined apart from a two step algorithm very similar to A* search algorithm. Moreover it does not solve the time dependent orienteering problem with time windows (TDOPTW) [10].

IPTIS: An integrated public transport information system with journey planning, bulletins and timetables but, it doesn't cover private driving or walking distances [16].

As it can be seen, all of the products use some kind of route planning algorithm. Some make use of extra criteria in order to calculate the optimal path and some other don't. The common thing in all the products is that none of them combines efficiently the proposed speed-up techniques.

In the review section we propose the optimal solution that a journey planning application must have, focusing on the field of route planning and the new algorithm to be used.

3.2 Search & Routing Algorithms

In this section we will start by defining the scheme of all routing algorithms, we then discuss the benefits and drawbacks of each one and present those algorithms that can be combined to produce a better design approach for our final journey planner.

3.2.1 Dijkstra's algorithm

One of the most commonly used algorithms for resolving the shortest path problem in road networks is Dijkstra's shortest path algorithm. The algorithm visits the nodes of the network starting from a given source and finds the shortest path (lowest cost path) between that node and every other node in the graph. In the beginning all nodes are temporarily labelled with the minimum distance. Once a node has been processed by the algorithm it takes a permanent label. Dijkstra's algorithm can be used for finding costs of shortest path from a single node to a single destination node by stopping the algorithm once the shortest path to that node has been found [7,22].

For a graph G(V,E), where V is a set of nodes and E a set of edges, the total time for finding a path between two nodes differs, depending on the data structure (implementation) that is used [1]. The pseudocode for Dijkstra can be seen in Figure 3.1.

```
for each u G:
        d[u] = infinity;
        parent[u] = NIL;
End for
                d[s] = 0; // s is the start point
                H = {s}; // the heap
while NotEmpty(H) and targetNotFound:
        u = Extract_Min(H);
        label u as examined;
        for each v adjacent to u:
                if d[v] > d[u] + w[u, v]:
                        d[v] = d[u] + w[u, v];
                        parent[v] = u;
                        DecreaseKey[v, H];
```

Figure 3.1: Dijkstra Pseudocode [1], Source: Ref. [5]

3.2.1.1 Naive Implementation (Priority Queues/Heaps)

This implementation requires that you keep a list of elements and search among them in order to find the element with the highest priority for each minimum operation. O(1) is the time that this implementation takes in order to insert an element in the list and O(n) priority queue operations [7]. The execution time of Emde Boas implementation [18] is O(n log n). However, in the real world the performance of priority queues for large networks is low due to the fact that access cache faults in the graph create a

bottleneck. Later on, in this paper, speedup techniques will dramatically prove how the queue size can be reduced.

3.2.1.2 Buckets – Basic Implementation, Overflow Bag, Approximate, Double

The problem of the bottleneck in the naive implementation can be disappeared by using a data structure where all labelled nodes can be maintained sorted by distance. This is where the bucket data structure appears. Its bucket in this structure stores temporarily labelled nodes within a range. These nodes look like a double linked list so the operations that can take place are, first check if bucket is empty, second add node to the bucket and third delete a node from the bucket. In Dial's implementation [19] a lot of memory is required so, two implementations arose in order to handle this issue, the overflow bag and the approximate buckets.

The overflow bag implementation (DKM) and the approximate buckets implementation (DKA) resolve the issue of overflow of the basic implementation. DKM stores temporarily labelled nodes whose distance is within a range, in a bucket and all other nodes are kept in a separate set known as the overflow bag. In DKA the nodes are maintained in a FIFO queue. Moreover the values of the distance labels in a bucket are not identical as in DKM but fluctuate within a certain range. The drawback in the DKA case is that space needs may reach high values [20].

The last implementation of Dijkstra's algorithm using buckets is called double as it uses double buckets and combines the two previous scenarios of DKM and DKA. High-level and low-level buckets are maintained and those two levels correspond to different ranges of distance labels [20].

3.2.1.3 Fibonacci Heaps

Fibonacci heaps is another implementation of Dijkstra's algorithm that improves its running time. This heap is a data structure consisting of a forest of trees. Each node in the tree has five pointers for the parent, the left and right sibling, one for one of the children and one for the data. These are used to efficiently access the children of a particular node. In general the implementation is very complex and difficult to implement correctly. However some approaches exist on the implementation of F-heaps [20].

3.2.2 Restricted Search algorithm

Apart from Dijkstra's algorithm and the way it does the search that is to say, starting from a node which is the starting point and moves in a circle until it reaches the destination, there are other approaches such as the restricted search algorithm.

In the restricted search algorithm instead of searching the entire circle (Figure 3.2), it searches in a small area within the part that remains from R2 if cut off by the two bold lines. The R1 has a diagonal straight line named S and Q and R2 is extended from R1 by an offset M2. The straight line SQ parallels the two bold lines with a M1 distance. M1 and M2 are two variables and their values must be decided in such a way to guarantee that the optimal path is inside the restricted area [5]. The pseudocode for the restricted search algorithm can be seen in Figure 3.3.

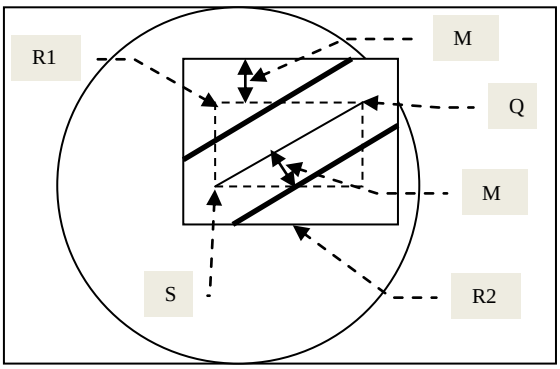


Figure 3.2: Restricted Search algorithm search area, Source: Ref [5]

```
Restricted Search Algorithm:
for each u in G:
        d[u] = infinity;  //node u
        parent[u] = NIL;
End for
        d[s] = 0;        //distance d
        H = {s};
while NotEmpty(H) and targetNotFound:
    u = Extract_Nin(H);
    label u as examined;
    for each v adjacent to u:
        if outOfRange(v), then continue;
            if d[v] > d[u] + w[u, v], then
                d[v] = d[u] + w[u, v];
                parent[v]=u;
                DecreaseKey[v, H];
```

```
Procedure outOfRange(Constraint Area A, Vertex v):
    //A is a polygon given;
    //v is a Vertex being checked;
    Make a straight-line L from v to the right of v;
        Counter = 0;
        For each edge e of A
                if L intersects with e
                        increase Counter by one;
if Counter is even
        return true;
else
        return false;
```

Figure 3.3: Restricted Search Pseudocode, Source: Ref [5]

3.2.3 A* Search, D* Search

A* is a very general graph traversal algorithm widely used and it is an extension of Dijkstra's algorithm. It achieves a good performance as it uses heuristics. What makes A* run more efficiently than other algorithms is that it takes into account the distance already coursed. In other words, it preserves the cost of distance from the start node and not only from the previously visited node. Nodes are stored in priority queues where the smaller f(x) for a node x, the bigger the priority. At every single step of the algorithm the node with the smallest f(x) is removed from the priority queue and its neighbours' values are updated. These neighbours are then added to the queue. The algorithm finishes when a goal node has the lower value f(x) from all the nodes appearing in the queue [5]. The pseudocode for A* search can be seen in Figure 3.4.

```
for each u G:
        d[u] = infinity;
        parent[u] = NIL;
End for
        d[s] = 0;
        f(V) = 0;
        H = {s};
while NotEmpty(H) and targetNotFound:
        u = Extract_Min(H);
        label u as examined;
        for each v adjacent to u:
        if d[v] > d[u] + w[u, v] , then
                d[v] = d[u] + w[u, v];
                        p[v] = u;
                        f(v) = d[v] + h(v, D);
                        DecreaseKey[v,H];
```

Figure 3.4: A* search Pseudocode, Source Ref: [5]

D* search derives from the Dynamic A* which means that the path cost can change while the algorithm is still running. D* search algorithm searches backward starting from the target node. As in A* search, D* algorithm keeps a list of nodes that are to be processed and its one of them can be assigned different labels that show whether the node is in the list or not and whether its cost is higher than the last time it was found in the list [21].

3.2.4 Bellman – Ford

Another algorithm that finds the shortest path in a graph is the Bell-Ford (BF) Algorithm. BF calculates the shortest path to all nodes in a graph from one source. The main idea is that the shortest path to all nodes apart from the source is considered as infinity. Relaxing edges is the next step which means that after checking that the path to a node that the edge is pointing to, is the shortest then we take it. So for its node that there is a path that we can choose to reach it we replace the infinity label with path's weight. This algorithm is applied N-1 times if N is the total number of the nodes in the graph [22,28]. It takes O(VE) time to execute, it is slower than Dijkstra's algorithm but it is more flexible. The pseudocode for Bellman-Ford can be seen in Figure 3.5.

```
for i=1 to |V[G]|-1
        do for each edge (u,v)∈E[G]
                do RELAX(u,v)
for each edge (u,v)∈E[G]
        do if D[v] > D[u] + ω ((u,v))
                then return FALSE
return TRUE;
```

Figure 3.5: BF Pseudocode, Source: Ref [22]

3.2.5 Floyd's algorithm

Next Algorithm for computing the shortest path in a graph is based on a design technique called dynamic programming. This means that easier small problems are first solved before the whole problem. Floyd's algorithm efficiently uses adjacency matrices to solve the problem. It can be easily compared to Dijkstra's algorithm as Dijkstra's calculates the shortest path from one node to all the others whereas Floyd's computes the shortest path from each node in the graph to all other nodes. The main idea for this calculation follows [17].

Given two nodes v,s the shortest path between them is either the edge (v, s) if there is one, or a combination of two paths from v to w and from w to s for a specific node w. In order to calculate the cost, direct edges are used as the lowest cost connection between the nodes, if there is not a direct edge then set the value to infinity or to a large constant. Next step is to select the next sub node k, if k does not exist then it means that all sub nodes have been visited and the calculation stops. Then, select the next starting node m, if there is no path m,k then select the next starting node and continue the same way, if all starting nodes have been visited then select a new sub node (previous step). Finally, check for all target nodes t if the path m,k along with the path k,t is lower in cost than the path m,t, if no path m,k exists the computation stops but if the new path is a lower cost path, save this cost in the position (m,k) in the adjacency matrix [17].

3.2.6 Prim's algorithm & Kruskal's algorithm

Before the explanation of the two algorithms it must be highlighted that they may differ in their methodology but the outcome from both is the Minimum Spanning Tree (MST). Moreover Kruskal's algorithm uses edges while Prim's algorithm uses node connections in mapping out the MST.

Prim's algorithm builds the MST [24] one node at a time. It starts at any node (eg. A) in the graph and finds the lowest cost node (eg. B) that leads to the start node. Next from either A or B it will find the next lowest cost node connection without looping and this continues until all the nodes are connected and an MST is the outcome of the process [23,28]. The pseudocode for Prim's algorithm can be seen in Figure 3.6.

```
procedure Prim(G: weighted connected graph with n vertices)
T := a minimum-weight edge
for i = 1 to n − 2
begin
        e := an edge of minimum weight incident to a vertex in T and not forming a
circuit
                in T if added to T
        T := T with e added
end
return(T)
```

Figure 3.6: Prim Pseudocode, Source: Ref. [23]

Kruskal's algorithm takes a graph with n nodes, keeps adding the lowest cost edge, while avoiding creating cycles, until all edges have been added. In the case where two or more edges have the same cost, the order in which they are chosen does not matter. This means that different minimum spanning trees may result but their total cost will be the same (minimum cost) [28]. The pseudocode for Kruskal's algorithm can be seen in Figure 3.7.

E(1) is the set of the sides of the minimum genetic tree.
E(2) is the set of the remaining sides.

STEPS
E(1)=0,E(2)=E
While E(1) contains less then n-1 sides and E(2)=0 do
From the sides of E(2) choose one with minimum cost-->e(ij)
E(2)=E(2)-{e(ij) }
If V(i),V(j) do not belong in the same tree then
unite the trees of V(i) and V(j) to one tree.
end (If)
end (While)
End Of Algorithm.

(Figure 3.7: Kruskal's Pseudocode, Source: Ref. [25]

3.2.7 Floyd–Warshall algorithm, Johnson's algorithm

The Floyd-Warshall algorithm is used to produce a solution for the all pairs shortest paths problem [9] on a weighted, directed graph. A single execution of the algorithm will find the shortest paths between all pairs of nodes in the graph. Consider a graph with nodes numbered from 1 to N. Moreover, consider a subset of these n nodes {1,2,3...,k}. Now, assume that in order to find the shortest path from node i to node j only nodes within the above set are used. Two situations are examined: first, k is an intermediate node on the shortest path and second k is not an intermediate node on the shortest path.

In the first case, the shortest path can be split into two paths, i to k and k to j. Remember that the intermediate nodes between i,k and j,k are within the set {1,2,3...,k-1}.

In the second case all intermediate nodes are within the set {1,2,3...,k-1}. Then a function F is defined F(i,j,k) equals the shortest path from node i to node j by using the transitional nodes in the set {1,2,3...,k} [9,26]. The pseudocode for Floyd-Warshall can be seen in Figure 3.8.

1: procedure Floyd-Warshall
2: for each in
3: F(i,j,k) = w(i,j), if k=0
4: min(F(i,j,k-1), F(i,k,k-1)+F(k,j,k-1)) if k > 0

Figure 3.8: Floyd–Warshall Pseudocode, Source Ref: [26]

Code line 3 says that if no transitional nodes are allowed then the shortest path between two nodes is the edge's weight (the edge that connects them).

Code line 4 says that transitional nodes exists so the lowest cost path between i and j through nodes {1,2,3...,k} is either the lowest cost path from i to j through nodes {1,2,3...,k-1} or the sum of the lowest cost path from i to k through nodes {1,2,3...,k-1} plus the lowest cost path from k to j through nodes {1,2,3...,k-1}. In this case we calculate both, and choose the smaller one.

Johnson's algorithm is another algorithm that is used in solving the all pairs shortest path problem in a directed, weighted graph. Firstly, it adds a new node with zero-weight edges from that node to all other nodes and runs the Bellman – Ford algorithm in order to check for cycles of negative weight and find f(s) the least weighted edge from the new node to a node s. Secondly, it weights the edges again using the f(s) values of the nodes. Finally, it runs Dijkstra's algorithm for each node and stores the calculated least weight to other nodes that have already been re-weighted using the f values of the nodes, as the final weight [27,28]. The pseudocode for Johnson's algorithm can be seen in Figure 3.9.

Compute G', where V[G'] = V[G] ∪{s}
E[G'] = E[G] ∪{(s, v): v Є V[G] }, and
w(s, v) = 0 for all v Є V[G]
If BELLMAN-FORD(G', w, s) = FALSE
then print "the input graph contains a negative weight cycle"
else for each vertex v ЄV[G']
do set h(v) to the value of δ(s, v)
computed by the Bellman-Ford alg.
for each edge (u, v) Є E[G']
do ŵ(u, v) ← w(u, v) + h(u) – h(v)
for each vertex u Є V[G]
do run DIJKSTRA(G, ŵ, u) to compute δ'(s, v) for all v Є V[G]
for each vertex v Є V[G]
do d(u, v) ← δ'(s, v) + h(v) – h(u)
return D

Figure 3.9: Johnson Pseudocode [28], Source Ref: [29]

3.2.8 Reverse-delete algorithm

The Reverse-Delete algorithm is used to give the minimum spanning tree from a weighted, directed graph. If a graph disconnection exists, this algorithm finds a MST for each disconnected edge of the graph. The group of all these MSTs is called minimum spanning forest and it consists of every node in the graph [24].

This algorithm is the reverse of kruskal's algorithm which starts with an empty graph and adds paths. The Reverse-delete algorithm starts with the initial graph and deletes edges. Its steps are, first, it starts with a graph G consisting of S edges, then goes through S in decreasing order of path cost. Then check, if by current edge's deletion,

further disconnection of the graph will occur. So, if G cannot be further disconnected the edge must be deleted [28].

3.2.9 Comparison Summary

Table 3.10 shows those algorithms discussed previously with their key advantages, disadvantages and time complexity.

Algorithms	**Advantages**	**Disadvantages**	**Time Complexity** (single-source, all-pairs shortest paths)
Dijkstra's	-Has been the initial point for all today's existing speed-up techniques. -There is more space for speed-up techniques	-Far too slow when used on huge databases.	-Emde Boas implementation is O(n log n), where n = number of nodes
Restricted Search	-Restricts the search area of the algorithm and cuts off the part that the shortest path is unlikely to appear.	-The fixed threshold may not get the proper result. -The target may be beyond the restricted area but in the shortest path, as in city traffic lines different categories with different driving speeds exist.	-Faster, as a much smaller search area is used.
A* Search	-Good performance as it uses heuristics. -Preserves the cost of all distances.	-A* solves only one search problem, instead of a series of similar problems.	-Time complexity depends on the heuristic.
D* Search	-Similar to A* but path cost can change while the algorithm is still running.	-D* solves only one search problem, instead of a series of similar problems.	-Slow as during traversing obstacles may occur. Re-planning for overcoming the obstacles is time consuming.
Bellman - Ford	-Optimal for additive edge weights	-Does not hold for bottleneck edge weights (an improved version of BF exists [22,28]) -Does not scale well. -Counts to infinity	-runs in O(nm) time on a graph with n nodes and m edges.
Floyd's	-Easier to code	-Very similar to Dijkstra's algorithm	-$O(n^3)$, n = number of nodes
Prim's & Kruskal's	-Can be fast in a dense graph	-Time consuming phases exist on both algorithms (i.e. sorting, initializations, while loop).	-$O(n^2)$, n = number of nodes. When using the simple implementation (adjacency matrix)

Floyd - Warshall	-Faster as it uses a different formulation for the shortest path problem.	-Can result to negative cycles in a graph.	-$O(n^3)$, n = number of nodes
Johnson's	-Can be faster than Floyd-Warshall in a sparse graph. -It uses Bellman-Ford to remove any negative edge weights.	-Already combines Bellman-Ford and Dijkstra, no space for improvement.	- $O(n^2 \log n + nm)$, n = number of nodes, m = number of edges
Reverse - Delete	-It is the reverse of kruskal's algorithm	-It is a greedy algorithm	- $O(m \log n (\log \log n)^3)$, n = number of nodes, m = number of edges

Table 3.10: Comparing routing & search algorithms [3,6,8]

3.3 Developed Speed-Up Techniques

Many different approaches have been developed in the field of speedup techniques of routing algorithms. In this section we present the techniques that can be a lot faster than Dijkstra's algorithm. A summary of these techniques will be given showing what is next to come in the present report. This means that a new technique will be proposed against existing speed-up techniques, combining extra data, such as weather and carbon emissions, in order to decide which the best route between two points is and which means of transportation should be used.

3.3.1 Static Routing

In static routing the weight of the edges between the nodes in a graph are pre-computed and stored to be used again and again in order to accelerate the queries. The drawback of static routing can be easily understood as, in large road networks it becomes extremely difficult to have shortest paths between all pairs of nodes pre-computed and stored.

3.3.1.1 Related work

Starting with Dijkstra's algorithm, its main drawback is that is very slow on large networks (that contain lots of nodes). Dijkstra's algorithm stops either when all nodes have been visited or when we want to know the distance from a source s to a destination t, it stops as soon as t has been visited. The size if search space of the algorithm, i.e. the number of the visited nodes is O(n) [1,2].

The naive implementation of Dijkstra's algorithm has a running time of $O(n^2)$. The use of *priority queue* operations in the algorithm implementation can lead to O(n log n) execution time. However, the impact of the bottleneck mentioned in 3.1.1 exists.

Bidirectional search runs Dijkstra's algorithm starting from source node to target node and the other way round simultaneously. Once some nodes have been visited from both directions, the shortest path can be determined by the data already collected. This search is used in many speed-up techniques as optional parameter or even mandatory [1].

3.3.1.2 Hierarchical speed-up techniques

Hierarchical approaches try to achieve the hierarchical structure of a given network [2]. In pre-processing step a hierarchy is extracted and can be used to speed-up all sub-queries.

Reach-Based routing: Let $R(w) := \max R_{st}(w)$ be the reach of node w, where $R_{st}(w) := \min(d(s, w), d(w, t))$ for all s-t shortest paths including w. Gutman [2] observed that a shortest-path search can be cut off at nodes with a reach too small to get to either source node or destination node from there. The basic approach was considerably strengthened by Goldberg et al. [2] by a clever integration of shortcuts (single edges that represent whole paths in the original graph) [Source: Ref [2]].

Highway Hierarchies(HH): group vertices and edges in a level hierarchy by alternatively using two procedures. First is contraction (node reduction) that removes low degree vertices by bypassing them with shortcut edges. This means that nodes of degree one and two are removed during this process. Second, edge reduction removes edges that only appear on shortest paths close to source or destination. This means that every node w has a neighbourhood radius r(w) that we are free to choose. HHs were the first speed-up technique that could be used on large road networks giving query times in some milliseconds [1]. The main reason for this positive outcome was that due to the contraction procedures the road network remains small and two-dimensional. Moreover, pre-processing can be done by using a small number of local searches starting from each node which resulted in the fastest pre-processing at that time. It must be mentioned that HHs are faster than route-based routing with respect to both query time and pre-processing time.

Advanced Reach-Based Routing: It seems that the pre-processing techniques mentioned in HHs can be adapted to pre-processing reach information. This makes reach calculation faster and more precise. More significantly, shortcuts make queries more achievable by reducing visited nodes and by reducing the reach-values of the nodes bypassed by shortcuts[1].

Highway-Node Routing(HNR): In [1,2] the multi-level running scheme with overlay graphs is generalized so that it works with arbitrary sets of nodes rather than only with separators. This is achieved by using a new query algorithm that stops suboptimal branches of search in the graph on lower levels in the hierarchy. So, by using important nodes for higher levels, query performance can be compared to the performance of HHs. In HNR the node classification is given by a pre-calculated HH. The advantage of HNR over HH is its easier search algorithm and a simple way to update the pre-processed data in case edge costs change due to traffic jams or delays. Moreover, the search space size using the HNR technique is greatly reduced.

Contraction Hierarchies(CHs): This is a special case of HNR where n levels appear, one for each node. Such an hierarchy can improve query performance. Furthermore, queries are more simplified as it is only needed to search upward in a search algorithm which means that considerable space is saved since each vertex is only stored at its lowest endpoint. CH pre-processing is divided in two phases. In the first phase nodes are ordered depending on the importance and in the second phase nodes are removed in that order. A node w is removed from the network in such a way that shortest paths between remaining nodes are not lost. The whole implementation may look similar

to HHs but it uses only node contraction. This technique handles removed nodes in a very careful way using priority queues where nodes are stored with priorities depending on how preferable a node is for removal. In addition to the matter of priority the CHs check the number of edges emerging in a graph if a node is removed (negative value may appear). Due to its efficiency CHs is used in almost all the advanced routing techniques [1].

Distance Tables: After the size reduction of a network G using the above techniques a new network G' is generated. In G' one can afford to pre-compute and store a complete table of distances for the remaining nodes. The use of such table can help one stop a query when G' is reached. In order to calculate the shortest path distance, it is sufficient to search al shortest path distances between nodes entering the new G'' by forward and backward search. Since the number of source nodes is small, one can achieve a speed-up close to two compared to the underlying hierarchical technique [1].

Transit-Node Routing(TNR): This technique pre-computes, apart from a distance table for transit nodes, all relevant connections between transit nodes and the remaining nodes. Three different approaches have been proved successful for selecting transit nodes: separators, border nodes of a partition, and nodes categorized as important from other speed-up techniques. It looks like the latter one is the most promising approach for route planning in road networks. Since only about seven to ten such access connections are needed per node one can almost reduce routing in large road networks to about ten table lookups. In addition several layers of transit nodes are introduced to solve the problem of local queries now that the shortest path does not touch any transit node. Among lower layer nodes, only the routes that do not touch the higher layers need to be stored. TNR reduces routing time to a few microseconds while it takes longer to pre-process and consumes additional space[1,2].

3.3.1.3 Goal-Directed speed-up techniques

The main idea of goal directed speed-up techniques is to direct the search to target t by choosing edges that limit the distance to t and by rejecting edges that in unlikely to belong to a shortest path to t. It has to be mentioned that such decisions are usually made by using pre-processed data.

Edge Labels: The main idea here is, to pre-compute the information for an edge p that specifies a set of nodes F(p) with the attribute that F(p) is a superset of all nodes that relax on a shortest path starting from p. Faster pre-computation is possible by separating the graph into o regions that almost have the same size and only a small number of margin nodes. Now F(p) is represented as an o-vector of edge flags [1] where flag j shows whether there is shortest path that contains edge p and leads to a node in region j. Edge flags can be calculated using a single source shortest path computation from all boundary nodes of the regions. A better improvement can be found (one search for each region) here [1,4].

Landmark A (ALT):* The ALT algorithm is based on A* search, Landmarks and the Triangle inequality. After landmarks (a small number of nodes) have been selected the distances d(v,k), d(k,v) to and from each landmark k are pre-computed for all nodes v. For nodes a and b, the triangle inequality gives for each landmark k two lower bounds

d(k,b)-d(k,a) <= d(a,b) and d(a,k)-d(b,k) <= d(a,b). The maximum of these lower bounds is used in an A* search. The original ALT, as described in [1,2], has fast pre-processing times and provides reasonable speed-ups. However it consumes a lot of space for large networks. In the section of "previous combinations" we will describe ways to reduce memory consumption. Finally, we must point out that the success of ALT depends a lot on the choice of landmarks as described in [Goldberg].

Precomputed Cluster Distances(PCD): The network is divided into clusters and then a shortest connection between any pair of clusters S and T is pre-computed. PCDs and A* search cannot be used together since already reduced edge weight may take negative values. However, PCDs offer lower and upper bounds for distances that can be used to limit searching. This approach gives a speed-up that can be compared to ALT as it uses less space. By using many-to-many routing techniques cluster distances can be efficiently computed [1].

Arc-Flags(AF): It computes a partition P of the graph. An element of a partition is called cell and a label is attached to each edge w. A label contains a flag for each cell which is true if a shortest path to a node starts with w. Then a modified Dijkstra version uses only those edges for which the flag of the target node's cell has the value "true". This big advantage is that this approach has a very fast and easy query algorithm. However, pre-processing time and memory consumption are the biggest drawbacks of this method as described in [2].

3.3.1.4 Implemented Combinations

Speed-up techniques can be combined. As mentioned in [2], a combination of geometric containers (an old approach of AFs), a multilevel method (early hierarchical approach) and A* search produces a speed-up of 62 for a railway transportation problem. In Holzer et al. [2004], different combinations are tested and it concludes that depending on graph type, different combinations turn out to be the best [2].

REAL: Goldberg et al. [2006, 2007] have combined an advanced version of Reach and the ALT algorithm to produce the REAL algorithm. In the most recent version Goldberg et al. [2007], a variant is introduced where landmark distances are stored only with nodes with high reach values. This means that memory consumption can be significantly reduced [2].

HH:* Delling et al. [2009b], combines HHs with ALT algorithm. Because the landmarks are not chosen from the original graph, but for some level h of the HH, this approach reduces pre-processing time and memory consumption. As a result, the query runs in two phases. In the first phase, a highway query runs until the search reaches level h, all nodes of level h reached during this phase are called entrance points to level h. For the remaining search, landmark distances are available so that a combined (goal-directed) algorithm can be used [Source Ref: 2].

SHARC: As described in Bauer and Delling [2008, 2009], SHARC combines data from HHs with the Arc-flags approach. The outcome is a unidirectional query algorithm, which has advantages in scenarios where bidirectional search cannot be used [2,4]. In road networks for example congestions can be predicted during rush hours. This situation can be dealt by using time-dependent networks which means that travel duration depends

on the departure time and this can be modelled by assigning travel time functions to the edges of the graph. In Delling [2008, 2009], SHARC has adapted this scenario [2,4].

3.3.2 Time-Dependent Routing

As we have seen, so far, most of the developed techniques indicate that the network is static or only let a small number of updates. However, in the real world travel duration relies on the departure time. It seems that most of the models for routing in many transportation systems are based on time-dependent networks. Focusing on a time-dependent scenario is much more challenging than the expected. The input data size increases as there are many changes during the day, in motorways, regarding travel time. From a technical point of view static techniques rely on a second search that starts from the target to the source (bidirectional search). This concept is rejected on time-dependent scenarios. Moreover, finding the shortest path statements become much more complex than previously in static routing. A major difference between time- dependent and static routing is the usage of functions for specifying edge weights, instead of constant numbers. Concluding, none of the above techniques have ever been used or adapted to this realistic approach [1].

Analyzing the speed-up techniques in general, we can see that the most major ingredients are Dijkstra's searches and contraction. Routing in time-dependent networks require the rise of both ingredients. Till today, three speed-up techniques have been adapted to the time-dependent approach. They either perform a backward search or use unidirectional search [1].

A with landmarks:* ALT performs correct queries as long as possible potentials exist. This means that by using the lower bound of each edge during pre-processing, ALT queries remain correct. A high speed-up can be achieved by a bidirectional approach. In this case, a time-independent backward search bounds the node set that has to be examined later on by the forward search. The disadvantage here is its small speed-up [1].

SHARC: SHARC is a unidirectional speed-up technique being as fast as bidirectional approaches. It is based on two bedrocks, contraction and edge flag computation. An edge flag can be set as soon as it is important for at least one departure time. This results in, setting arc-flags of quickest paths for all departure times between two points, true. By setting suboptimal arc-flags, pre-processing times can be reduced for the price of query performance. Depending on the level of derangement, time-dependent SHARC is up to 214 times faster than Dijkstra [1,4].

Contraction Hierarchies: In their simple implementation, node ordering (first phase) can be done on a static graph. But, the second phase needs to replace Dijkstra searches by profile searches. The query algorithm is less obvious and forward-backward Dijkstra searches on the static version are replaced by four phases. First, a time-dependent forward-upward Dijkstra search. Second, backward exploration of the downward edges leading to the destination (target). Third, suboptimal paths pruning and fourth, a forward-downward Dijkstra search starting, from all the promising places where forward search and backward exploration met that is confined to the edges explored in phase two, simultaneously. Many different forms of upper and lower bounds on travel time can be used to accelerate pre-processing, improve pruning and save space [1].

3.3.3 Comparison Summary

Table 3.11 demonstrates the performance of different speed-up techniques on two different data sets.

	Europe				USA			
	PREPROCESSING		QUERY		PREPROCESSING		QUERY	
method	time [min]	overhead [B/node]	#settled nodes	time [ms]	time [min]	overhead [B/node]	#settled nodes	time [ms]
Dijkstra	0	0	9,114,385	5,591.6	0	0	11,847,523	6,780.7
Bidirectional D.	0	0	4,764,110	2,713.2	0	0	7,345,846	3,751.4
Reach	**45**	**17**	**4,371**	**3.06**	**28**	**20**	**2,405**	**1.77**
HH	13	48	709	0.61	15	34	925	0.67
HNR	15	24	981	0.85	16	1.6	784	0.45
CH	25	-2.7	355	0.18	27	-2.3	278	0.13
TNR	164	251	N/A	0.0056	205	244	N/A	0.0049
ALT	13	70	82,348	120.1	19	89	187,968	295.4
AF	2,156	25	1,593	1.1	1,419	21	5,522	3.3
REAL	103	36	610	0.91	114	45	538	0.86
HH	14	72	511	0.49	18	56	627	0.55
SHARC	**81**	**14.5**	**654**	**0.29**	**58**	**18.1**	**865**	**0.38**
CALT	11	15.4	1,394	1.34	13	16.1	2,697	3.01
pCH-0.5%	19	-2.8	97,913	4.28	21	−2.3	121,636	50.57
pCH-5.0%	15	-2.9	965,018	53.63	15	−2.3	1,209,290	667.80
eco CHASE	32	0.0	111	0.044	36	−0.8	127	0.049
gen CHASE	99	12	45	0.017	228	11	49	0.019
pCHASE-0.5%	21	-1.6	2,544	0.83	25	−1.5	4,693	1.40
pCHASE-5.0%	31	3.6	12,782	4.14	96	4.3	22,436	7.21
ReachFlags	348	37	1,101	0.84	671	35	693	0.50
pReachFlags	229	30	1,168	0.76	318	25	1,636	1.02
TNR+AF	229	321	N/A	0.0019	157	263	N/A	0.0017

Table 3.11 [Source: Ref. 2]: All the speed-up techniques previously discussed, along with measurements. The cells of the table between the two bold lines come from new combinations of speed-up techniques introduced and described in the February 2010 published paper [2]. Four categories can be distinguished in the table: Hierarchical Methods, Goal-Directed Methods, Developed combinations and newest combinations.

3.4 Summary

To sum up, there is still space for improvement in the field of applications regarding journey planning. Journey planning systems need to combine routing algorithms and speed-up approaches in order to provide the best solution for the shortest path problem.

In paragraph 3.1 none of the products mentioned, combines all these features along with a fast and less space consuming routing algorithm. In paragraph 3.2 we presented existing searching and routing algorithms focusing on their pseudo codes and the advantages and disadvantages of each. Based on the pseudo codes and Table 3.10 we conclude that Dijkstra's algorithm is the most appropriate to use in our new implementation as Dijkstra can be optimised using new combinations of speed-up techniques. Moreover, after comparing the different speed-up techniques in Table 3.11, it seems that there is room for SHARC-routing improvement and also a new computation of reach values with SHARC. Another interesting approach not yet analysed, would be to adapt the contraction routine from Geisberger et al. [2008] to SHARC. These approaches hold on to the fact that SHARC first increases the hierarchy when the query begins and then runs a goal-directed query starting from the highest level and then moves lower to the hierarchy as the search is moving closer to the target cell. Despite the fact that this seems as fast as if we were using a bidirectional approach, SHARC can be used in schemes where other approaches fail due to their bidirectional mechanism. So it can be easily understood that SHARC is the best speed-up technique for route planning that is time-dependent.

The description of the problems along with the solutions proposed above lead to a new algorithmic approach that will try to cover most of the needs of a traveller who plans a journey, in an efficient optimal manner.

Chapter 4

Algorithms Design

In this section we will focus on the design principles followed for the three different implementations of Dijkstra (Simple, Binary Heap, Fibonacci Heap) and the speed-ups of SHARC and Reach values.

4.1 The simple Dijkstra

In this implementation of Dijkstra there is a class named Graph that consists of vertices and edges, an Edge class that has a source and a destination and a class Vertex holding the id and name of each node. Moreover, this implementation is a very simple one as it does not use any performance optimization.

This simple version of Dijkstra partitions all nodes in two sets. Settled and unsettled. When the algorithm starts all nodes are in the unsettled sets which means that their evaluation is still pending. After a shortest path from the source node to a node has been found this node is moved to the settled set. The distance of each node when the algorithm starts is set to infinity (a very high value). Furthermore, initially only the source node is in the unsettled set. The algorithm runs till the unsettled set is empty. In each loop a node, out of the unsettled ones, with the lowest distance from the source is chosen. All outgoing edges from the source are read. Then, the algorithm evaluates for each unsettled destination node of these outgoing edges if the known distance between the source and that node can de decreased in case we use the selected edge. If this can happen then the distance is updated and the current node is added to the nodes that need to be evaluated. It is also important to mention that Dijkstra also allocates the pre-successor of each node as it is heading to the source [30].

4.2 Heaps

A heap is an abstract data structure that consists of a set of items (Figure 4.1). Each of the items has a valued key subject to the following operations [34]:

- make heap: returns a new, empty heap
- insert(z, h): insert item z with predefined key into the heap h
- find min(h): return an item of minimum key in heap h
- delete min(h): delete an item of minimum key from the heap h and return it

Additionally, there are some useful operations on heaps:

- link(h1 ,h2): merge the two disjoint heaps h1, h2 and return the final heap
- decrease key(d, j, h): decrease the key of item j in heap h by subtracting the non-negative real number d
- delete(j, h): delete an arbitrary item j from the heap h

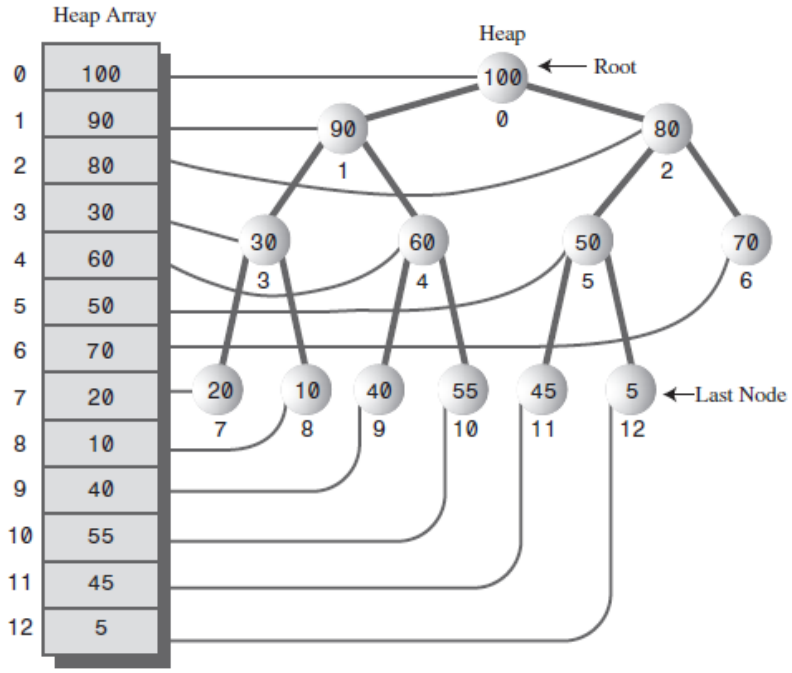


Figure 4.1: A heap and its underlying array, Source Ref: [42]

4.2.1 Dijkstra with Binary Heap

A binary heap is a heap data structure which is created using a binary tree. Moreover, a binary heap is simpler and uses space more efficiently than a binary tree. In the current implementation we make use of the priority queue because it enqueues elements and can process the element with the highest priority first. We develop the data type of priority queue by keeping an associative array in which each priority is mapped to a list of elements having that priority. Adding or removing an element from the queue takes O(log n) time, whereas peeking an element with the highest priority without the need of removing it takes constant time. This happens because a search of all keys must be made in order to find the largest one. In our Java code the binary heap is constructed from an array [31] as Figure 4.2 shows.

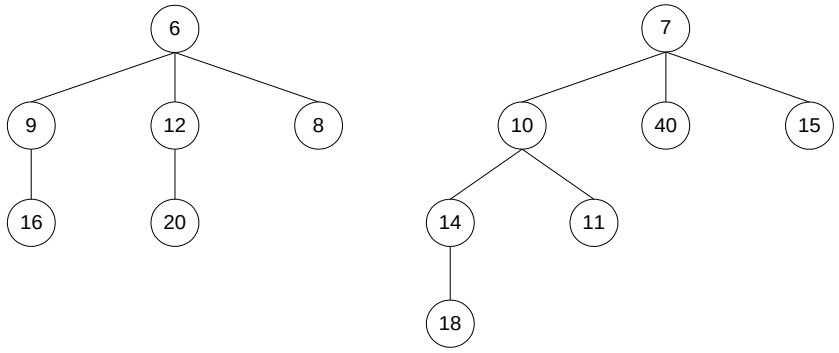
```
public BinaryHeap( Comparable [ ] items ) {
    currentSize = items.length;
    array = new Comparable[ items.length + 1 ];

    for( int i = 0; i < items.length; i++ )
        array[ i + 1 ] = items[ i ];
    buildHeap( );
  }
```

Figure 4.2: Source Ref: [31]

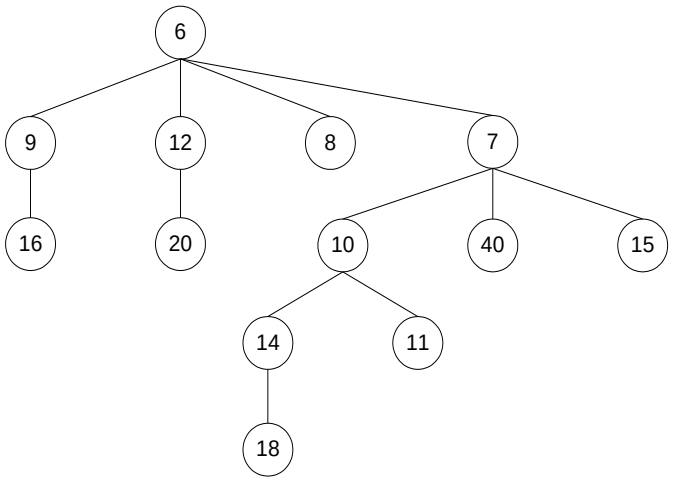
A node in the binary heap has four pointers, one for its parent, two for its children and one for the data.

4.2.2 Dijkstra with Fibonacci Heap

In order to implement Dijkstra with Fibonacci heap we make use of heap-ordered trees. A heap-ordered tree is a rooted tree that contains a set of items, one item in each node. These items are arranged in heap order. This means that if *v* is a node with a parent *p(v)* then the key of the item in v is no less than the key of the item in its parent. Hence, the tree root contains an item with the minimum key. One of the most important operations on heap-ordered trees is the merging (linking), which combines two disjoint trees into one. This means that if we have two trees with roots *i* and *j*, these trees can be merged by comparing the keys of the items in *i* and *j*. If the key of the item in i is smaller, we make j a child of i, otherwise, we make i a child of j (Figure 4.3).



(a) Two trees



(b) After linking

Figure 4.3: Linking two heap-ordered trees, Source Ref: [34]

A Fibonacci heap is a collection of item-disjoint heap-ordered trees. The number of children a node has is called *rank*. Each node has a pointer to its parent and a pointer to one of its children. The children of each node are doubly linked in a circular list. The roots of all trees in the heap are also doubly linked in a circular list. In order to access the heap we use a pointer to a root that contains an item of minimum key. This root is called

the minimum node of the heap. If this minimum node is null it is denoted that the heap is empty (Figure 4.4). The above description requires space in each node for: one item, five pointers, an integer and a bit (shows if the node has been processed or not). Due to the double linking of the lists of roots and children a deletion could take O(1) time. The circular linking makes concatenation of lists in O(1) time [34].

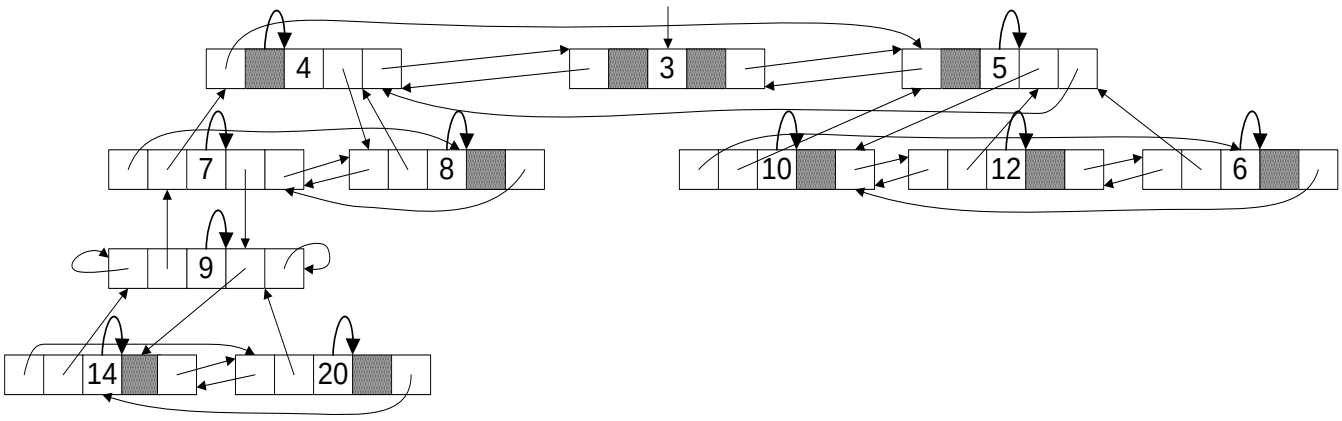


Figure 4.4: F-Heap pointers, 5 pointers in each node: left sibling, the parent, some child, the right sibling and the data, the middle field represents the key of the node, Source Ref: [34]

Continuing with the heap operations we have the *make heap* which in order to perform it we return a pointer to null. For *find min(h),* we return the item of the minimum node in the heap h. To perform the *insert(z, h),* we create a new heap consisting of a node containing z and replace the heap h by the linking of h and the new heap. For the linking *link(h1 ,h2)* to be performed, we combine the root lists of *h1*, *h2* into a single list and return as the minimum node of the new heap either the minimum node of *h1* or the minimum node of *h2*, whichever contains the item of the smaller key. All these operations take O(1) time [34].

The *delete min(h)* operation is the most time consuming. Deletion begins by removing the minimum node, assume *v* from *h*. Then we concatenate the list of children of i with the list of roots of h other than *v* and the linking step until it no longer applies. Linking step: We find two trees having roots with the same rank and link them. Once there are no two trees with roots having the same rank, we create a list of the remaining roots. The running time of the delete min operation is O(log n), where *n* is the number of items in the heap. When deleting the minimum node, the number of trees increases by at most log n and each linking step decreases the number of trees by one.

In order to perform the *decrease key(d, j, h), d* is subtracted from the key of *j*, then the algorithm finds the node v containing j and cuts the edge that joins v to its parent *p(v)*. To do this, v is removed from the list of children of *p(v)* and the parent pointer of *v* is set to null. The previous cut puts the sub-tree rooted at v into a new tree of h and also decreases the rank of *p(v)* and adds *v* to the list of roots of *h*. If the new key of *j* is smaller than the key of the minimum node, we define *v* to be the minimum node again. This operation works as *d* is non-negative, so decreasing the key of *j* leaves the heap order

unchanged within the sub-tree rooted at *v*, whereas it may violate the heap order between *v* and its parent *p(v)* (Figure 4.5). The decrease key operation takes O(1) time [7,34].

The next operation is similar to *decrease key* and is called *delete*. To perform the *delete(j, h),* the algorithm finds a node *v* containing *j*, cuts the edge that connects *v* with its parent, forms a new list of roots by concatenating the list of children of *v* with the original list of roots and deletes node v (Figure 4.5). A delete operation takes O(1) time, unless the deleted node is the minimum one [34].

Discussing further the Dijkstra implementation using the F-heap let G be a directed graph having a node know as the source s, and each of the edges in the graph has a non-negative length. The length of a path in G is the sum of its edge lengths. The distance of a node v from a node u is the minimum length of a path from u to v.

The required heap node operations for the Dijkstra F-heap algorithm are [32]:

- The AddChild which adds a child to the current node by inserting it into the children list and linking it. The operation takes O(1) time.
- The AddSibling which adds a node in the children list where the current node belongs to. The operation takes O(1) time.
- The remove operation which removes the node from the sibling list and updates any pointers affected by the removal. The operation takes O(1) time.

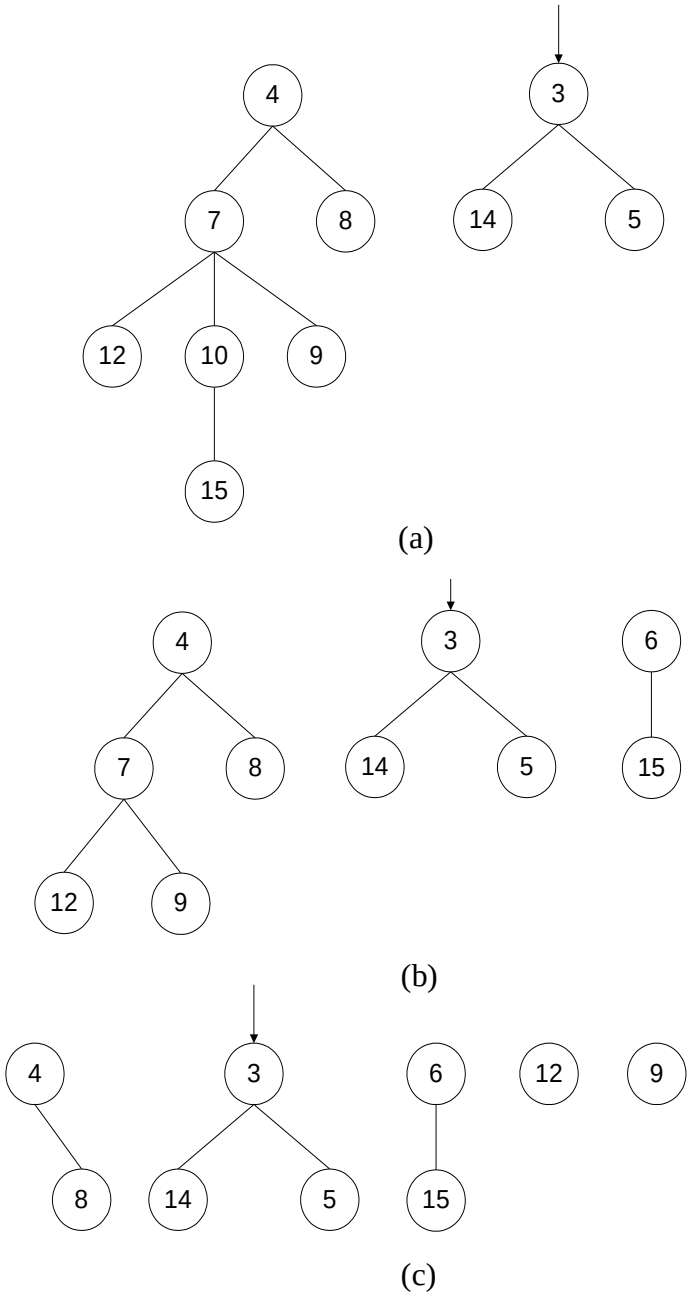


Figure 4.5: decrease key & delete operations, (a) the original heap, (b) after decreasing key 10 to 6, (c) after deleting key 7, Source Ref: [34]

The Fibonacci heap implementation was found in [32] written in C++. The code was translated to Java and altered for the needs of our data sets.

4.3 Speedups on Dijkstra

The two speed-up techniques which will be analysed are the SHARC and the Reach values.

4.3.1 SHARC

SHARC routing for static scenarios is implemented for both Dijkstra binary heap and Dijkstra Fibonacci heap. The simple Dijkstra cannot be speeded-up as it is a very basic code without making use of any optimization performance such as heaps. Heaps are needed in order to apply the SHARC routing. When we talk about static scenarios we mean static graphs or road networks where each edge (road) has a specific weight. It is not a time-dependant network as railways or bus lines where departure and arrival time are those that determine the weight of an edge. In general, the SHARC query is a standard multi-level arc-flags Dijkstra, while the pre-processing melds ideas from the hierarchical approaches.

The pre-processing of SHARC combines some ideas from hierarchical approaches like highway hierarchies and REAL. Firstly, during the initialization phase, a two-core of the graph is extracted and a multilevel partition of G is performed. Then, an iterative process starts and at each step we shrink the graph by bypassing all unimportant edges and simultaneously setting the arc-flags to true for removed edge. On the contracted graph the arc-flags of each level (the iteration step in which the node was removed from the graph) are computed by growing a partial centralised shortest-path tree from each cell. After each step finishes, the input is pruned by finding which of the edges have their arc-flag set to false. Lastly, in the finalization phase, the output graph is assembled and the arc-flags of the edges that were removed during the contraction are refined. Each phase is explained separately in the following text, emphasizing more on the partition of the graph, the contraction and the arc-flags as those three are the most important operations that take place in the SHARC routing [4].

- *Partitioning*: The graph is partitioned every one thousand nodes and each partition that is created has a node called boundary node and it is the node via which the nodes of a partition can connect to the nodes of other partitions in the graph. The partitioning of the graph is a very crucial step as some of the arc-flags are set here and the speed-up heavily depends on them.

- *Contraction*: The graph is contracted by bypassing its nodes iteratively until there is no other node to bypass. For a node to be bypassed, first the node n, then its incoming and outgoing edges are removed from the graph. Then, for each node u being the tail of an incoming edge and for each v being the head of an outgoing edge, we introduce a new edge (shortcut) of length len(u,n) + len(n,v). If an edge connecting u and v already exists we only keep the one with the smaller length. The number of edges of the path that a shortcut represents on the graph at the beginning of the current iteration step is called the hop number of the shortcut. A node being bypassed affects the key of its neighbours and their bypass-ability too.

For that reason, the order in which nodes are bypassed changes the final contracted graph.

To sum up, contraction has two different reduction types. The node-reduction where unimportant nodes are pruned as described above and edge-reduction where edges are pruned and shortcuts are added to preserve the distances between the remaining nodes (Figure 4.6). Hence, directly after the node-reduction and edge-reduction is performed.

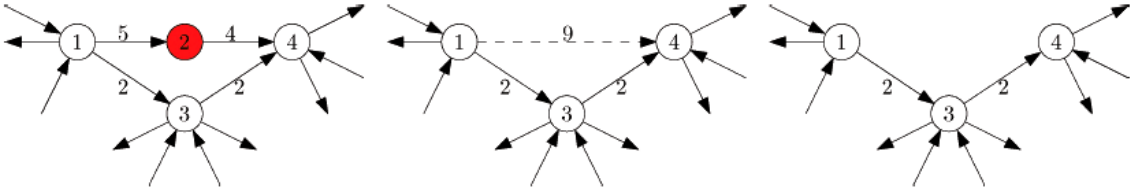


Figure 4.6: Edge reduction, node 2 is removed and a shortcut with weight 9 is added. However, the shortest path from 1 to 4 is via node 3. So shortcut (1,4) is removed. Source Ref: [4]

- *Boundary-Shortcuts*: An advantage of SHARC routing over the arc-flags is that the contraction routine decreases the number of hops of shortest paths in the network resulting in smaller search spaces. In order to further reduce the hop number, we add more shortcuts in the graph as pre-processing chooses paths with fewer hops. Hence, shortcuts between boundary nodes are added and pre-processing decides whether to use them in a path or not.

- *Arc-Flags*: The pre-processing of the Arc-flags is divided into two parts; partitioning and setting the arc-flags. Partitioning was discussed previously and in our implementation the number of the boundary nodes is kept very low as this adds on the speed-up. The next step that takes place during the pre-processing is the computation of the arc-flags. The approach used is called centralized and is a label-correcting algorithm (or centralized tree) that runs on every region (partition) in the graph and sets labels on all edges depicting the distances to the boundary node of the region. Those distances are updated continuously till no further improvement can be made.

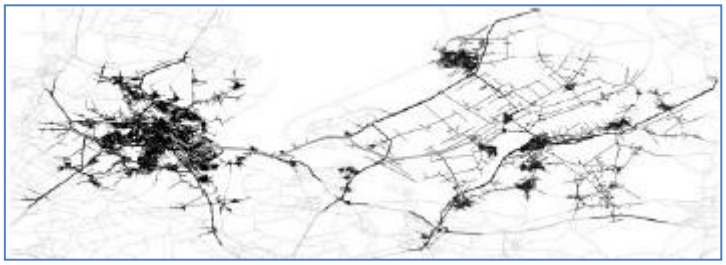
- *Output graph*: The output graph that comes from the pre-processing is the original graph enhanced by all the shortcuts that are in the contracted graph at the end of at least one iteration step. Hence, all edge flags that are still false after the last iteration step can be pruned.

It has to be mentioned that SHARC improves the time-consuming pre-processing that other speed-up techniques have such as multi-level arc-flags because of the integration of contraction in SHARC.

The query of SHARC is basically a multilevel arc-flags Dijkstra adapted from the two-level arc-flags Dijkstra presented in Mohring et al. [4]. The query is a modified Dijkstra which operates on the output graph. These modifications are: Firstly, the lowest

level i, on which the node u and the target node t are in the same region, is computed when node u is settled. Secondly, by relaxing the outgoing edges from node u, only the edges having an arc-flag set in level i for the corresponding region of t are considered [4].

SHARC routing was the one chosen to be implemented because its search space was the smallest than all other speed-up techniques. This is shown in Figures 4.7 and Figure 4.8.



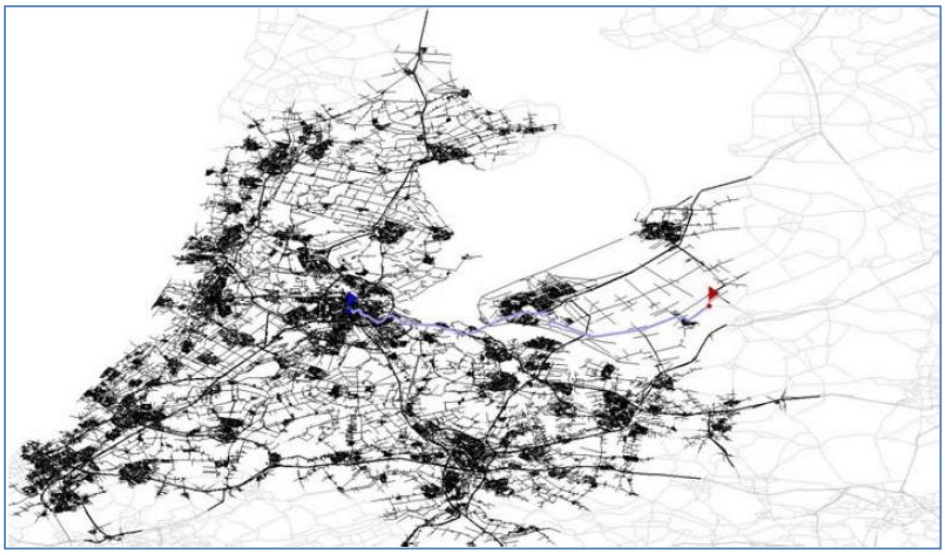
Bidirectional Dijkstra

Bidirectional ALT

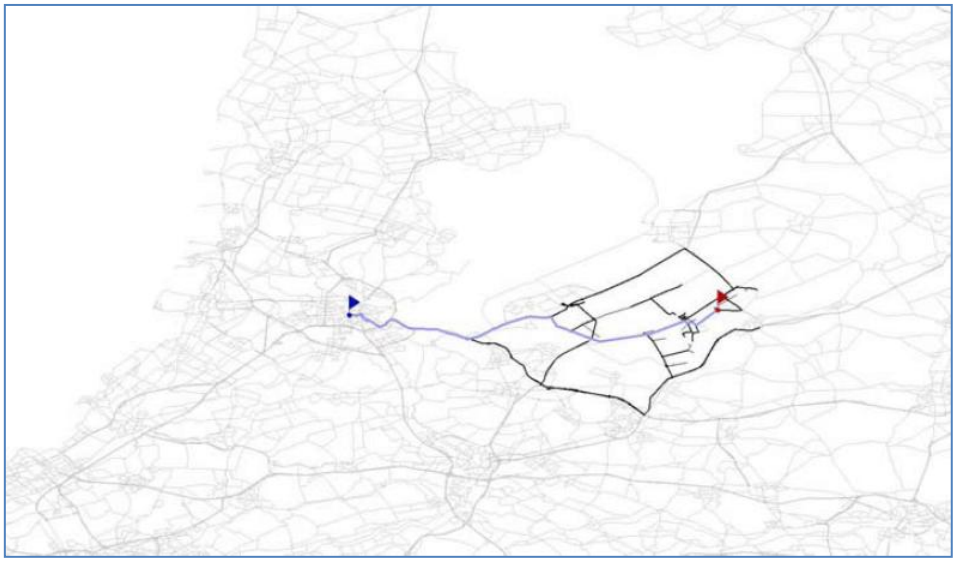
Highway-Node routing

SHARC routing

Figure 4.7: search space of the examined speed-ups, Source Ref: [35]



(a)



(b)

Figure 4.8: the blue mark is the source, the red mark is the target, (a) search space of Dijkstra, (b) search space of SHARC, Source Ref: [36]

Figures 4.7, 4.8 show the search space of some of the mentioned speed-up techniques running on a road network the same query. A black edge represents that the query algorithm has relaxed this edge. We must also note that in SHARC and Highway-Node routing representations shortcuts are unpacked and inserted into the graph for visualization purposes. As a result the search space may look a bit bigger than the search space of other techniques, but the number of relaxed edges may be smaller [35,36].

4.3.2 Reach values

Reach is a very recent approach and it is a centrality measure used to minimize the search space [38]. Hence, a node in the graph is important for shortest paths, if it is located in the middle of long shortest paths. Nodes that are located at the beginning or the end of long shortest paths are less central. The figure below depicts a path between a start *s* and a target node *t*. The reach in the path *P* of a node *u* is the minimum between two lengths; that of P_{su} and of P_{ut}. The reach *r(u)* is defined as the maximum reach of all shortest *s* – *t* paths in the graph *G* containing *u* [37] (Figure 4.9).

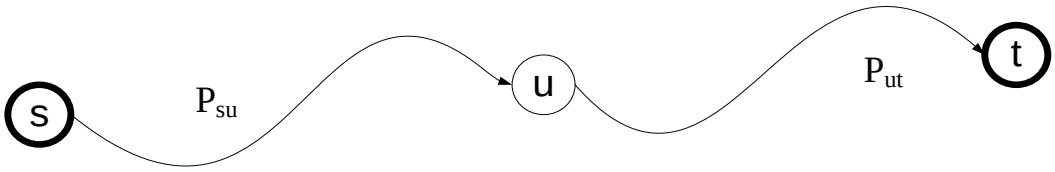


Figure 4.9: The reach of a node u

The notion of "length" or "distance" used here is not necessarily the same as the notion "weight" or "cost". This means that the length of a path in not necessarily the same as weight or cost of the path. For example, in a road network travel time usually is used as the cost metric and the edge weights reflect travel time and not travel distance. In our implementation the reach metric is the travel distance as we make use of static graphs and not time-dependant.

Reach formal definition: Given a weighted graph G = (V,E) and a shortest s-t path P, the reach of a node u on the path P is defined as r(u,P) = min{l_{Psu}, l_{Put}} where P_{su} and P_{ut} the sub-paths of P from s to u and from u to t (Figure). The reach r(u) of node u is defined as the maximum reach for all shortest s-t paths in the graph G containing u [37,38].

When searching for a shortest path $P_{st,}$ we can ignore a node u if the distance l_{Psu} from s to u is larger than the reach of u r(u) and the distance l_{Put} from u to t is larger than the reach of u. While the Dijkstra algorithm is running the first condition can be easily checked. For the second one, we know that it stands if among the reach and a lower bound of the distance from u to t, the reach is smaller. For the reach of all nodes to be computed a single-source all-target computation of the shortest path for every node is performed. In order to decide whether a node will be pruned from the graph or not the following inequality is checked (if it is true, then w is pruned, Figure 4.10) [37]:

$$r(w) < \min(d(u) + \ell(u, w), \; LB(w, \; t))$$

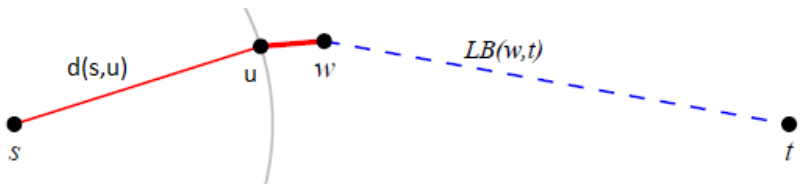


Figure 4.10: Reach algorithm, Source Ref: [38]

As reported in [38] a speed-up of up to 20 can be succeeded when using reach-based routing (Figure 4.11).

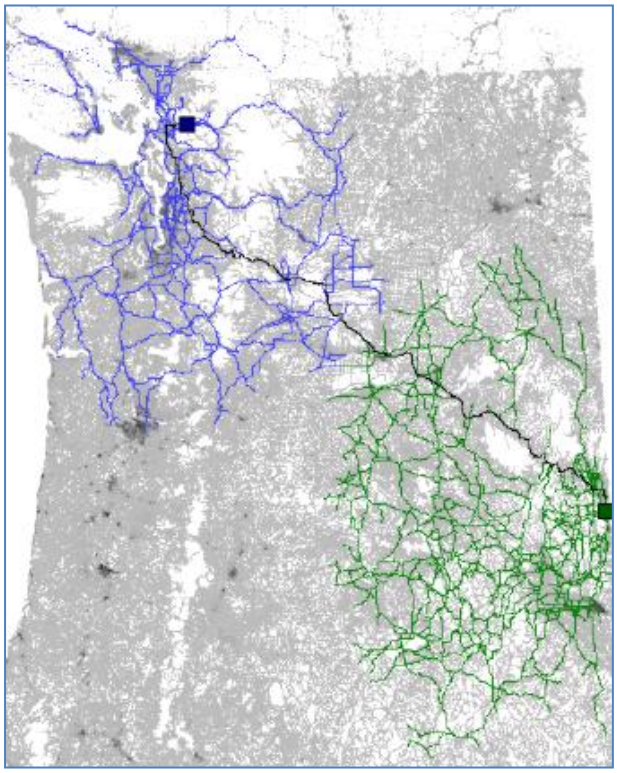


Figure 4.11: Reach algorithm,
Source Ref: [37]

The algorithm for reach values implemented in this project follows the following: Let G be a directed graph G=(V,E) with positive weights and reach metric m consistent with distance function d: (VxV)->R. The algorithm is a modified Dijkstra algorithm in which a function test(u) is called whenever a node is inserted into the priority queue. If test returns true, the node is inserted into the priority queue, otherwise is not inserted. In [38] it is proved that the use of the test function reduces the number of insertions into the priority queue so this means less pre-processing time. The function test uses the following [38]:

- r(u,G): the maximum value of r(u,Q) over all least-cost paths Q in graph G containing u.
- m(P): where P is the computed path from the start node s to u at the time u is to be inserted into the priority queue
- d(u,t): the distance between u and the destination t

The return value of test is true if $r(w) < min(d(u) + \ell(u,w), LB(w, t))$, otherwise is false.

The performance of the algorithm as in [38] is explained depends on the number of nodes which are rejected by the test function. Moreover, it depends on the distribution of the values of reach for the nodes in the graph G. In road networks and generally in networks with a high degree of hierarchy (), most nodes have low reach values (short reaches) and only few of them have high reach values (long reach). The distribution mentioned, fluctuates an exponentially decreasing function of reach.

Finally, the reach algorithm developed in the present project is less intricate than the one presented in [38] as it embeds the function calculated in the test function in the main code part of reach and thus it is not isolated in a test function.

Chapter 5

The new speed-up
SHARC & Reach values combination

In this chapter some of the most important snippets of code of the new algorithm, written in Java, will be analyzed. Furthermore, the way in which this new speed-up was embedded in the two implementations of Dijkstra with binary heap and Dijkstra with Fibonacci heap will be explained.

5.1 Overview of speed-up techniques and the *new* combination

In Figure 5.1, the red edges depict the three techniques combined in order to develop the new algorithm of SHARC + Reach values for this project. All the other speed-ups were implemented earlier this year and last year by [2,4]. Moreover, Figure 5.1 shows how hierarchical and goal-directed speed-up techniques can be combined. The speed-ups are drawn as nodes. The nodes on the left represent goal-directed techniques whereas those on the right represent hierarchical techniques.

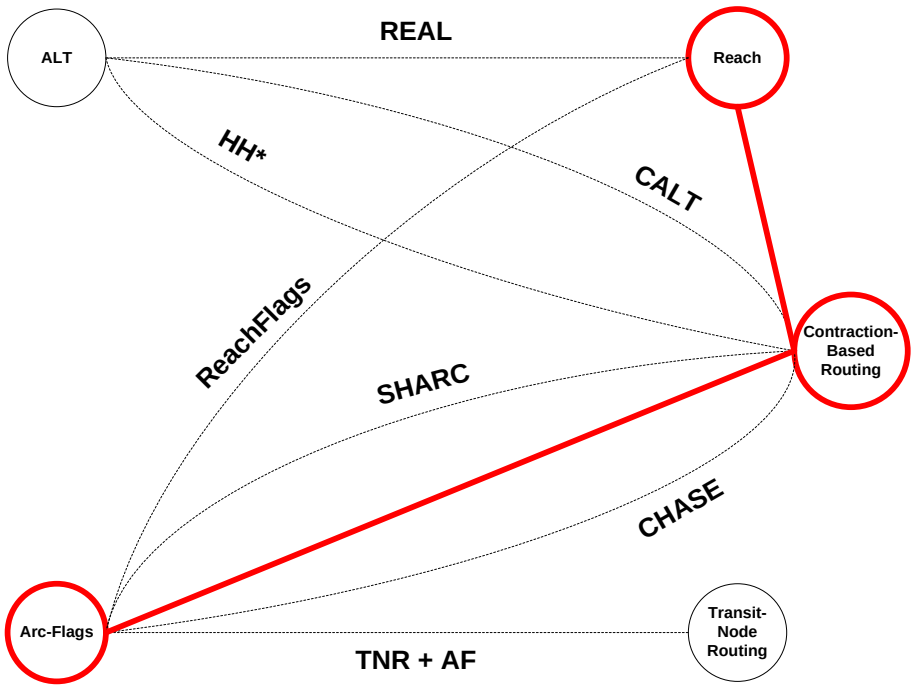


Figure 5.1: The new speed-up algorithm & older combinations of speed-up techniques,
Initial figure Source Ref: [2]

5.2 SHARC + Reach values Implementation

In this paragraph we refer to the most important parts of the code that was implemented for the combination of SHARC with Reach values along with further explanation.

5.2.1 Pre-processing

The partition of the graph takes place at the same time that the nodes are generated and inserted into an ArrayList. The variable *region* defines the partition in which the node of each iteration step will belong to:

```
for (int j = 0; j < n; j++) {
        if(j % region == 0 && j >= 2000) region = (j + region)/1000;
        Node v = new Node(j, -1.0, region, false, false, 0);
        vertices.add(v);
}
```

A method named SHARC is defined for the computation of the boundary nodes and the arc flag for every edge on the graph:

```
static void SHARC(int target) throws IOException{ .... }
```

When the partition of each node is computed; the first node that is found to have different partition number than the previous node is flagged as a boundary node:

```
while(node.hasNext()){
        ....
        if (vertex.getRegion() != vertices.get(index).getRegion()){
                        vertices.get(index).setBoundary(true);
                boundNodes[i] = vertices.get(index).data;
                ......
        }
        ......
}
```

Next, the arc-flag of its each edge is set to a default value *true* apart from the incoming and outgoing edges of the boundary nodes:

```
....
for(int k=0; k<regionCount-1; k++){
        if((arc.getSource().data != boundary[k]) ||
                (arc.getDestination().data != boundary[k])){
                if(arc.getSource().getRegion() ==
                arc.getDestination().getRegion()){
                        arc.setArcflag(true);
                        }
                }
}
```

Here, a method named *SharcExec* is called and in that method apart from the contraction routine that takes place, the reach values are also computed but it will be analysed in more details later on. After the call to the *SharcExec* method a iteration runs. This iteration scans all the edges in the graph and if the tail and the head of the edge belong to the shortest s-t path and are of the same region (partition) then that edge's arc-flag is set to *true*:

```
.....
SharcExec(vertex, SharcTarget);
.....
while(edge.hasNext()){
        Edge arc2 = edge.next();
        if((arc2.getSource().state == State.UNLABELED) &&
                (arc2.getDestination().state == State.UNLABELED)){
                if(vertex.getRegion() == arc2.getDestination().getRegion()
                && vertex.getRegion() == arc2.getSource().getRegion()){
                        arc2.setArcflag(true);
                }
        }
        .....
}
```

One very important step is to change the labels of all nodes back to *State.UNLABELLED* before our algorithm exits the SHARC method. This must be done, so that the *exec* method that runs the query starts with the correct state for each node and that is all nodes unlabelled:

```
for(Node checknode : vertices){
        checknode.state = State.UNLABELED;
        checknode.children = null;
}
```

The computation of the reach values and some of the SHARC operations run in the *SharcExec* method. This method is called on every partition of the graph and its main function is to find the shortest s-t paths from all nodes in that partition to the boundary node. At the beginning the keys (distance between the start and the current node) of all nodes have to be set to infinity. For the algorithm infinity is the negative double -1.0:

```
private static void SharcExec(Node start, int target) throws IOException
{
   for(Node checknode : vertices){
        checknode.key = -1.0;
}
```

In this iteration, if the node that enters the heap has the flag deleted set to *true* then the iteration continues to the next node. In that way, the pre-processing time drops further as the rest of the lines in the code are not executed for the deleted node:

```
do
{
      ......
      if(v.getDeleted() == true){
            continue;
       }
       .....
} while (!heap.isEmpty());
```

Next, we have the computation of Reach for every node in the graph. All the *if* conditions used, implement the inequality introduced in 4.3.2 paragraph.

```
......
double value = Math.min(start.key - temp.key,temp.key);
if (value > temp.getReach()){
      temp.setReach(value);
}
else continue;
for (int i = 0; i < temp.incomingEdges.size(); i++) {
      Edge currentEdge = temp.incomingEdges.get(i);
      if (currentEdge.tail == temp.pred)
            usedEdge = currentEdge;
}
double value2 = Math.min(((start.key - temp.pred.key)
                  + usedEdge.length),(temp.key));
if((temp.getReach() < value2) && (temp.getBoundary() == false )
            &&  (temp != vertices.get(target))){
            temp.setDeleted(true);
}
```

The final step in the *SharcExec* method is this loop where we set the state of each node to the default value and any children created are also deleted. This is how the partition that was examined returns to its initial state. However, all the info (arc-flags, deleted nodes/edges, reach values) needed about each node and each edge is stored permanently in the two ArrayLists used; one for the vertices and one for the edges:

```
for(Node checknode : vertices){
      checknode.state = State.UNLABELED;
      checknode.children = null;
}
```

5.2.2 Query

Our query algorithm is a unidirectional one. Below the most important part of the query code is presented. The *if* condition that checks if the flag of an edge is *false* or the edge has been pruned, decreases the query time and speeds-up the Dijkstra algorithm either with binary or Fibonacci heap. The speed-up can be explained if we consider that less iterations take place in the *outerIterator* method so this leads us to fewer computations.

```
private static void outerIterator() throws IOException
{
    .....
    if(currentEdge.getArcflag() == false || currentEdge.getDeleted() == true){
        continue;
     }
    .....
}
```

5.2.3 General comments about the code

For the implementation of the new combination, the initial Dijkstra code had to be split in methods with one main method. In that way, we are able to more easily distinguish the parts that refer to the pre-processing and the query part. The variable *RegionCount* holds the total number of partitions in the graph and is used to stop the iteration process; where all partitions are scanned in order to find the shortest s-t path from all nodes to the current boundary node. All the nodes of a partition do not have predecessors in a different partition. The methods that have been created for implementing the new algorithm along with a description are as follows:

- ***public static void main****(String[ ] args) { }*
  In this method the input file that is going to be checked is passed as a parameter, the vertices are generated and the edges are created. Both of them are saved in *ArrayLists.* The *SHARC* and *exec* methods are called at the end of the *main* method so that the pre-processing and query function are executed respectively.

- ***static void SHARC****(int target) { }*
  The SHARC method is responsible for setting the arc-flags, pruning nodes and edges that are not used in any shortest path and adding shortcuts.

- ***private static void SharcExec****(Node start, int target) { }*
  The *SharcExec* method is called inside the body of *SHARC* method and is executed for every partition in the graph. In the beginning it does a normal search of the shortest path from all nodes to the boundary node for the current partition. Inside the iterations that the shortest path is calculated the reach value for each node is computed. In that way we prune unimportant nodes before the *SHARC* method does its own pruning on the same partition.

- ***private static void innerIterator****(Edge currentEdge, Node headOfCurrentEdge, Node v) { }*
  The innerIterator method inserts a vertex with infinite key in the heap or decreases the key of a vertex with finite key.

- ***private static void outerIterator****() { }*
  In the *outerIterator* method first the minimum path is deleted from the heap and then inside an iteration the *innerIterator* method is called. Here is where we check whether an edge has been deleted or has its arc-flag set to false. If yes then we continue the iteration without calling the *innerIterator* method.

- ***private static void exec****(int start, int target) { }*
  Finally, the exec is the method where in the beginning we insert the target node and then we start iterate over the heap till is empty. Moreover, this is the method that prints the results; the shortest path found if there is one and distance from the start to the target node.

5.3 Summary

In this chapter the new speed-up algorithm and its components were introduced. The reason for choosing those two speed-up techniques was that SHARC gives very good query times among other speed-up techniques as it can be seen in Table 3.11. Reach values do not seem to give good results when used individually. However, the combination of goal-directed and hierarchical techniques always gives the best results as mentioned in [2]. Moreover, because this combination has never been implemented in the past it does worth trying it as we would never assume that the combination is excellent only from a theoretical point of view. Testing is important in order to prove that a speed-up over route planning algorithms improves what is already implemented.

The most difficult part in the implementation was that there were not published papers explaining in detail either SHARC or Reach values. Hence, the hard part of the implementation was not only the novel algorithm's design but also the initial implementations of SHARC and Reach values.

In the next chapter, we report the experiments carried out and an analysis on these results.

Chapter 6

Experiments & Analysis

In this chapter an extensive experimental evaluation of the novel SHARC + Reach values algorithm will be presented. The new algorithm will be evaluated using various input files representing cities. The tests were executed on dual core of an Intel 1.73 GHz running Windows 7 64-bit. The machine has 3 GB of RAM and 1 MB of L2 cache. All the algorithms were compiled with Eclipse IDE Version: 1.2.2.20.

6.1 Test files format & use

The experimental evaluation is made through tests for the static networks. Here, we focus on road networks. As inputs we use three different test files that represent three different cities; a province of Austria, a province of Scotland and Pennsylvania. All the test data come from [39,40]. However, the test files needed altering so that the weight for each edge could be included in the file. A small Java program was written for this alteration in which a random function was called each time a new edge weight was added to the file. The test files consist of three columns; the first stands for the source node, the second stands for the target node and the last one as already mentioned stands for the edge weight. The files include the total number of the edges in the graph, whilst the nodes were generated when the algorithm was first run. The nodes appear as numbers. The first line in each file is a number that is passed as an input to the program in order for the ArrayList of nodes to be generated inside a small loop. Figure 6.1 demonstrates what a test file looks like.

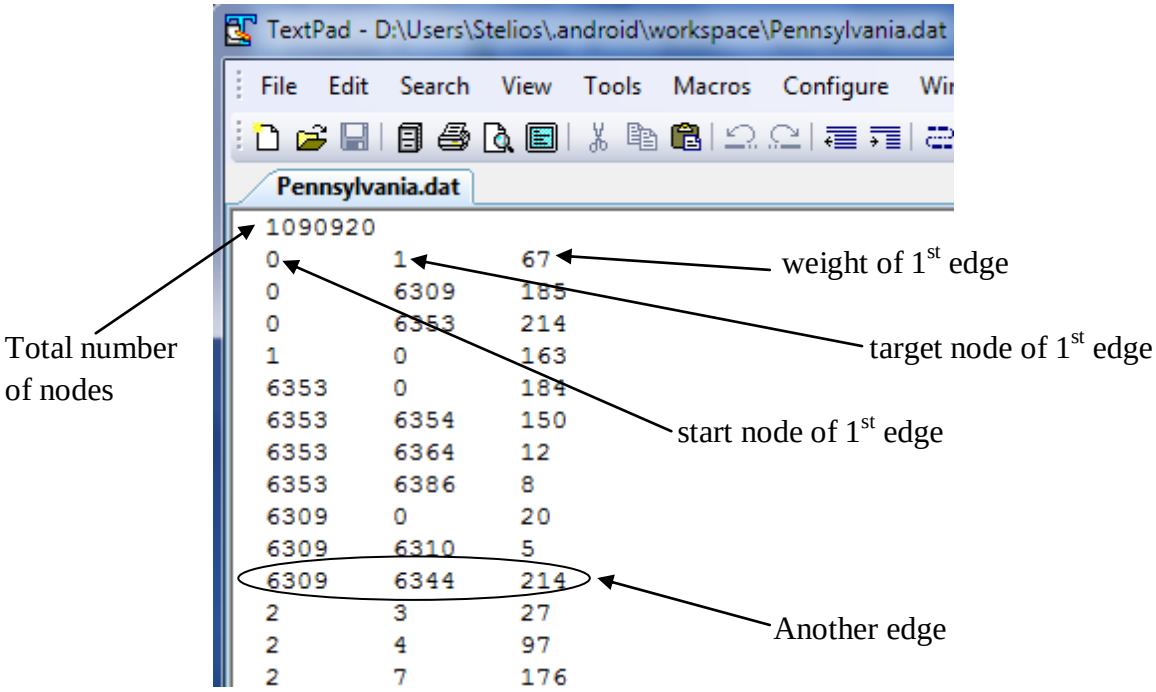


Figure 6.1: Pennsylvania test file

6.2 Results

In this section the results of the seven different algorithms, all implemented by the author, are presented and compared, over the three sets of data. All the outputs come from the console window of the Eclipse compiler.

Austria

In Figure 6.2, a province in Austria with 11648 nodes and 46160 edges is tested. After the file *Austria.dat* is loaded in our Dijkstra simplest implementation two arraylists are created; one for the vertices and one for the edges.

The number of the vertices and the edges follows.

Then, the calculation of the shortest path begins. Having 0 as the start and 11647 as the target node in the graph the result for the basic implementation is 8712 yards which is the shortest distance between the above nodes.

The path and the number of the vertices passed follow the distance result.

Finally, the time in milliseconds needed for the shortest path to be found is 127800ms (2.13 mins).

```
Shortest Path Algorithm(DijkstraSimple)

Loading ../austria.dat
Vertices loaded...
Edges loaded...
Vertices: 11648
Edges: 46160

Building shortest path tree...
Shortest Path found:
Distance: 8712.000000
0 - 111 - 222 - 332 - ..... - 11644 - 11646 - 11647
Vertices passed: 190
This run took 127800 ms
```

Figure 6.2

In Figure 6.3 (a) & (b) that follow it is shown that Dijkstra with Fibonacci heap runs faster than the simple Dijkstra or even the one with the binary heap. The query time drops from 127800ms on the simple Dijkstra to 93ms on the Dijkstra with binary heap to end up with 50ms on the Dijkstra with Fibonacci heap. At this point it must be mentioned that because in our binary heap implementation there is not a *decrease key* method the path and the distance and path results differ from those of the simple and the Fibonacci heap algorithm.

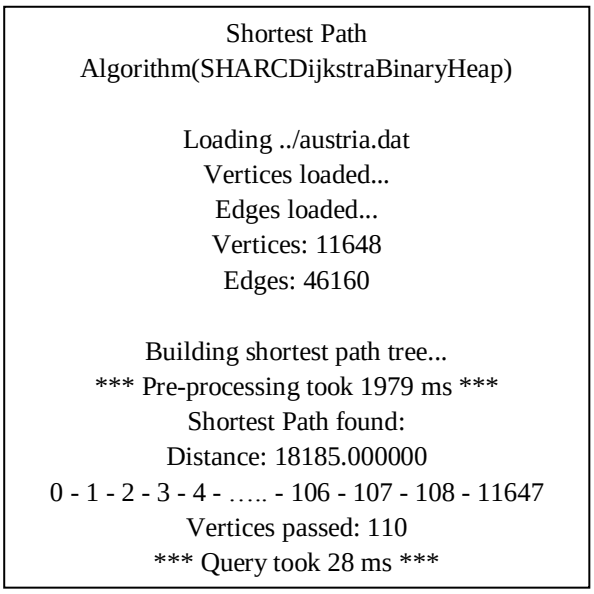
```
Shortest Path Algorithm(DijkstraBinaryHeap)

Loading ../austria.dat
Vertices loaded...
Edges loaded...
Vertices: 11648
Edges: 46160

Building shortest path tree...
Shortest Path found:
Distance: 18185.000000
0 - 1 - 2 - 3 - 4 -.....- 106 - 107 - 108 - 11647
Vertices passed: 110
This run took 93 ms
```

(a)

```
Shortest Path Algorithm(DijkstraFibonacciHeap)

Loading ../austria.dat
Vertices loaded...
Edges loaded...
Vertices: 11648
Edges: 46160

Building shortest path tree...
Shortest Path found:
Distance: 8712.000000
0 - 111 - 222 - 332 - ..... - 11644 - 11646 - 11647
Vertices passed: 190
This run took 50.0 ms
```

Figure 6.3 (b)

Figure 6.4 (a) & (b) show the speed-up that SHARC adds to the Dijkstra with binary and Fibonacci heap respectively. Pre-processing time appears here for the first time because at this point the arc-flags are set, unimportant nodes are pruned and shortcuts are added. Pre-processing for the SHARC-binary heap combination is 1979ms and 315ms is for the SHARC-Fibonacci heap. Query time drops further by adding SHARC, from 93ms to 28ms for binary heap and from 50ms to 24ms for Fibonacci heap. It can be observed that because the graph is not very dense, the time results are almost the same. This changes as the graph becomes bigger in the tests that follow.



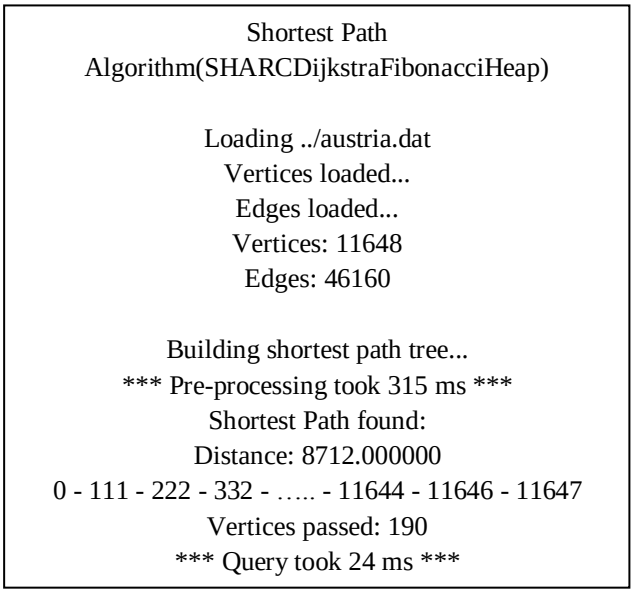
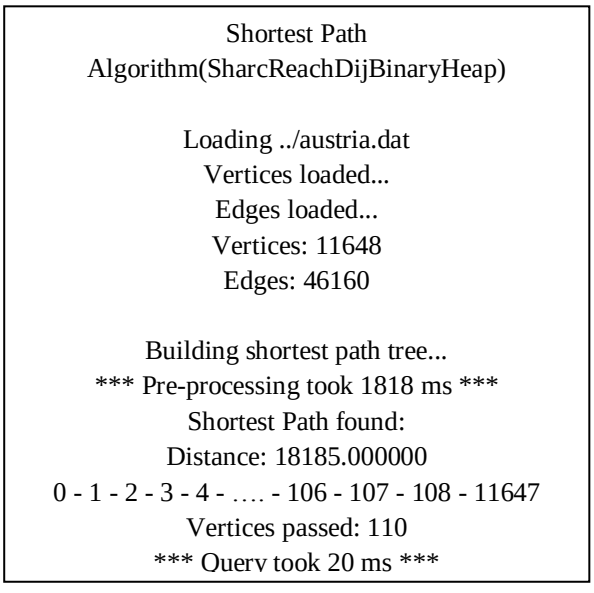


Figure 6.4 (a) (b)

Figure 6.5 (a) & (b) depict the results of the new algorithm (SHARC + Reach values) and it can be observed that there is a small improvement for the binary heap as pre-processing time drops from 1979ms to 1818ms and query time drops from 28ms to



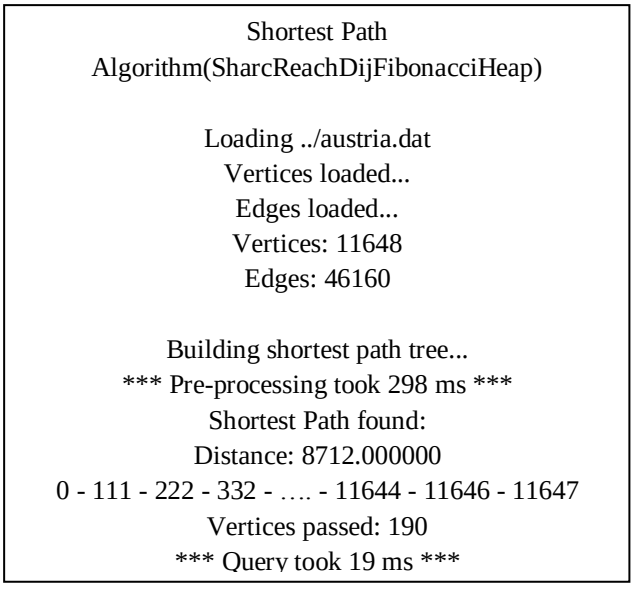


Figure 6.5 (a) (b)

20ms. Moreover, for the Fibonacci heap we have a pre-processing time improvement from 315ms to 298ms and query time from 24ms to 19ms. Hence, this demonstrates that the new algorithm provides a speed-up on the road network of the province in Austria.

Scotland

In Figure 6.6, a province in Scotland with 63360 nodes and 252432 edges is tested. After the file *Scotland.dat* is loaded in our Dijkstra simplest implementation two arraylists are created; one for the vertices and one for the edges.

The number of the vertices and the edges follows.

Then, the calculation of the shortest path begins. Having 0 as the start and 63359 as the target node in the graph the result for the basic implementation is 222 yards which is the shortest distance between the above nodes.

The path and the number of the vertices passed follow the distance result.

Finally, the time in milliseconds needed for the shortest path to be found is 702900ms (11.72 mins).

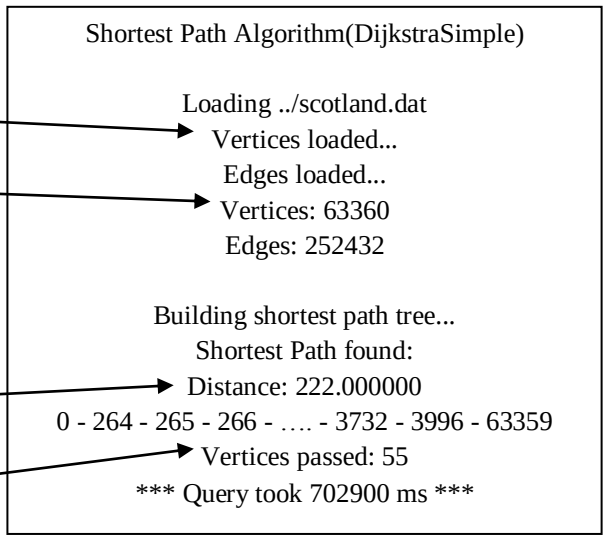
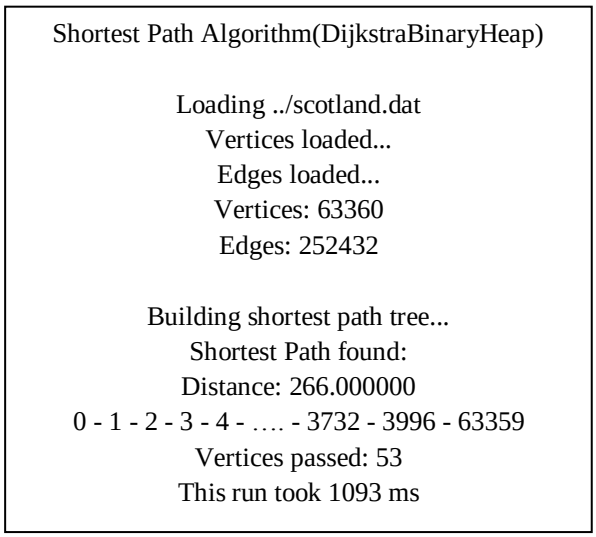


Figure 6.6

In Figures 6.7 (a) & (b) that follow it is shown that Dijkstra with Fibonacci heap runs faster than the simple Dijkstra or even the one with the binary heap. The query time drops from 702900ms on the simple Dijkstra to 1093ms on the Dijkstra with binary heap to end up with 213ms on the Dijkstra with Fibonacci heap.



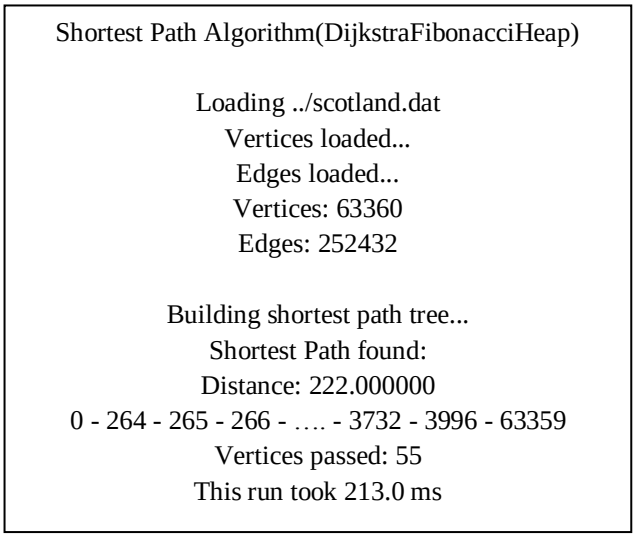
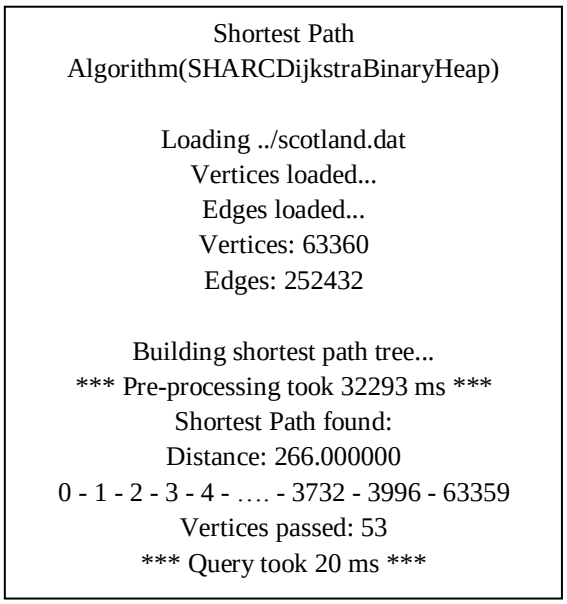


Figure 6.7 (a) (b)

In Figure 6.8 (a) & (b) the results show the speed-up that SHARC adds to the Dijkstra with binary and Fibonacci heap respectively. Pre-processing time appears here for the first time because at this point the arc-flags are set, unimportant nodes are pruned and shortcuts are added. Pre-processing for the SHARC-binary heap combination is 32293ms and 12610ms is for the SHARC-Fibonacci heap. Query time drops further by adding SHARC, from 1093ms to 20ms for binary heap and from 213ms to 14ms for Fibonacci heap. It can be observed that because the graph is denser than the previous one but not very large, the query times are close (20ms, 14ms). This changes as the graph becomes much bigger in the last test.



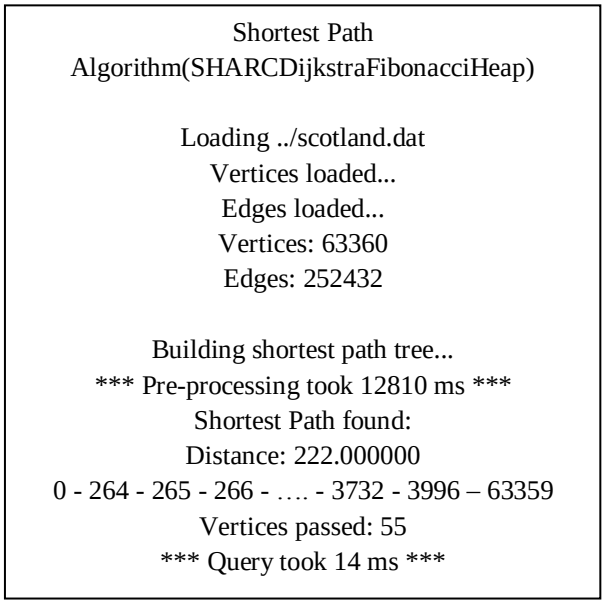
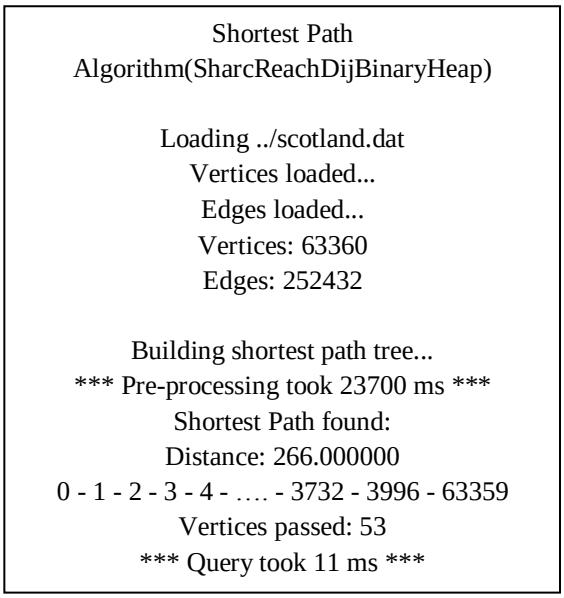


Figure 6.8 (a) (b)

Figure 6.9 (a)&(b) show the results of the new combination algorithm (SHARC + Reach



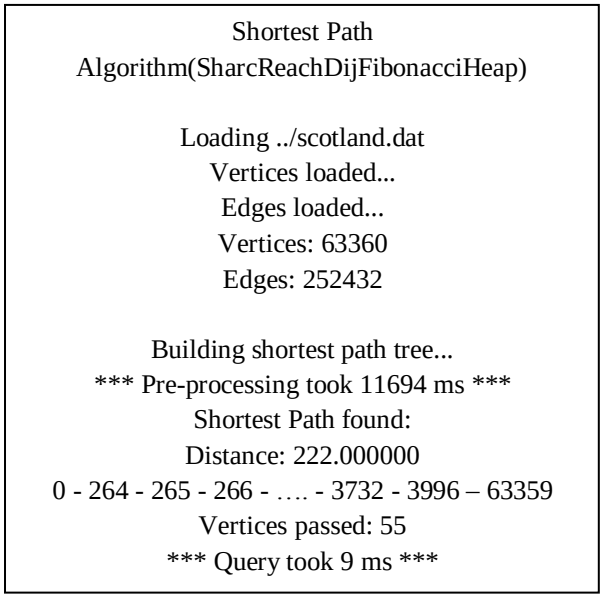


Figure 6.9 (a) (b)

values) for Scotland and it can be observed that there is once again an improvement for the binary heap as pre- processing time drops importantly from 32293ms to 23700ms and query time drops from 20ms to 11ms. Moreover, for the Fibonacci heap we have a pre-processing time improvement from 12810ms to 11694ms and query time from 14ms to 9ms. Hence, it demonstrates again that the new algorithm provides a speed-up on the road network of the province in Scotland too.

Pennsylvania

Finally, in Figure 6.10 the city of Pennsylvania with 1090920 nodes and 3083796 edges is tested. After the file Pennsylvania.*dat* is loaded in our Dijkstra simplest implementation two arraylists are created; one for the vertices and one for the edges.

The number of the vertices and the edges follows.

Then, the calculation of the shortest path begins. Having 0 as the start and 1090919 as the target node in the graph the result never appears within a reasonable time. The time in milliseconds needed for the shortest path to be found does not appear too. The simple Dijkstra needs approximately a couple of days to finish and find the shortest path in this network.

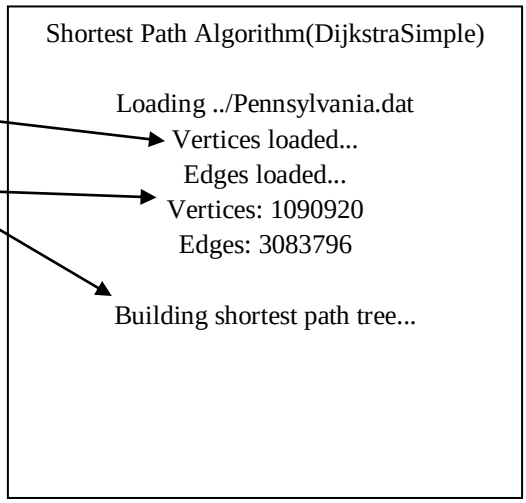
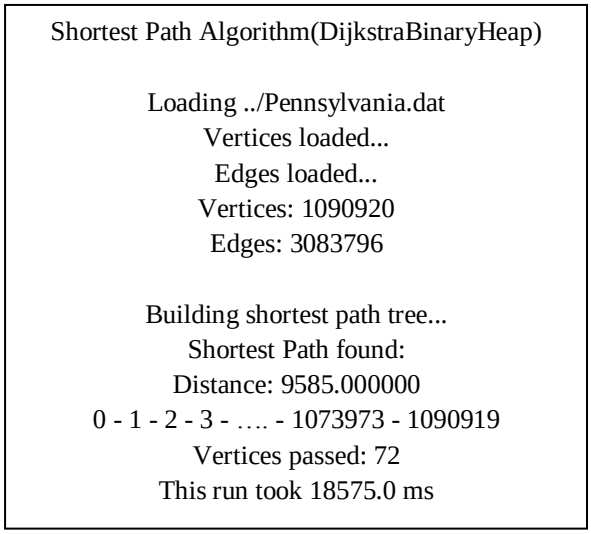


Figure 6.10

In Figure 6.11 (a) & (b) that follow it is shown that Dijkstra with Fibonacci heap runs faster than the simple Dijkstra or even the one with the binary heap. The query time drops from 18575ms on the Dijkstra with binary heap to 4035ms on the Dijkstra with Fibonacci heap.



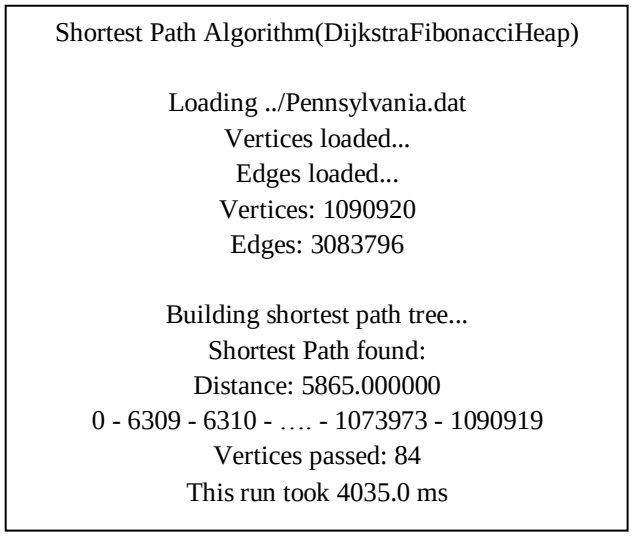


Figure 6.11 (a) (b)

Figure 6.12 (a) & (b) results show the speed-up that SHARC adds to the Dijkstra with binary and Fibonacci heap respectively. Pre-processing time appears here for the first time because at this point the arc-flags are set, unimportant nodes are pruned and shortcuts are added. Pre-processing for the SHARC-binary heap combination is 3340526ms and 2841676ms is for the SHARC-Fibonacci heap. Query time drops further by adding SHARC, from 18575ms to 98ms for binary heap and from 4035ms to 46ms for Fibonacci heap. It can be observed that the graph is much denser than the previous ones. Moreover, the binary heap implementation always remains slower than Fibonacci heap and this is reflected in both pre-processing and query time.

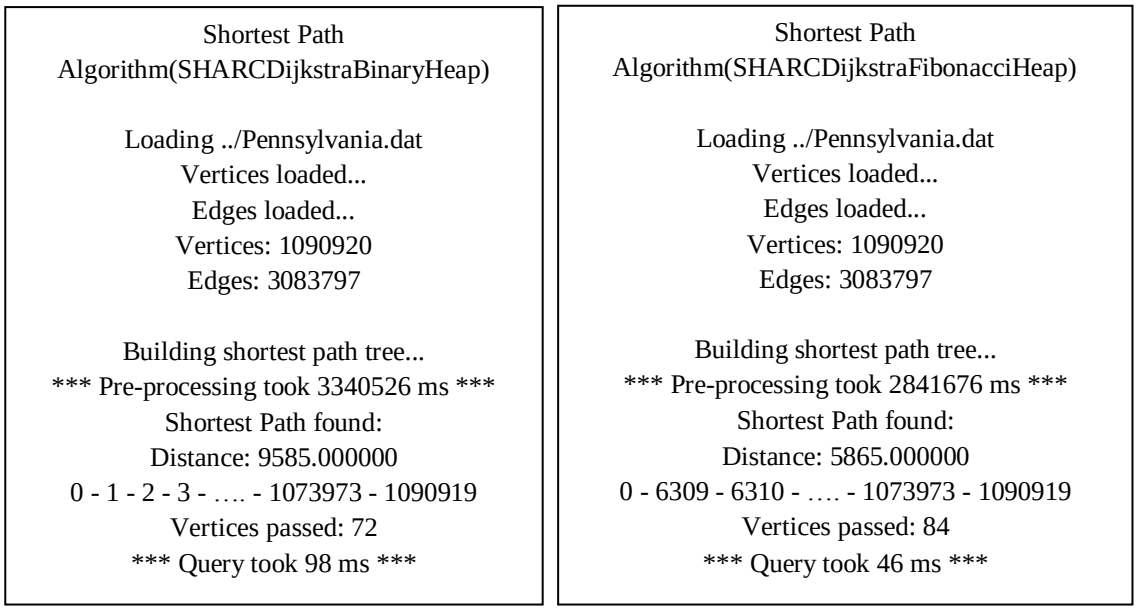


Figure 6.12 (a) (b)

Figure 6.13 (a)&(b), depicts the results of the new algorithm (SHARC + Reach

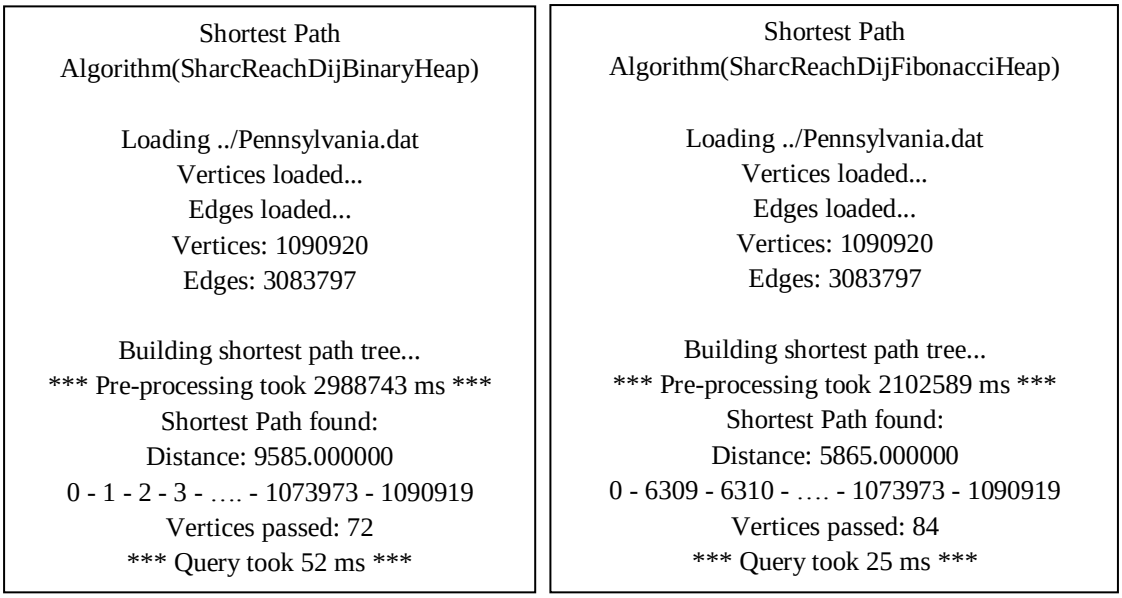


Figure 6.13 (a) (b)

values) for the city of Pennsylvania. There is an improvement for the binary heap as pre-processing time drops from 3340526 ms to 2988743ms and query time drops from 98ms to 52ms. Moreover, for the Fibonacci heap we have a pre-processing time improvement from 2841676ms to 2102589ms and query time from 46ms to 25ms. Hence, it demonstrates that the new algorithm provides a speed-up on the road network of city of Pennsylvania too.

6.3 Analysis

Figure 6.14 reports the results of the seven different implementations of Dijkstra's algorithm. It compares the three implementations of Dijkstra without any speed-up added, with Dijkstra Binary and Fibonacci heap along with the speed-up of SHARC and finally with the new combination of SHARC + Reach values. It can be

Start node: 0 Target node: n-1		Simple Dijkstra	Dijkstra Binary Heap	Dijkstra F-Heap	Dijkstra Binary Heap + SHARC	Dijkstra F-Heap + SHARC	**Dijkstra Binary Heap + SHARC + Reach**	**Dijkstra F-Heap + SHARC + Reach**
Austria Nodes: 11648 Edges: 46160	Pre-processing (ms)	0	0	0	1979	315	**1818**	**298**
	Query (ms)	127800 (2.13m)	93	50	28	24	**20**	**19**
Scotland Nodes: 63360 Edges: 252432	Pre-processing (ms)	0	0	0	32293	12610	**23700**	**11694**
	Query (ms)	702900 (11.72m)	1093	213	20	14	**11**	**9**
Pennsylvania Nodes: 1088092 Edges: 3083796	Pre-processing (ms)	0	0	0	3340526 (56m)	2841676 (47m)	**2988743 (50m)**	**2102589 (35m)**
	Query (ms)	*No result*	18575	4035	98	46	**52**	**25**

Figure 6.14: Comparison table of the seven algorithms developed for the needs of the project. It summarizes pre-processing and query times for the three cities.

observed that the new algorithm gives excellent pre-processing and query times. Moreover, we conclude that the binary heap implementation is slower than the Fibonacci heap and remains slower even when the speed-ups are added, that backs up the results found by [31,34]. Interestingly, the new speed-up (SHARC + Reach values) has a lower query time for Scotland than Austria but for the price of worst pre-processing time. This is due to the fact that as the graph becomes bigger more time is needed for it to be partitioned and more edges have to be pruned. The city of Pennsylvania gives the highest pre-processing and query times as the graph not only has the biggest number of nodes but also the most massive number of edges. However query times of Pennsylvania do not differ a lot from those of the other two cities. This is due to the fact that pre-processing is doing most of the computations and only a small number of iterations take place during the query. Inside these iterations a condition that checks whether an edge is important or not and also checks the condition of the nodes (whether they are pruned or not) minimizes the number of calculations.

6.4 Summary

In every scenario, Dijkstra with Fibonacci Heap speeded-up with the new algorithm (SHARC + Reach values) gives us the best results with the smallest pre-processing and query time. All processing that the target machine was doing was limited where possible to just the algorithm, however variations in runtime were still inevitable. Therefore all the results come from an average of 30 runs of each of the seven algorithms on each data set. In the graphical representation that follows the range of the results (pre-processing, query time) from our measurements can be seen along with their averages. The range is represented by the red lines where the lowest point is the smallest measurement and the peak is the highest one. Figures 6.15 & 6.16 show the averages.

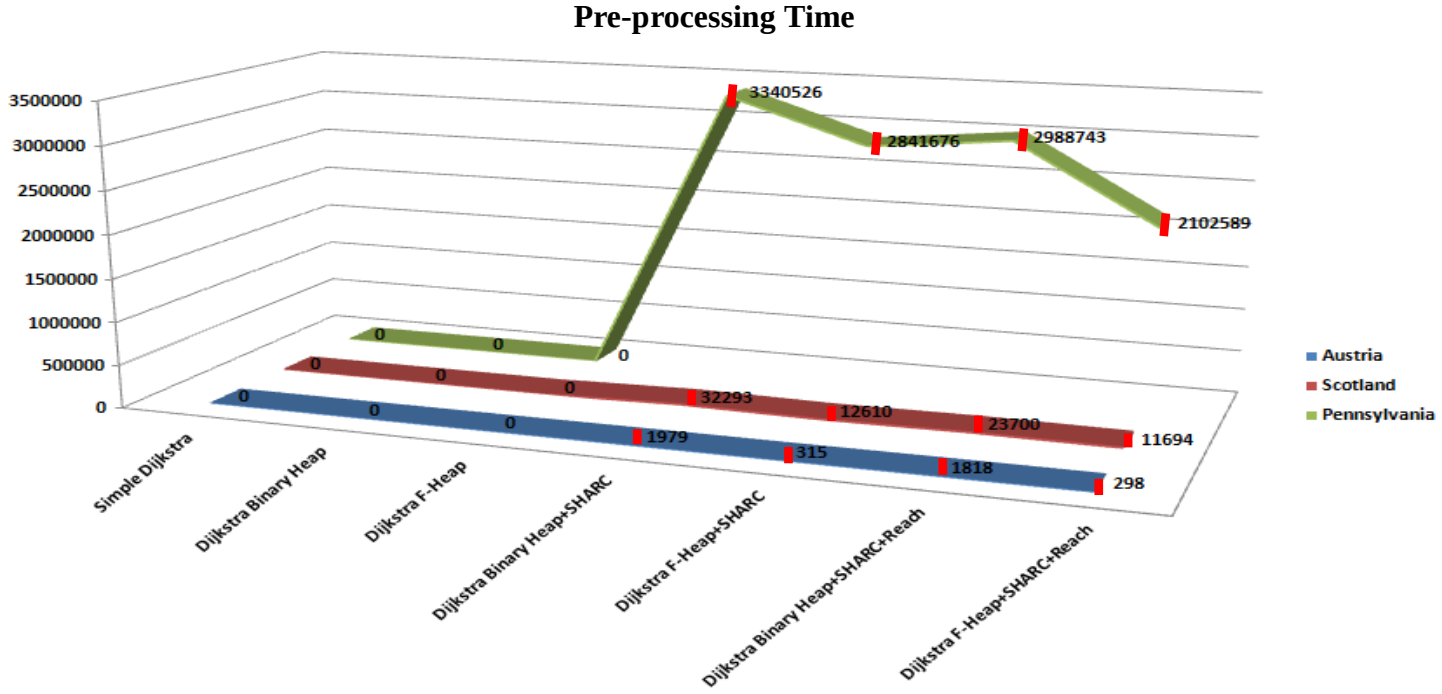


Figure 6.15: Pre-processing time chart along with the range of the measurements; lowest and highest value

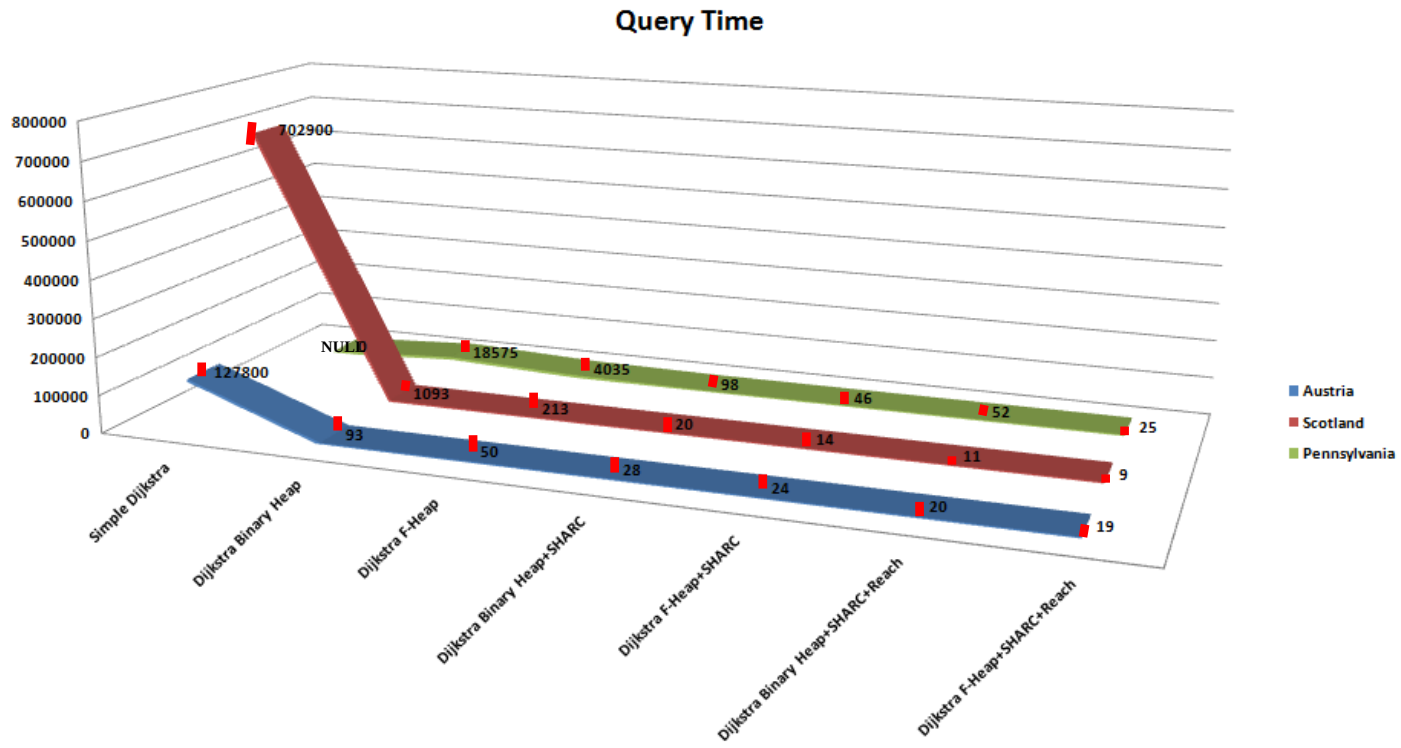


Figure 6.16: Query time chart along with the range of the measurements; lowest and highest value

Chapter 7

Conclusions & Future Work

In this project we introduced a new algorithmic approach which combines goal-directed with hierarchical methods; SHARC and Reach values. This new approach can be interpreted as a unidirectional algorithm because our implementation of SHARC is a unidirectional approach. As a result, a faster combination than SHARC is presented for different scenarios and inputs. The new approach yields excellent speed-ups in denser graphs. In more sparse road networks it does speed-up but with a smaller rate. In this project once again [2] it is proved that when goal-directed and hierarchical methods are combined, we end up with techniques that single out for their robustness to the input. When the implementation of the algorithms reached to the point where SHARC would be added we knew that it would speed-up Dijkstra. However, the pre-processing time appeared to be large. This drawback was overcome by embedding the Reach approach in our SHARC. The use of Reach values helps us prune unimportant edges and nodes from the graph during the pre-processing in a different way than SHARC that uses the contraction method. Hence, we end up with less iterations during pre-processing and query.

Furthermore, by using real-world inputs for the experiments and not staying only in the theoretical approach of the algorithms we get more meaningful results. Before 2005, it was really hard to find online sources providing real big road networks being represented as graphs. In this project real-world road networks were used [39,40].

Regarding future work, we can extend the idea to different directions. The new algorithm that combines SHARC with Reach values in static graphs can be implemented and tested on time-dependant graphs. Hence, the idea can be extended from road networks to bus, rail networks where departure, arrival time and the duration of the trip along with possible delays are the parameters that have to be examined. Some techniques have already been implemented for the computation of time-dependent shortest paths with very encouraging results [36]. These techniques are the Core-ALT, SHARC and Contraction Hierarchies, all from a time-dependant view, not static. Additionally, instead of using the standard SHARC on the new algorithm other time-dependant variants of SHARC can be tested. The aggressive variant of SHARC uses exact flags during pre-processing so it tends to have long pre-processing times combined with a better quality of flags. The economical version of SHARC uses approximate flags and has shorter pre-processing times for the price of worst flags, while heuristic SHARC uses heuristic flags and cannot guarantee the correctness of the queries [36].

Another idea is to embed the new algorithm into a practical application such as a journey planner. This would give a better idea on how the new speed-up approach utilizes a real-world application that can also make use of other parameters such as weather conditions and CO2 emissions.

Moreover, it would be interesting to test different partitioning levels to optimize the new algorithm. The point at which we partition the graph in our approach a significant role plays the level of the partition. This means that we can make use of different partition levels that can further reduce our pre-processing and query times. Currently we make use of a one-level partition so there is more space to test two-level partitions or even more. Two-level partition means, as described in [41], that after we

partition the graph once we take each region and partition it again. This technique may seem to increase the number of iterations in the graph but it prunes unimportant edges and nodes at an early stage so there is no need to settle them again when we reach the first level partition. In this case more, refinements of the graph need to run as it changes all the time.

****

Bibliography

[1] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner, Engineering Route Planning Algorithms, Universitat Karlsruhe (TH), 76128 Karlsruhe, Germany, pp. 117–139, 2009

[2] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Chieferdecker, Dominic Schultes and Dorothea Wagner, Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra.s Algorithm, Karlsruhe Institute of Technology, February 2010

[3] Roch Guérin, Fellow, IEEE, and Ariel Orda, Senior Member, IEEE, Computing Shortest Paths for Any Number of Hops, 2002

[4] Reinhard Bauer, Daniel Delling, Universitat Karlsruhe (TH), SHARC: Fast and Robust Unidirectional Routing, May 2009

[5] Liang Dai, Dr. Anil Maheshwari, Fast Shortest Path Algorithm for Road Network and Implementation, Honours Project, Fall Term, 2005

[6] P. Biswas, P. K. Mishra and N. C. Mahanti, Computational Efficiency of Optimized Shortest Path Algorithms, Department of Applied Mathematics, Birla Institute of Technology, Mesra (India), 2005

[7] Jon Sneyers, Tom Schrijvers, Bart Demoen, K.U.Leuven, Belgium, Dijkstra's Algorithm with Fibonacci Heaps: An Executable Description in CHR, February 2006

[8] Timothy M. Chan, More Algorithms for All-Pairs Shortest Paths in Weighted Graphs, School of Computer Science, University of Waterloo, Canada, June 2007

[9] Ph.D. Zoltán A.Vattai, Floyd-Warshall Again, Budapest University of Technology and Economics, 1996

[10] Bjorn Zenker, Bernd Ludwig, ROSE – Assisting Pedestrians to Find Preferred Events and Comfortable Public Transport Connections, September 2009

[11] Paul André, Max L. Wilson, Alisdair Owens, Daniel Alexander Smith, Journey Planning Based on User Needs, May 2007

[12] Dejian Meng, Stefan Poslad, A Reflective Context-Aware System for Spatial Routing Applications, December 2008

[13] Gilly Leshed, Theresa Velden, Oya Rieger, Blazej Kot, Phoebe Sengers, In-car gps navigation: engagement with and disengagement from the environment, April 2008

[14] Abolghasem Sadeghi Niaraki, Kyehyun Kim, Ontology based personalized route planning system using a multi-criteria decision making approach, March 2009

[15] Jongchan Lee, Sanghyun Park, Minkoo Seo, Sang-Wook Kim, ACE-INPUTS: A Cost-Effective Intelligent Public Transportation System, August 2007

[16] Jeppesen Rail, Logistics and Terminals, IPTIS - Public transport journey planning

[17] Floyd's algorithm, accessed on 05/05/10: University of Auckland: http://www.cs.auckland.ac.nz/~ute/220ft/graphalg/node21.html, http://www.fearme.com/misc/alg/node88.html

[18] Van Emde Boas, P., Kaas, R., and Zijlstra, E. Design and implementation of an efficient priority queue. Math. Syst. Theory 10 (1977), 99-127.

[19] Dial, R. B. (1969) Algorithm 360: Shortest Path Forest with Topological Ordering. Communications of the ACM, 12, 632-633

[20] Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures University of Western Ontario, accessed on 05/05/10: http://publish.uwo.ca/~jmalczew/gida_1/Zhan/Zhan.htm#References

[21] Anthony (Tony) Stentz, The D* Algorithm for Real-Time Planning of Optimal Traverses, 1994

[22] Mario Scheu, Single-Source Shortest Paths (One-to-all shortest path), 2001

[23] Gilles Cazelais, Prim's Algorithm, December 2006.

[24] Graham, R. L. and Hell, P. "On the History of the Minimum Spanning Tree Problem." Ann. History Comput. 7, 43-57, 1985

[25] Univeristy of Patra, accessed on 05/05/10: http://students.ceid.upatras.gr/~papagel/project/kruskal.htm

[26] Erik Reinhard, More Dynamic Programming Floyd-Warshall Algorithm, University of Central Florida, http://www.cs.ucf.edu/~reinhard/classes/cop3503/lectures/DynProg02.pdf

[27] Johnson's Algorithm, accessed on 05/05/10: http://www.itl.nist.gov/div897/sqg/dads/HTML/johnsonsAlgorithm.html, National Institute of standards & technology

[28] Michael T. Goodrich and Roberto Tamassia, Algorithm Design Foundations, Analysis, and Internet Examples, Chapter 7

[29] Chang-Ju Lin, Johnson's algorithm, http://www.caveshadow.com/CS566/Chang-Ju%20Lin%20-%20Johnson%27s%20Algorithm.ppt

[30] Lars Vogel, Dijkstra's Shortest Path Algorithm in Java, 2009, http://www.vogella.de/articles/JavaAlgorithmsDijkstra/article.html

[31] Mark Allen Weiss, Priority Queue - Binary Heap - Implementation in Java, http://www.java-tips.org/java-se-tips/java.lang/priority-queue-binary-heap-implementation-in-3.html

[32] Dijkstra's Algorithm for Network Optimization Using Fibonacci Heaps, 2009, http://www.codeproject.com/KB/recipes/Dijkstras_Algorithm.aspx

[33] Dominik Schultes, Route Planning in Road Networks, 2008, Route Planning in Road Networks, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.4401&rep=rep1&type=pdf

[34] Michael L Fredman & Robert Endre Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, journal 1987

[35] Reinhard Bauer, Daniel Delling, and Dorothea Wagner, Experimental Study of Speed Up Techniques for Timetable Information Systems, 2010

[36] Daniel Delling and Dorothea Wagner, Time-Dependent Route Planning, 2009

[37] Andrew V. Goldberg, Reach for A∗: an Efficient Point-to-Point Shortest Path Algorithm, Microsoft Research, 2009

[38] Ron Gutman, Reach-based Routing: A new approach to shortest path algorithms optimized for road networks, January 2004

[39] CNOP - A Package for Constrained Network Optimization, http://www.ziegelmann.org/ziegelmann/CNOP_data.htm

[40] Jure Leskovec, Stanford large network dataset collection, http://snap.stanford.edu/data/

[41] Rolf H. Mohring and Heiko Schilling & Birk Schutz, Dorothea Wagner, and Thomas Willhalm, Partitioning Graphs to Speedup Dijkstra's Algorithm, 2006

[42] Robert Lafore, Data structures and algorithms in Java, Second edition

Appendix: Source Code

****

SharcReachDijFibonacciHeap.java

****

```
public class SharcReachDijFibonacciHeap {

	private static ArrayList<Node> vertices = new ArrayList<Node>();
	private static ArrayList<Edge> edges = new ArrayList<Edge>();
	private static int n = 0;
	private static FibonacciHeap heap = new FibonacciHeap();

	public static void main(String[] args) throws NumberFormatException,
	IOException {

		System.out.print("Shortest Path Algorithm(SharcReachDijFibonacciHeap)\n\n");

		if (args[1] == null) {
			System.out
			.print("1 Argument required: shortestpath.exe [graph file name]\n");
			System.out.print("Example: ./Dijkstra.exe scotland_big.dat\n");
		}

		// Read source file
		System.out.printf("Loading %s\n", args[1]);
		BufferedReader indat = new BufferedReader(new FileReader(args[1]));
		String inp = null;

		if (indat != null) {
			inp = indat.readLine();
			n = Integer.parseInt(inp);
			int region = 1;
			for (int j = 0; j < n; j++) {
				if(j % region == 0 && j >= 2000) region = (j + region)/1000;
				Node v = new Node(j, -1.0, region, false, false, 0);

				vertices.add(v);
			}

			System.out.print("Vertices loaded...\n");

			vertices.get(n - 1).state = State.LABELED;
			// Read edges and initialize
			while (indat.ready() == true) {
				inp = indat.readLine();
				StringTokenizer st = new StringTokenizer(inp);

				String tail = st.nextToken();
				String head = st.nextToken();
				String length = st.nextToken();

				int tail2 = Integer.parseInt(tail);
				int head2 = Integer.parseInt(head);
				double length2 = Double.parseDouble(length);
				Edge edge = new Edge(vertices.get(tail2), vertices.get(head2), length2,
				false, false);
				edge.head.addIncomingEdge(edge);
				edge.tail.addOutgoingEdge(edge);
				edges.add(edge);
			}
		} else {
			System.out.print("Could not open input data...\n");
		}

		System.out.print("Edges loaded...\n");
		System.out.printf("Vertices: %d\nEdges: %d\n\n", vertices.size(), edges
				.size());
		System.out.print("Building shortest path tree...\n");

		int start = 0;
		int target = n-1;
		ExecutionTime.startTime();
		SHARC(target);
		System.out.println("*** Pre-processing took " + ExecutionTime.endTime() + " ms ***");
		ExecutionTime.startTime();
		exec(start, target);
		System.out.println("*** Query took " + ExecutionTime.endTime() + " ms ***");
	}

	// execution of SHARC
	static void SHARC(int target) throws IOException
	{
		int regionCount = vertices.get(n-1).getRegion();
		int index = 1, index2 = 0, i = 0;
		int[] boundary = new int[regionCount-1];
```

```
int[] boundNodes = new int[regionCount-1];

Iterator<Node> node = vertices.iterator();
while(node.hasNext()){
		Node vertex = node.next();
		if (index == n) break;
		if (vertex.getRegion() != vertices.get(index).getRegion()){

			vertices.get(index).setBoundary(true);
			boundNodes[i] = vertices.get(index).data;
			i++;
			boundary[index2] = vertices.get(index).getRegion();
			index2++;
		}index++;
}

Iterator<Edge> edge = edges.iterator();
while(edge.hasNext()){
		Edge arc = edge.next();
		for(int k=0; k<regionCount-1; k++){
			if((arc.getSource().data != boundary[k]) ||
					(arc.getDestination().data != boundary[k])){
				if(arc.getSource().getRegion() ==
					arc.getDestination().getRegion()){
					arc.setArcflag(true);


				}

			}
		}
}
node = vertices.iterator();
int k = 1;
int SharcTarget = 0;
while(node.hasNext()){
		Node vertex = node.next();
		if(vertex.getBoundary() == false){
			continue;
		}
		if (k<boundNodes.length)
			SharcTarget = boundNodes[k];
		else
			SharcTarget = target;
		k++;
			SharcExec(vertex, SharcTarget);    //execute dikstra for region boundary nodes

			edge = edges.iterator();
			while(edge.hasNext()){
				Edge arc2 = edge.next();
				if((arc2.getSource().state == State.UNLABELED) &&
						(arc2.getDestination().state ==
					State.UNLABELED)){
					if(vertex.getRegion() == arc2.getDestination().getRegion()
					&& vertex.getRegion() == arc2.getSource().getRegion()){

						arc2.setArcflag(true);

					}
				}
				else{
					arc2.setDeleted(true);
				}
			}
		}
		for(Node checknode : vertices){
			checknode.state = State.UNLABELED;
			checknode.children = null;
		}
}

// execute SHARC & Reach values on every partition
private static void SharcExec(Node start, int target) throws IOException {
		int f=0;
		for(Node checknode : vertices){
			checknode.key = -1.0;
		}
		heap.insertVertex(vertices.get(target));
		do
		{			f++;
		Node v = heap.deleteMin();
		v.state = State.SCANNED;

		if(v.getDeleted() == true)
			continue;

		for (int i = 0; i < v.incomingEdges.size(); i++) {
			Edge currentEdge = v.incomingEdges.get(i);
			Node headOfCurrentEdge = currentEdge.tail;
```

```
		if (headOfCurrentEdge.state != State.SCANNED){
			if (headOfCurrentEdge.state == State.UNLABELED) {
				// Insert a vertex with infinite key

				headOfCurrentEdge.state = State.LABELED;
				headOfCurrentEdge.pred = v;
				headOfCurrentEdge.key = v.key + currentEdge.length;
				heap.insertVertex(headOfCurrentEdge);
			} else if (headOfCurrentEdge.key > v.key + currentEdge.length) {
				// decrease the key of a vertex with finite key
				headOfCurrentEdge.pred = v;
				heap.decreaseKey(v.key +
			currentEdge.length,headOfCurrentEdge);
			}
		}
		else{
			currentEdge.setDeleted(true);
		}
}
} while (!heap.isEmpty());

Edge usedEdge = null;
Node temp = vertices.get(start.data);
while (temp != null) {
		if(temp.data == target || temp.pred == null) break;
		temp = temp.pred;
		double value = Math.min(start.key - temp.key,temp.key);
		if (value > temp.getReach()){
			temp.setReach(value);

		}
		else continue;
		temp = temp.pred;
		for (int i = 0; i < temp.incomingEdges.size(); i++) {

			Edge currentEdge = temp.incomingEdges.get(i);

			if (currentEdge.tail == temp.pred)
				usedEdge = currentEdge;
		}
		if(temp.pred == null || usedEdge == null)  continue;
		double value2 = Math.min(((start.key - temp.pred.key) +
		usedEdge.length),(temp.key));
		if((temp.getReach() < value2) && (temp.getBoundary() == false)
```

```
				&& (temp != vertices.get(target))){

			temp.setDeleted(true);
		}
	}

	for(Node checknode : vertices){
		checknode.state = State.UNLABELED;
		checknode.children = null;
	}
}

private static void innerIterator(Edge currentEdge, Node headOfCurrentEdge, Node v)
{
	if (headOfCurrentEdge.state == State.UNLABELED) {
		// Insert a vertex with infinite key
		headOfCurrentEdge.state = State.LABELED;
		headOfCurrentEdge.pred = v;
		headOfCurrentEdge.key = v.key + currentEdge.length;
		heap.insertVertex(headOfCurrentEdge);
	} else if (headOfCurrentEdge.key > v.key
			+ currentEdge.length) {
		// decrease the key of a vertex with finite key
		headOfCurrentEdge.pred = v;
		heap.decreaseKey(v.key + currentEdge.length, headOfCurrentEdge);
	}
}

private static void outerIterator() throws IOException
{
	// Delete minimum path
	Node v = heap.deleteMin();
	v.state = State.SCANNED;

	for (int i = 0; i < v.incomingEdges.size(); i++) {
		Edge currentEdge = v.incomingEdges.get(i);
		Node headOfCurrentEdge = currentEdge.tail;

		if(currentEdge.getArcflag() == false || currentEdge.getDeleted() == true){
			continue;
		}

		if (headOfCurrentEdge.state != State.SCANNED) {
			innerIterator(currentEdge, headOfCurrentEdge, v);
		}
```

```
		}
	}

	private static void exec(int start, int target) throws IOException {
		heap.insertVertex(vertices.get(target));
		do
		{
			outerIterator();
		} while (!heap.isEmpty());
		// Print out path
		Node temp = vertices.get(start);
		if (temp.pred == null) {
			System.out.print("There exist no s-t paths\n");
			return;
		}

		int vertexCount = 0;
		System.out.print("Shortest Path found:\n");
		System.out.printf("Distance: %f\n", vertices.get(start).key);

		while (temp != null) {
			System.out.printf("%d", temp.data);
			if(temp.data == target) break;
			temp = temp.pred;
			if (temp != null)
				System.out.print(" - ");
			vertexCount++;
		}
		System.out.printf("\nVertices passed: %d\n", vertexCount+1);
	}
}
```

****

FibonacciHeap.java

****

```
public class FibonacciHeap{

private Node[] rootListByRank = new Node[100];
private Node minRoot;

public FibonacciHeap() {
		minRoot = null;
}

private boolean link(Node root) {
		// Insert Vertex into root list
		if (rootListByRank[root.rank] == null) {
			rootListByRank[root.rank] = root;
			return false;
		} else {
			// Link the two roots
			Node linkVertex = rootListByRank[root.rank];
			rootListByRank[root.rank] = null;

			if (root.key < linkVertex.key || root == minRoot) {
				linkVertex.remove();
				root.addChild(linkVertex);
				if (rootListByRank[root.rank] != null)
					link(root);
				else
					rootListByRank[root.rank] = root;
			} else {
				root.remove();
				linkVertex.addChild(root);
				if (rootListByRank[linkVertex.rank] != null)
					link(linkVertex);
				else
					rootListByRank[linkVertex.rank] = linkVertex;
			}
			return true;
		}
}

public void dispose() {
		rootListByRank = null;
}

public final boolean isEmpty() {
		return (minRoot == null);
}

public final boolean insertVertex(Node node) {
		if (node == null)

			return false;
```

```
		if (minRoot == null)
			minRoot = node;
		else {
			minRoot.addSibling(node);
			if (minRoot.key > node.key)
				minRoot = node;
		}
		//minRoot.addChild(new Node());
		return true;
	}

	public final void decreaseKey(double delta, Node vertex) {
		vertex.key = delta;

		if (vertex.parent != null) // The vertex has a parent
		{
			// Remove vertex and add to root list
			vertex.remove();
			minRoot.addSibling(vertex);
		}
		// Check if key is smaller than the key of minRoot
		if (vertex.key < minRoot.key)
			minRoot = vertex;
	}

	public final Node findMin() {
		return minRoot;
	}

	public final Node deleteMin() throws IOException
	{
		if(minRoot.children == null){
			minRoot.addChild(new Node());
	}
		Node temp = minRoot.children.leftMostSibling();
		Node nextTemp = null;
		// Adding Children to root list
		while (temp != null) {
			nextTemp = temp.rightSibling; // Save next Sibling
			temp.remove();
			minRoot.addSibling(temp);
			temp = nextTemp;
		}
		// Select the left-most sibling of minRoot
		temp = minRoot.leftMostSibling();
		// Remove minRoot and set it to any sibling, if there exists one
		if (temp == minRoot) {
			if (minRoot.rightSibling != null)
				temp = minRoot.rightSibling;
			else {
				// Heap is obviously empty
				Node out = minRoot;
				minRoot.remove();
				minRoot = null;
				return out;
			}
		}
		Node out = minRoot;
		minRoot.remove();
		minRoot = temp;

		// Initialize list of roots
		for (int i = 0; i < 100; i++)
			rootListByRank[i] = null;

		while (temp != null) {
			// Check if key of current vertex is smaller than the key of minRoot
			if (temp.key < minRoot.key) {
				minRoot = temp;
			}
			nextTemp = temp.rightSibling;
			link(temp);
			temp = nextTemp;
		}
		return out;
	}
}
```