3.2 Important Functions

The programming language used (and referenced here) is C# on the Xamarin platform for Android development.

3.2.1 Dijkstra’s Algorithm

Line 120, SmartphoneMapper/SmartphoneMapper/NodeClass.cs

The creation of shortest paths between rooms (nodes) in the smartphone application is dependent on **Dijkstra’s algorithm** (essentially graph traversal by choosing nodes to proceed with from a min-heap), a description of which is given below.

First, an array with Boolean values is formed (with length equal to quantity of nodes in the graph), with all values initially set to *false*, describing whether a particular node has been visited (as with the connectivity check described earlier).

Next, a similar array is formed (with same length) but with numerical (e.g. float) values instead, with all values initially set to infinity (here referred to as the shortest path array). This represents the current shortest path to each respective node, achieved by traversing the graph greedily (with respect to edge weights) and accumulating weights as traversal progresses.

The origin node in question is then ‘chosen’ to commence the traversal; this is done so by setting its shortest path in the array as 0 (see below).

The following loop is then repeated:

* From the shortest path array, select the node with the lowest shortest path that has not yet been traversed. Mark this node as visited in the Boolean array.
* Consider all of its connected nodes that have not yet been traversed (i.e. are *false* in the Boolean array). For each one, if the shortest path of the current node plus the weight of the edge between the current and connected node (i.e. the cost of travelling to the connected node through the current node) is less than the current shortest path of the connected node (from the array), then it replaces the shortest path in the array. This is because, by performing this comparison, it has been determined that passing through the current node to reach the connected node is the more cost-effective way of arriving at it.

This loop is repeated until the current node considered is the destination node; at this point, the cost of the shortest path has been determined (the shortest path of the destination node in the array), but not the sequence of nodes in the path itself. To determine this, the path is analysed starting from the destination; the predecessor of each node (defined during the shortest-path algorithm) is appended to the beginning of a list (using head-end recursion), followed by the predecessor of that node, followed by the predecessor of that node, ad infinitum until the beginning is reached, and a list of nodes is formed.

This is summarised in the following pseudocode, accepting an adjacency list, the origin node, and the destination node as inputs, and returning a list of nodes involved in the path:

Syntax: ‘*x* appendedto *y*’ means list *y* with element *x* appended at the end.

shortestPath(List<float> Input, int Origin, int Destination)

Initialise BoolArray (with all elements initially assigned value of *false*);

Initialise ShortestPathArray (with all element initially assigned value of infinity);

ShortestPathArray[Origin] = 0;

Initialise int[] PredecessorArray;

Initialise currentNode;

While (currentNode != Destination)

currentNode = node with lowest value in ShortestPathArray and false in BoolArray\*

foreach(node in Input[currentNode]) //i.e. for each connected node

if(BoolArray[node] == false)

Let newCost = ShortestPathArray[currentNode] + node.EdgeWeight;

if(newCost < ShortestPathArray[node])

ShortestPathArray[node] = newCost;

Predecessor[node] = currentNode;

Endif

Endif

Endfor

Endwhile

Return Destination appended to backTrace(Input, Origin, Destination, ShortestPathArray);

Int[] backTrace(List<int> Route, int Origin, int Destination, int[] PredecessorArray);

if(Destination == Origin)

Return singleton list containing Origin;

else

int predecessor = PredecessorArray[Destination];

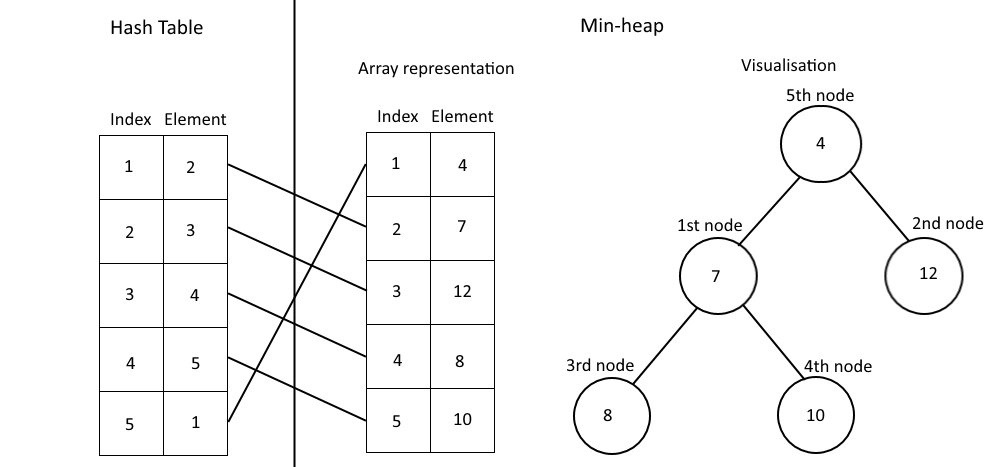
Route = backTrace(Route, Origin, predecessor, PredecessorArray);

Route.Add(predecessor);

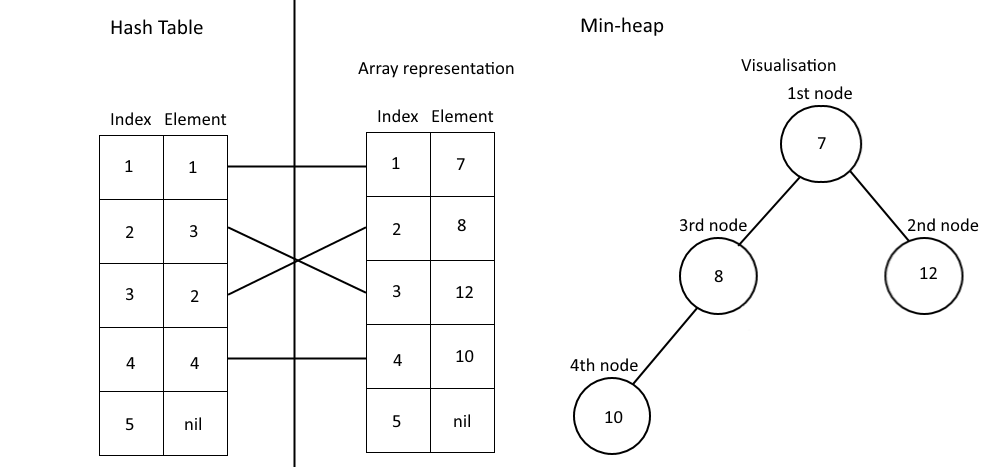
Endif;

//when first called, Route is null.

One element missing from this summary is the selection of the (unvisited) node with the lowest shortest path, marked with an asterisk in the pseudocode. This greatly affects the order of time-complexity of the algorithm as a whole: If this is achieved by manually scanning through every element of the **unsorted array** and ‘**bubbling out’** the smallest element that is yet unselected, the time taken is O(*V*), where *V* is the number of vertices (nodes) in the graph. A better alternative is to utilise a **binary min-heap**; the minimum element can be accessed in O(*1*) time, and can be popped immediately after (the node will never be examined again) in O(*log(V)*), leaving the next minimum available to be accessed. Importantly, the data structure must still allow values at particular indices to be updated (updating the shortest path to nodes); while a min-heap would not support this, it can be supported by using a **hash-table,** where indices of the hash table correspond to nodes, and the (integer) elements of the hash table correspond to the position of the nodes in the min-heap (see diagram below). Since every operation in the min-heap takes O(*k*) time (where *k* is a constant), and accessing the hash table is O(*1*), the asymptotic running time of using a min-heap is unaffected.



Demonstration of the hash table combined with the min-heap. Above is min-heap before the minimum element is popped, and below is after.



Using these, we can compare the asymptotic running times of an unsorted array with that of the min-heap and hash table. Since, at worst, all edges and all vertices will be traversed, the time-complexity of finding the cost of the shortest path is **O()**, where ***E*** represents the number of edges, represents the time-complexity of updating a node’s shortest path cost and represents the time-complexity of popping the minimum shortest path cost from the data structure.

In the case of the unsorted array, changing a node’s shortest path can be done by direct access, and so is of O(*1*). Removing the minimum element by scanning the array and ‘bubbling’ it out runs in O(*V*). Therefore the total complexity is O(*E* + *V*2) = O(*V*2+ *V*2)[[1]](#footnote-1) = O(2*V*2) = **O(*V*2)**.

In the case of the min-heap, changing a node’s shortest path has two steps; directly altering a node in the heap’s value (O(*1*), since hash table access is O(*1*) and heap access is O(*1*) due to being stored as an array) and subsequently restructuring the heap to satisfy the min-heap property (O*(log(V)*), coming out overall to O(*log(V)*). Popping the minimum element also has complexity of O(*log(V)*). Therefore total time-complexity is O(*Elog(V)* +*Vlog(V)*) = **O((*V* + *E*)*log(V)*)**. Clearly this is the better implementation, especially since the number of edges is far less than the number of vertices squared due to the sparsity of the graph, as explained before.

3.2.2 Search Function

Line 65, SmartphoneMapper/SmartphoneMapper/NodeClass.cs

Another element of the smartphone application is node searching using strings as search criteria. The aim is to return a list of nodes that contain the search criteria supplied in either name, description, or associations, where this list is ordered to prefer matches in name, followed by description, followed by association. This can be achieved in the following manner:

First, a Boolean array is established with length equal to number of nodes, with all elements initially assigned a value of *false* (signifying whether a node is already on the list). Next, an integer list is created, where values contained will represent the nodes. Now, iterate over the nodes and if the search term is found to be a substring of the iterated node’s **name**, add the node to the integer list and mark the node as added (*true*) in the Boolean array. Repeat this process again for node **descriptions**, followed by for node **assocations**. This will produce a list of nodes with matches in name first, followed by description, followed by assocations.

3.2.3 User-Manipulated View

SmartphoneMapper/SmartphoneMapper/GestureRecognizerView.cs

The logic of the smartphone application’s user-manipulated view is identical to that of the desktop application, with the exception of screen touch position substituting mouseclick position and screen pinch touch gesture substituting mousewheel scrolling.

1. *E* *V2* = O(*V*2); this can be demonstrated by considering an adjacency matrix, as all possible edges can be stored in the *V* x *V* cells of the matrix. [↑](#footnote-ref-1)