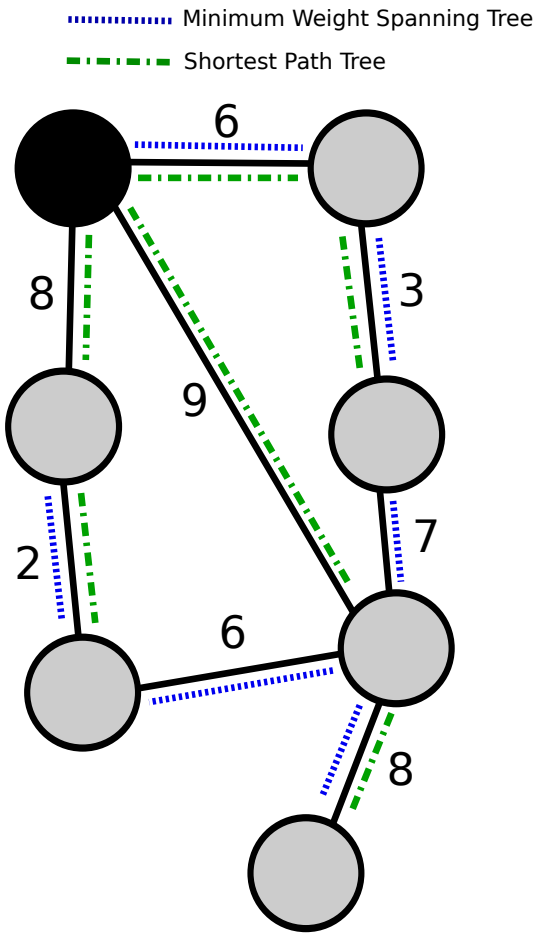
Assignment 3 - Theory

Andrew Hobden (V00788452)

October 30, 2014

1 Minimum Spanning vs Shortest Path Tree

Dijkstra's Single Source Shortest Path problem produces a Shortest Path Tree. Compare the Shortest Path Tree to the Minimum Weight Spanning Tree on the graph below.



The resulting Miniumum Weight Spanning Tree has $weight = 32$, while the resulting Shortest Path Tree has $weight = 36$. Therefore the Shortest Path Tree produced by Dijkstra's Since Source Path problem is not necessarily the Minimum Weight Spanning Tree.

2 Dijksta Correctness

We'll use the following assumptions:

- We know that Dijkstra's works on Undirected Graphs. *(Ulrike explicitly said this was allowed)*
- Any directed graph D_a can be seen as a subset of the undirected graph U_a. This is because an undirected graph can be transformed into a directed graph with each undirected edge e_a being replaced by pair of oppositely pointing directed edges $\{e_b, e_c\}$.
- Dijkstra's Algorithm always pulls in the lowest cost edge into it's set that doesn't connect to an already scanned node. It cannot select edges from outside of the set travelling into the set, only outgoing edges.

We'll make the following observations based on this:

- Optimal Substructure: A subpath of any shortest path is itself a shortest path.
- Triangle Inequality: If $\delta(u, v)$ is the shortest path from u to v, then $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$ for some x in the graph.
- No Cycles in Shortest Path: There are no cycles in shortest path$p$, otherwise two edges may be removed from p and it will result in a path with the same source and destination, but a lower cost.
- No Backtracking: Since there are no cycles, p contains only edges $E_P = e_0, ..., e_{i-1}, e_i \in p$ travelling in one 'direction'. That is, there be cannot a part of P_D which requires the algorithm backtrack, since this would result in a cycle between $e_{\{j,k\}}$ and $e_{\{g,h\}}$.

Theorem: (For a directed graph) If Dijksta's algorithm draws a vertex into it's set, that the shortest path to that vertex has been found.

Proof: Contradiction. Let$\delta(a, k)$ be the path such that such that$\delta(a, b) + \delta(b, k) \leq \delta(a, c) + \delta(c, k)$, where the vertices in$\delta(a, b)$ and k are already in the set, but some c is not. This means that that $\delta(a, b)$ must be of less weight than $\delta(a, c)$, however Dijkstra's algorithm would have selected$\delta(b, k)$ to pull k into the set instead of$\delta(c, k)$ only if the total distance to k would have been less for $\delta(a, k)$.

Theorem: Dijkstra's algorithm will not use an undirected edge more than once.

Proof: Any shortest path cannot contain cycles, this also means it cannot backtrack. If there was a shortest path with a cycle or backtracking, it could not be a shortest path, since it could be simplified.

It follows from the above theorems, as well as the obervations, that Dijkstra's Algorithm will perform correctly on Directed Graphs so long as it is only permitted to select edges originating from it's set travelling outward.

3 Shortest Paths to a Vertex Pseudocode

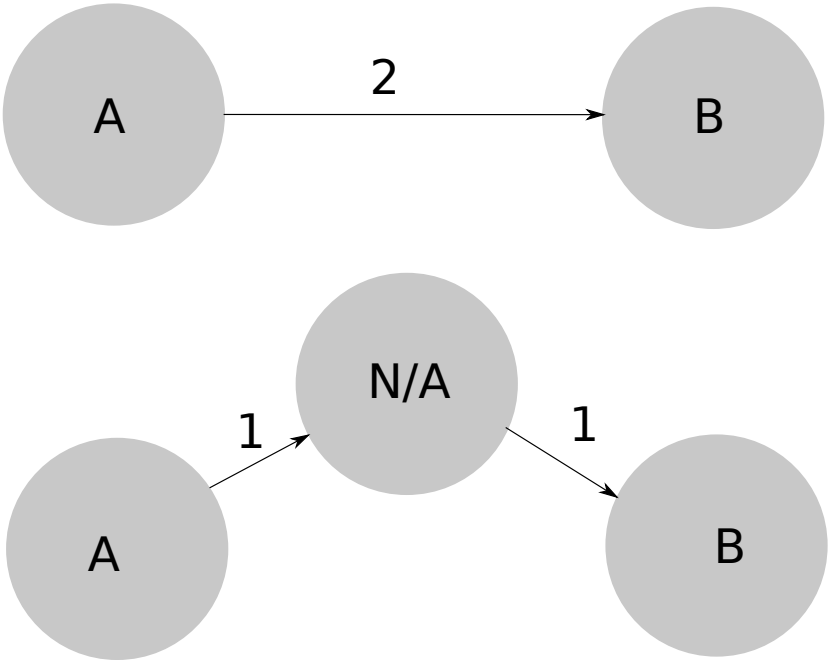
This "Single Destination Shortest Path" problem can be solved by simply performing Dijkstra's "Single Source Shortest Path" problem and reversing the determined paths.

```
1  fn SingleDestinationShortestPath(G: Graph, source: Node):
2    let V = G.vertices
3    let E = G.edges
4    let D: [int]  // Distance from source
5    let P: [Node] // Predecessors of i
6    let Q: PriorityQueue<int>
7
8    // Reverse all the paths in the graph.
9    for each e in E:
10     swap(e.destination, e.source)
11   end for
12
13   // Do Dijkstras
14   for each v in V:
15     if v == source:
16       dist[v] = 0
17     else
18       dist[v] = infinity
19     end if
20     Q.add_with_priority(v, dist[v])
21   end for
22
23   while Q is not empty:
24     let u = Q.extract_min()
25     // Pull into cloud
26     u.scanned = true
27     // Search neighbors
28     for each v in u.neighbors:
29       // Not in cloud
30       if v.scanned == false:
31         let new = D[u] + length(u,v)
32         if new < D[v]:
33           D[v] = new
34           P[v] = u
35           Q.decrease_priority(v, new)
36         end if
37       end if
38     end for
39   end while
40
```

```
41    // P is now the Single-Source Shortest Path
42    // We already swapped the edges, so swap back
43
44    // Inverse P
45    let S: [Node] // Successors of i
46    for index i in P:
47      S[P[i]] = i // Inverse direction
48
49    return S
50  end fn
```

4 Shortest Path Algorithm

The shortest path between vertices A, B in a direct graph $G = \{V, E\}$ with only edges of weight 1 can be determined in $O(V + E)$ time using a Breadth First Search. With a graph containing only edges of weights 1 and 2, it's possible to maintain this behaivor by having "imaginary" vertices with two weight 1 edges connecting in place of the weight 2 edge. Since the number of edges of weight two will always be less than E, the algorithm maintains a $O(V + E)$ time.



The following pseudocode demonstrates this application. Note how in the traceback the imaginary nodes are not added to any paths outputted.

```
1  fn oneOrTwoWeightSearch(G: Graph, s: Vertex, d: Vertex):
```

```
2     let Q = [s] // Queue of paths
3     while Q is not empty:
4       let item = Q.dequeue() // Next to search
5       item.visited = true
6       if item = d: // What we're looking for
7         return trace(item)
8       end if
9       for edge in item.edges:
10        if edge.destination.visited = false:
11          if edge.weight = 2:
12            let imaginary = new Vertex
13            imaginary.imaginary = true
14            // Edge from item to imaginary
15            imaginary.edges.push(item->imaginary)
16            // Edge from imaginary to the end of the edge
17            imaginary.edges.push(imaginary->edge.
                  destination)
18            imaginary.parent = item
19            Q.enqueue(imaginary)
20          else: // Weight 1
21            edge.destination.parent = item
22            Q.enqueue(edge.destination)
23          end if
24        end if
25      end for
26    end while
27    // d is not in G
28    return None
29  end fn
30
31  fn trace(item: Vertex):
32    if item.parent = null: // If it's the source
33      return [item]
34    else if item.imaginary = true:
35      return trace(item.parent) // Don't add.
36    else:
37      return trace(item.parent).enqueue(item)
38    end if
39  end fn
```

5 Application Areas

1. Logistics and shipping. Dijksta's algorithm is can be utilized by logistics companies to plan packages routes and optimize shipping. For example,

if a graph where each vertex is a depot, and edges are weighted according to the distance and/or cost to move items to that depot from others, Dijkstra's can be used to find the shortest or cheapest path to move the item along.

2. Geographic Information System. For an application like Google Maps, which produces directions between points A and B, Dijkstra's can be used to determine the optimal path along roads or other methods of transit like ferries, buses, or freeways. In this case, edge weights do not necessarily represent the distance travelled, but might be weighted based on a number of factors, including traffic and weather warnings.

3. Network Routing. In order to determine the optimal route for packets within a communications network Dijkstra's algorithm can be used, in this case edge weights might be determined by the link rate between routers.