COMP2007 - Assignment 1

Matthew Watson
SID: 440267858

1 Decide for each pair (u,v) in Q if there is a path between u and v in G

(a) **Description of how your algorithm works**
First, we read from stdin to a list of edges. Then for each edge we read nodes u and v in the pair to a dictionary of lists containing an adjacency list to represent the provided graph. We then call on **check_path**() providing the adjacency matrix and the start and end edges. This then is parsed to **bfs**() to perform breadth first search to find a possible path. Breadth first search works by first visiting all of a node's children before then visiting those children's children. It is used in this program to create a BFS path from the given start node and then for each step in the path, it checks if it reaches the desired end node and thus found a path, otherwise it breaks and determines there is no path. For each query, the program returns 1 if a path is found or 0 if there is no path found.

(b) **Argue why your algorithm is correct**
As this algorithm uses Breadth First Search (BFS) it can be assumed that it will find all paths given a start node and list of edges given common knowledge. The extension of BFS in this program is simply to check that the end node is contained in the BFS path returned from performing BFS from the start node.

(c) **Prove an upper bound on the time complexity of your algorithm**
As this implementation of BFS uses an adjacency list representation, for each vertex, u,v, we simply traverse the nodes which are adjacent to it in *O(m+n)* time. As we run BFS for each path query q, this will require *O(q)* time. Thus, we find that the total upper bound is:

$$O(q) \cdot O(m+n)$$
$$= O(q \cdot (m+n))$$

2 Find the cheapest way to connect all the vertices into one connected network

(a) **Description of how your algorithm works**
First we read in the data, storing a dictionary of the vertices and edges (with associated weights). When reading in, we do a check to see if the edge is in the list A of existing fibre links and if so, make its weight negative. We then use Kruskal's algorithm to create a minimum spanning tree from the list of vertices. To apply Kruskal's algorithm to forcefully use existing links, the algorithm sorts the edge weights with the smallest at the start. We observe that since the outputted MST starts with the smallest edge weights, we make any mandatory edges negative which forces them to all be added before the positive edge weights of potential fibre links are. After sorting, Kruskal's algorithm checks for each edge if it will form a cycle by calling on the union-find abstract data type and only if it doesn't form a cycle does it add it to the MST. For each edge it adds, it also adds the absolute weight of that edge to *totalweight* to keep track of the total weight of the MST that is returned at the end to 2 s.f.

(b) **Argue why your algorithm is correct**
Claim: After running this modified Kruskal's algorithm on a connected, positively weighted graph G with subset edges, A, that are existing and mandatory edges, its output is *totalweight*, the total weight of the minimum spanning tree, T, that contains A and the cheapest collection of edges to form G.
Proof: First, T is a spanning tree which has already been proved by common knowledge.
Second, T is a spanning tree of minimum weight using subset edges A. Again, this has been proved by common knowledge, however in this adaptation which must include subset edges, A:

- Mandatory edges, A, have negative weights and thus are cheaper than all possible edges, G.
- Edge weights of G are all positive as per the specification of given inputs: *"a positive (not necessarily unique) edge cost c_e for each edge in E"*
- Given Kruskal's algorithm will add edges to the MST starting with those of the lowest weight, all negative edge weights which in this case can only be the mandatory edges, A, will be added to the MST first and then will complete the MST with the cheapest remaining edges as per Kruskal's algorithm.

(c) **Prove an upper bound on the time complexity of your algorithm**
This algorithm uses Kruskal's algorithm, with the only additional step of checking if the edges being read in are of subset A which takes at most *O(m)* time. Otherwise,

the addition and/or rounding of *totalweight* for each edge added to the MST takes *O(1)* time however this is superseded by the overall run time of that loop as per the following. Looking at Kruskal's algorithm, sorting edges takes *O(m log m)* time. After sorting, it then iterates through all edges and applies the aforementioned union-find algorithm. Within that, the find and union operations can take at most *O(log n)* time. Thus, overall complexity is:

$$O(m) + O(m\, log\, m) + O(log\, n)$$

$$= O(m\, log\, n)$$