COMP2007 Assignment 1 Report

Jiashu Wu 460108049 jiwu0083

Question 1

1.1 Description of how the algorithm works (In plain English)

My algorithm performs a Breadth First Search (BFS) for each query. Since there are q queries, the algorithm will run BFS q times.

For each query, use the first node as the starting vertex of BFS, and then add all the unvisited neighbours to form a new layer, we call this an iteration. In each iteration, we check whether the destination node (which is the second node in that query) is included in this layer, if yes, the algorithm returns true and stop iterating since we have already find a path between these two nodes, otherwise we continue iterating. If all the iterations have finished but the destination node is still not included in all layers, we draw the conclusion that there is no such a path which connects these two nodes, and the algorithm return false in this scenario.

1.2 Prove the correctness of the algorithm

I will use Prove by Induction to prove the correctness.

- Inductive Claim:
  At iteration k, the layer produced by BFS contains all the vertices which is connected with the starting vertex S and have distance k to S.

- Base Case
  At iteration 0, there is only one node in this layer, which is the starting vertex S itself. Apparently, it is connected with itself and have distance 0 to itself.

- Inductive Hypothesis
  We assume that the inductive claim holds for k = L - 1.

- Inductive Step
  For k = L:
  At L^{th} iteration, we use all the unvisited neighbours of all vertices in $L\text{-}1^{th}$ layer to form a new layer. Since the inductive claim holds at $L\text{-}1^{th}$ iteration, the $L\text{-}1^{th}$ layer contains all the vertices which is connected with the starting vertex S with distance L-1, and also it is obvious that for every node n its neighbour is connected with itself, therefore we can say that the nodes in the L^{th} layer are all connected with the starting vertex S. Also since the distance between node n and its neighbour is 1, so all the vertices in L^{th} layer have distance (L - 1) + 1 = L from the starting vertex S.

- Corollary to Inductive Claim
  We can draw the conclusion that at iteration k, the layer produced by BFS contains all the vertices which is connected with the starting vertex S and have distance k to S.

Therefore, we can use BFS to decide whether there is a path between node u and v. Therefore, the algorithm is correct.

1.3 Prove an upper bound of the time complexity of the algorithm

1.3.1 Upper bound of the time complexity

The upper bound of the time complexity is O (q*(m + n)), where n stands for the number of vertices in graph G, m represents the number of edges in graph G, and q is the number of queries.

1.3.2 Prove the correctness of the time complexity

Loading the data requires O(m + n + q) work. Since loading both the number of vertices and number of edges takes O(1) + O(1), loading all the edges and build an adjacency list takes O(m + n) since we have n vertices and m edges in total, and loading all the queries takes O(q) times since we have q queries. Therefore loading the data takes O(1) + O(1) + O(q) + O(m + n) = O(m + n + q) work.

For each BFS operation, firstly marking all vertices except s as unvisited takes O(n) work since we have n vertices in total. And then we need to traverse all the vertices in graph G, since there are n vertices, thus this requires O(n) work. For each vertex, we also need to consider all the incident edges, for each vertex there are deg(u) incident edges. Each edge (u, v) will be counted exactly twice in sum (in deg(u) and deg(v)). And also since we are using adjacency list representation of the graph, processing each edge only costs O(1) work, therefore the total time processing edges is 2m. Thus for each BFS it requires O(n) + O(2m) which is O(n + m) work.

And in this question we have q queries in total, we need to perform BFS q times, therefore all the BFS takes q*O(n + m) = O(q * (n + m)).

Thus the whole algorithm takes O(q * (n + m)) + O(n + m + q) = O(q * (n + m))

Question 2

2.1 Description of how the algorithm works (In plain English)

Firstly, my algorithm sums all the original weights of edges in set A and save it into a variable, then change the weight of all the edges in A into a small number (this number is smaller than all the original edge weights of all edges). This is to make sure that later when we perform the Prim's algorithm to find the minimum spanning tree (MST), all these edges in set A will be guaranteed to be chosen.

After finishing all these preparation steps, my algorithm randomly chooses a vertex as starting vertex, to perform the Prim's Algorithm. For Prim's Algorithm, firstly we initialize an array, which stores the distances between the starting vertex and each vertex. Initially we set all the distance to infinity except for the starting vertex, which is 0. Then we initialize a priority queue (implemented using binary heap so that maximum height of the heap will be log(n)) and insert all the vertices into the priority queue by using distance as priority and vertex number as content, note that initially the distance for all vertices are set to infinity,

except for the starting vertex, which is set to 0. And then we initialise a set to contain all explored vertices. And then, we keeping popping the vertex with the minimum priority from the priority queue, add it into explored set, and traverse all its incident edges, if we find an edge with weight less than the priority, change its priority to this new cost and update distance array, otherwise we keep popping from the priority queue until empty. Finally, we add all the costs in the distance array (ignoring these "small numbers") and then add the original weights sum of all edges in set A (which is stored in a variable at the beginning), and return it as the minimum cost of the MST.

2.2 Prove the correctness of the algorithm

In this section, I will prove the correctness of my modified Prim's Algorithm.

Proof: Since the only thing that Prim's Algorithm does is keep using the Cut Property to find the minimum spanning tree (MST). Thus, if we can prove the correctness of the Cut Property, the Prim's Algorithm will be automatically correct. Thus, I will prove the Cut Property below using Exchange Argument.

Cut Property: Let S be any subset of vertices, and let e be the minimum cost edge with exactly one endpoint in S. Then the MST T* contains e.

Proof the correctness of Cut Property (By Exchange Argument):

Let's suppose that the minimum cost edge e does not belong to MST T*. Thus, adding e to T* will creates a cycle C in T*. Therefore, edge e is both in the cycle C and in the cutset D corresponding to S. We can know that there exists another edge, say f, that is also in both C and D. We denote T' = T* U {e} – {f}, which is also a spanning tree. (T' contains all edges of T* except edge f, and T' also contains edge e). Since cost(e) < cost(f), thus cost(T') < cost(T*), which contradicts with T* is MST. Thus, minimum cost edge e must belong to MST T*. Thus, the Cut Property holds. Thus, the Prim's Algorithm holds.

Since we have adjusted the weight of all the edges in A into a number which is smaller than the original weights of all edges, and the Cut Property holds, therefore we can guarantee that these edges can be chosen during the Prim's Algorithm since the edge with minimum cost must belong to the MST (Cut Property). Therefore, our modified Prim's Algorithm is correct, which means our algorithm can produce the correct minimum spanning tree which uses all the edges in A.

2.3 Prove an upper bound of the time complexity of the algorithm

2.3.1 Upper bound of the time complexity

The upper bound of the time complexity of this algorithm is O(m*log(n)), where n represents the number of vertices in graph G, and m is number of edges in graph G.

2.3.2 Prove the correctness of the time complexity

Initially we need to load the number of vertices, number of edges, which takes O(1) and O(1) respectively. Then we need to load all the edges and their weight, meanwhile build an adjacency list of the graph, which takes O(m + n) work since we have n vertices and m edges in this graph. After this, we need to load the size of set A and all edges in set A. Since there can have at most m edges in set A, thus it requires O(1) + O(m) work. Therefore, the whole loading process takes O(1) + O(1) + O(m + n) + O(1) + O(m) = O(m + n) work.

For the Prim's Algorithm part, first we need to create an array d, which stores all vertices and a weight, since we have n vertices in total, this will take O(n) work. Then we need to push all the vertices and their weight into the priority queue. Since the priority queue is implemented with binary heap and the height of the binary heap remains to be log(n), thus each push operation takes O(log(n)) time, therefore pushing n vertices costs O(n * log(n)) work.

After all these pushing operations, there should have n entries in the priority queue, therefore we need to pop n times, and each pop operations takes O(log(n)) time as well, since the height of the priority queue is log(n). Thus all the popping operations requires n * O(log(n)) = O(n * log(n)) work.

After popping of each vertex, we need to traverse all its incident edges, and for each vertex u it has deg(u) incident edges. Since we are using adjacency list representation of the graph, each visit takes O(1) work and the sum of all these deg(u) is 2*m. (Since each edge will be visited twice, one in deg(u) and another in deg(v)). If a cheaper cost has been found we need to decrease the priority, which also costs O(log(n)), since the height of the heap is log(n). Since we can do decrease key operation at most m times, therefore all the decrease key operations require O(m * log(n)) work.

Thus the whole algorithm requires O(m + n) + O(n * log(n)) + O(n * log(n)) + O(m * log(n)) = O((m + 2*n) * (log(n)) + O(m + n) = O((m + n) * log(n)) = O(m * log(n)) (Since m >= n)