Question 1 (12 marks)

Written by: Shoaib
Verified by: Mark, Tanuj

Prove that every connected undirected graph G contains at least one vertex whose removal (along with all incident edges) does *not* disconnect the graph. Give an algorithm that for any connected graph $G = (V, E)$, given by its adjacency list, finds such a vertex in $O(|V| + |E|)$ time in the worst case. Justify your algorithms correctness and worst-case time complexity.

Proof. Removal of one vertex where $|V| = 1$ does not disconnect the graph trivially. If $|V| > 1$, we consider the breadth-first tree(BFT) G_T derived from running breadth-first search (BFS) on an arbitrary starting vertex s. Since G is connected, by correctness of BFS, G_T holds all $v \in G.V$ and simple paths from s to v for all $v \in G.V$. We claim that removal of a leaf vertex $u \in G_T$ does not disconnect G. Indeed, removal of u and the edge $(u.parent, u)$ leaves other edges in the tree intact. Consider G with u removed, since there is always a path from s to v for all $v \in G.V$. There exists a path $\langle v_0, v_1, \cdots, s, \cdots, u_1, u_0 \rangle$ for any $v_0, u_0 \in G.V$. Therefore G is still connected. Since $|V| > 1$, G_T has at least one leaf, so there is always a vertex whose removal does not disconnect the graph. □

The algorithm is as follows

1. Call BFS(G, s) for arbitrary starting vertex s. Assign G_T to breadth first tree returned from the search
2. Traverse down G_T from root to a any arbitrary leaf l. Then return l

The correctness of the algorithm follows from the previous proof, where we proved that removal of a leaf from G_T does not disconnect the graph
BFS(G, s) takes $O(|V| + |E|)$. the BFT G_T contains all $v \in G.V$ hence $|G_T| = |V|$. We know that worst case time complexity of an unbalanced tree traversal is $O(|G_T|) = O(|V|)$. Hence the algorithm has overall worst case time complexity of $O(|V| + |E|)$

Question 2 (24 marks)

Written by: Tanuj
Verified by: Shoaib, Mark

Let $G = (V, E)$ be an undirected graph. G is ***bipartite*** if the set of nodes V can be partitioned into two subsets V_0 and V_1 (i.e., $V_0 \cup V_1 = V$ and $V_0 \cap V_1 = \varnothing$), so that every edge in E connects a node in V_0 and a node in V_1. For example, the graph shown below is bipartite;

this can be seen by taking V_0 to be the nodes on the left and V_1 to be the nodes on the right.

Note that G is bipartite if and only if every connected component of G is bipartite.

a) (8 marks) Prove that if G is bipartite then it has no *simple cycle* of odd length. Hint: Give a proof by contradiction.

Solution. Proof by contradiction. Assume there exists a simple path of length k where k is odd in G. In other words, there exists a cycle $C = \langle v_0, \cdots, v_k \rangle$ such that $v_0 = v_k$, $v_1 \neq \cdots \neq v_k$ and k is odd. Since G is bipartite, we have $(v_i, v_{i+1}) \in G.V$ if and only if $v_i \in V_0$ $v_{i+1} \in V_1$ or $v_i \in V_1$ $v_{i+1} \in V_0$. Assume $v_0 \in V_0$, then the k-th vertex along the path $v_k \in V_1$. However, by $v_0 = v_k$ we have $v_0 = v_k \in V_0$, hence contradiction. Hence there is no simple cycle of odd length in a bipartite graph. □

b) (8 marks) Prove that if G has no simple cycle of odd length (i.e., every simple cycle of G has even length) then G is bipartite. (Hint: Suppose every simple cycle of G has even length. Perform a BFS starting at any node s. Assume for now that G is connected, so that the BFS reaches all nodes; you can remove this assumption later on. Use the distance of each node from s to partition the set of nodes into two sets, and prove that no edge connects two nodes placed in the same set.)

Proof. Assume G is connected. Pick arbitrary s and perform BFS(G, s). Now we partition v for all $v \in G.V$ depending on $v.d$. Specifically, we assign v to be in V_0 if $v.d$ is even and in V_1 if $v.d$ is odd. Note that any $u, v \in G.V$, the path $P = \langle u, \cdots, s, \cdots, v \rangle$ connects u to v and has a path length of $u.d + v.d$. If $u, v \in V_0$. $u.d + v.d$ is even. Or if $u, v \in V_1$, since $u.d$ and $v.d$ are odd, $u.d + v.d$ is still even. Proof by contrapositive, assume that G is not bipartite. Without loss of generality, assume $e = (u, v) \in G.E$. Then the simple cycle formed by P and e is given by $C = \langle u, \cdots, s, \cdots, v, u \rangle$ (BFS yields simple path). Note that C has path length of $u.d+v.d+1$, which must odd if $u, v \in V_0$ or $u, v \in V_1$. Hence there exists a simple path C of odd length. The contrapositive hence holds. Specifically, if G has no simple cycle of odd length, then G is bipartite. If G has no simple cycle of odd length for every connected components, then G is bipartite for every connected components, hence G is bipartite. □

c) (8 marks) Describe an algorithm that takes as input an undirected graph $G = (V, E)$ in adjacency list form. If G is bipartite, then the algorithm returns a pair of sets of nodes (V_0, V_1) so that $V_0 \cup V_1 = V$, $V_0 \cap V_1 = \varnothing$, and every edge in E connects a node in V_0 and a node in V_1; if G is not bipartite, then the algorithm returns the string not bipartite. Your algorithm should run in $O(n + m)$ time, where $n = |V|$ and $m = |E|$. Explain why your algorithm is correct and justify its running time. (Hint: Use parts (a) and (b).)

Solution. The algorithm is follows. We assume $G.Adj[v]$ to be the adjacency list containing vertices u such that $(v, u) \in G.E$. We also use a disjoint set ADT with union-by-rank and path compression.

(a) Call MAKE-SET(V_0) and MAKE-SET(V_1) where V_0 and V_1 are representative values for identifying the two sets.

(b) Initialize v for all $v \in G.V$ color to white and distance to infinity. Loop over v for all $v \in G.V$ and check for its color, call BFS(G, v) if v is white.

(c) In each BFS(G, v), in addition to the step described in textbook/lecture. During the step to explore each $v \in G.Adj[u]$, call UNION(FIND-SET(V_1), MAKE-SET(v)) if $v.d$ is odd and UNION(FIND-SET(V_0), MAKE-SET(v)) if $v.d$ is even.

(d) After BFS explored every vertex $v \in G.V$, we check if G is bipartite. Loop over $G.Adj[v]$ for all $v \in G.V$. In each adjacency list, compare FIND-SET(v) and FIND-SET(u) for all $u \in G.Adj[v]$. If FIND-SET(v) == FIND-SET(u), then return string *not bipartite*

(e) Return a pair of sets (FIND-SET(V_0), FIND-SET(V_1)) if the previous loop did not return already.

By correctness of BFS, every $v \in G.E$ is enqueued once. Hence every $v \in G.E$ is either in the set V_0 or V_1, where we decide which set to union based on $v.d$. We proved previously that this particular way of partitioning $G.V$ gives rise to two disjoint sets which represents a bipartite graph if and only if there is no edge $e = (u, v) \in G.E$ such that u and v is within the same set. Therefore we evaluate this condition in step *d*). The loop over every $G.Adj[v]$ for all $v \in G.V$ checks all edge $e = (u, v)$ such that e does not connect vertices of the same set. If FIND-SET(v) == FIND-SET(u) is true, then this contradicts definition of a bipartite graph, hence G is not a bipartite graph. So the return value *not bipartite* is correct. Otherwise, If the loop exists without encountering return statement, part *e*) is executed, The step implies that there is no edge that connects to vertices of the same set. Hence G is a bipartite graph. The return value (FIND-SET(V_0), FIND-SET(V_1)) is correct.
The BFS has a worst time complexity of $O(n + m)$. The FIND-SET and UNION operation for inserting v into the two sets V_0 and V_1 is executed once until all $v \in G.V$ is inserted. Hence there is exactly n FIND-SET and UNION operations in during BFS. Since FIND-SET is executed once for every edge in the adjacency list, hence there are exactly m FIND-SET when checking if G is bipartite. Together with 2 MAKE-SET operation at the beginning, there are $2n + m + 2$ disjoint sets operations. Since disjoint sets of size k has worst case sequence complexity of $O(k\alpha(k)) = O(k)$, where $\alpha(k)$ is an extremely slow growing function which we will be treating as a constant. The disjoint sets operations in the algorithm has worst case time complexity of $O(2n + m + 2) = O(m + n)$. Together with worst case time complexity of $O(m + n)$ for BFS. The worst case time complexity of the algorithm is hence $O(m + n)$ □

Question 3 (24 marks)

Written by: Mark
Verified by: Shoaib, Tanuj

You are riding your bike and you have an aerial map of the routes you could take. This map contains the (x, y)-coordinates of n bike pump stations. There is a straight bikeway that goes directly between any two bike pump stations. You want to travel from station s to station t. Since your tires arent very good, the best route from s to t is one that minimizes the distance that you have to travel between any two successive stations on this route. Your task is to design an $O(n^2 logn)$-time algorithm for finding a best route.

a) Restate this task as a graph problem. To do so: (i) describe the graph: its vertices, edges, and edge weights, and (ii) restate the task as a graph problem on this graph.

b) We learned several algorithms that construct trees from an input graph. One these trees can be used to find the desired path efficiently. Explain how and prove it.

c) Describe a corresponding algorithm for solving this problem in plain and clear English.

d) Explain why the worst-case running time of your algorithm is $O(n^2 logn)$.

NOTE: your algorithm can use any algorithms that we studied in the course as black-boxes, and you can refer to their worst-case time complexity as stated in lecture or in the textbook.

Solution. The idea behind this problem will be to use a "limited" minimum spanning tree algorithm to create a rudimentary minimized-distance path algorithm.

a) *The graph setup.*

- *Vertices.* From the problem description, we have n bike pump stations. Thus, let each vertex be a bike pump station, giving us $|V| = n$ vertices, which each vertex corresponding to a bike pump station.
- *Edges.* From the problem description, we know that every bike pump station has a *straight path* to every *other* bike pump station. Thus, we let each edge (u, v) be assigned to represent one of these straight paths, where u and v are vertices representing bike pump stations. Then we have $nC2$ edges total since each vertex is connected to every other vertex, which means we have $|E| = \frac{1}{2}(n-1)n$ total edges, so $|E| = f(n)$ aka a function of $|V| = n$.
- *Edge weights.* The n bike pump stations, we can assume, are randomly scattered on the map, which uses (x, y) coordinates. Then each edge weight w is simply an integer represented by the distance formula $w = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ given the positional coordinates $u = (x_1, y_1)$ and $v = (x_2, y_2)$ of two bike pump stations u and

v connected by an edge (u, v). We then assign (u, v) this weight $w(u, v)$ which is thus a function of u and v's positions. The details of these weights won't need to be optimized as long as we assume arbitrary values of w, so that our algorithm will generalize.

- The graph will be in **adjacency list representation**.

The question, restated as a graph problem on the graph.

- We are given a graph G such that $|V| = n$ and every vertex is connected to every other vertex, so we have $|E| = \frac{1}{2}(n-1)n$ and each edge is assigned a weight w based on the Euclidean distance between two edge-connected vertices. Starting from an input vertex s, we want to find the best route to a destination vertex t such that w between every two successive stations on this route is minimized when compared to ther local distances, and our algorithm to do so must run in time $O(n^2 \log n)$.

This sets up our ability to look through a few different algorithms to use for tackling this problem.

b) Recall that a *minimum spanning tree* is a tree constructed out of a weighted, undirected graph where the weights w are, in total, minimized. Our algorithm wants a path such that every edge has minimum weight between two adjacent vertices on the path. The obvious choice for the basis of this algorithm is thus **Prim's algorithm**. With Prim's algorithm, we build a minimum spanning tree from a root node r, allowing us to reach individual nodes and add them to the tree in a connected fashion. Furthermore, each node will have a pointer to its *parent* (denoted $v.\pi$ for vertex v in CLRS), which is the node that Prim's algorithm was at before reaching this particular node. This gives a chain of connected nodes that will serve as the basis of the path.

Prim's algorithm works because it is a *greedy algorithm that works by scoping out adjacent, reachable vertices* (in contrast to Kruskal's algorithm, which works from independent edges and "stitches" them together using disjoint sets to form minimum spanning trees). Thus, from a root node, the algorithm will use the greedy heuristic to continually choose to follow the edge with the cheapest cost, adding the vertex reached from such an edge. Because this greedy heuristic is thus applied to every edge in Prim's, it follows that *any edge between two vertices in the minimum spanning tree would have minimum weight out of all other possible options.* This is precisely what our algorithm requires.

Finding the desired path. All we need to do is make the following modification to Prim: starting from an input node s, when Prim's algorithm reaches the destination node t, we stop building the minimum spanning tree and we follow the *parent* pointers to move back from t to s, collecting the edges. Because of Prim's greedy heuristic, each of these edges will have minimum possible weight between two adjacent nodes on the path, which is exactly what our algorithm requires, and then we can simply return the path as the collection of

edges. Thus, because of how Prim's algorithm works and its vertices-based greedy heuristic, it follows that we can find the desired path by following the parent-chain.

c) Drawing from the explanation in b), we can sketch out an outline of the algorithm below:

- **MINIMIZED-PATH(G,s,t)** where G is the adjacency list representation of the graph, s is the starting vertex, and t is the destination vertex. We assume implicitly that each edge has its weight w already assigned to it.
- Initialize an array of size n^2 called P which will contain the path edges.
- We define DESTINATION-PRIM as a variation on Prim's algorithm with the following subroutines and modifications:
  - The min-priority queue is a **binary min-heap**.
  - When observing a node and checking adjacent nodes along the edges (lines 9 - 11), if the added node v is in fact our destination node t, we can stop building the minimum spanning tree here. Then we would follow the nodes, starting with this t, backward by following the chain of parent pointers to get back to s, and each edge we traverse is added to P.
- Finally, we return P which represents the path from s to t where all adjacent distances have minimum weight.

Correctness follows from the correctness of Prim's algorithm constructing a minimum spanning tree, and then simply following the minimized weights in the returned path.

d) An outline of the algorithm MINIMIZED-PATH's run time:

- Initialization of P is constant time.
- From class/CLRS, Prim's algorithm takes worst case time $O(|E| \log |V|)$ when using a binary min-heap. Let $|V| = n$ and $|E| = \frac{1}{2}(n-1)n = \frac{1}{2}n^2 - \frac{1}{2}n$ as described in a). DESTINATION-PRIM is essentially identical to a normal Prim's algorithm in worst-case considerations, so the building of the minimum spanning tree to reach destination t is still $O(|E| \log |V|)$. However, we then use parent pointers to traverse and collect the edges from t to s, which is a subroutine dependent on the number of edges in the graph, thus this takes $O(|E|)$ and in total DESTINATION-PRIM takes $O(|E| \log |V| + |E|)$. The return takes constant time.
- Thus, the algorithm's total time complexity is $O(|E| \log |V| + |E|)$ which is just $O((\frac{1}{2}n^2 - \frac{1}{2}n) \log n + (\frac{1}{2}n^2 - \frac{1}{2}n))$. This is of order $O(n^2 \log n - n \log n + n^2 - n)$ which is of order $O(n^2 \log n)$.

$O(n^2 \log n)$ is precisely the time we need for this algorithm, as required. □