CSC263 Assignment 5

Zikun Chen, Yue Li, Yueen Ma

March 30, 2017

Question 1

(Written by Yue Li, Read by Yueen Ma, Zikun Chen)

(a) Prove: every connected undirected graph G contains at least one vertex whose removal (along with all incident edges) does not disconnect the graph.

- For any connected graph $G = (V, E)$, let egdes $E = \{e_0, e_1, ..., e_n\}$, let vertices $V = \{v_0, v_1, ..., v_m\}$
- Want to find such a vertex $v \in V$ that its removal (along with all incident edges) does not disconnect the graph.
- Find the spanning tree of graph G, denoted by $T = (V_T, E_T)$,
Note that T contains all vertices in graph G.(by the property of spanning tree), then $V_T = V = \{v_0, v_1, ..., v_m\}$, Let $E_T = \{e_0, e_1, ..., e_k\} \subseteq E$, where $k \leq n$(WLOG)

- By the property of spanning tree, T is a connected graph without any cycle, now remove one of the leaves in T, denoted by v_m(WLOG);
also remove the edge containing v_m in E_T, denoted by e_k(WLOG).
Let $T^{'} = (V_{T'}, E_{T'})$ be the spanning tree after this removal, $E_{T'} = E_T - \{e_k\}$
By the propery of spanning tree: $T^{'}$ is connected, but note that we removed more edges from the graph G than we have to, since some edges that are not connected with v_m are removed to find a spanning tree T.

- Let $E^{''} = E - E_T = \{e_{k+1}, e_{k+2}, ..., e_n\}$, which is the set of edges being removed to find a spanning tree.
Now we need to add $E^{''}$ back, except the edges contains v_m. This operation leaves the graph connected, since $E_{T'}$ is already sufficient to keep the rest of the graph connected, adding more edges cannot disconnect it.
This means that we have found such a vertex that its removal (along with all incident edges) does not disconnect the graph.

Therefore, every connected undirected graph G contains at least one vertex whose removal (along with all incident edges) does not disconnect the graph.

(b) **Pseudocode**

Note:
Colour White means node hasn't been discovered;
Colour Grey means node is discovered;

```
1  def Find_vertex(G, s)
2      total_vertex <- 0
3      for each vertex u in G.Adj - {s} // Each vertex in head of Adj list except s
4          u.colour <- White
5          total_vertex <- total_vertex + 1
6          u.parent <- NIL
7      total_vertex <- total_vertex + 1
8      count_vertex <- 1
9      s.colour <- Grey
10     Q <- Empty                        // Initialize an empty queue
11     ENQUEUE(Q, s)
12     While Q != Empty
13         u <- DEQUEUE(Q)
14         for each v in G.Adj[u]
15             if v.colour == White
16                 v.colour <- Grey
17                 v.parent <- u
18                 count_vertex <- count_vertex + 1
19                 if count_vertex == total_vertex - 1
20                     for each vertex u in G.Adj
21                         if u.colour == White
22                             return u
23                 ENQUEUE(v)
24         u.colour <- Black
```

Idea and Correctness Justification

Suppose a connected graph contains n vertices in total. In order to find a vertex in given graph such that its removal remains the original graph connected, we want to find a connected graph such that it does not include the vertex, which means that the total number of vertices in the result graph is $n - 1$.

By the given adjacency list, apply variation of Breadth-First search(Find_vertex). This gives us a way to connect the vertices one at a time.

Whenever a vertex v is discovered(colour of v is set to Grey), which means that it is connected to its parent vertex u indicated by edge(u, v) and added to the BFS Tree, and we increment the count_vertex by

one. So when count_vertex reaches $n-1$, we know that the number of nodes/vertices in the BFS Tree reaches $n-1$, which means that all the vertices are connected ***except the vertex that could be removed***, we call it removed vertex.

Since the edges in the BFS Tree can keep the rest of the graph connected and the edges do not contain the undiscovered vertex, the removal of the undiscovered vertex and its edges does not make the BFS Tree disconnected. Then the graph with the same first $n-1$ does not disconnect.

Since the removed vertex has not been discovered, which means that the colour of it should be white, then we use a linear search on the heads of the linked lists in the adjacency list of the graph to find it and return it.

Worst-case Time Complexity Analysis
- Count the total number of vertices and set colour takes $O(|V|)$ times

- In worst case, to connect $(n-1)$ vertices together, applying breadth-first search. Because the alogrithm scans the linked lists starts with each vertex only when the vertex is dequeued, it scans each linked list at most once, in the worst case. The count_vertex might reach $n-1$ when the last linked list is scanned, since the sum of the lengths of all the linked lists in adjacency lists is $\Theta(|E|)$, it takes at most $O(|E|)$ time
- To find the vertex to be removed, which means find the vertex that is white among all vertices, in worst case, which takes at most $O(|V|)$ time

Therefore the worst case time complexity is
$O(|V|+|E|+|V|) \in O(|V|+|E|)$

Question 2

(Written by Zikun Chen, Read by Yueen Ma, Yue Li)

(a) Suppose G is bipartite, then every connected component of G is bipartite

If every connected component has no cycle (acyclic), then there is no simple cycle. Done.

Proof by Contradiction: Suppose that there exist a simple cycle of odd length, called O.

Say the cycle is in one of G's connected component C. And C is bipartite, so let A and B be the two partitions of C.

Let the odd cycle be represented as a set of vertices $O = v1, v2, \cdots, vk, v1$ with k being an odd number. And without loss of generality, assume v_1 is in A.

Claim: $v_i \in A$ for i odd and $v_i \in B$ for i even.

Proof by induction:
Base Case:

$i = 1$: v_1 is in A.
$i = 2$: Since C is bipartite so the edge (v_1, V_2) must connect vertices from different partition.
Therefore, v_2 is in B

Inductive Hypothesis:
$v_n \in A$ for n odd and $v_n \in B$ for n even.

Show $v_{n+1} \in A$ for $n+1$ odd and $v_{n+1} \in B$ for $n+1$ even.

If n is even, then n+1 is odd. By IH, $v_n \in B$ and C is bipartite, so (v_n, v_{n+1}) connects two vertices from different parts of the partition. Therefore, $v_{n+1} \in A$

Similarly, if n is odd, then n+1 is even. By IH, $v_n \in A$ and C is bipartite, so (v_n, v_{n+1}) connects two vertices from different parts of the partition. Therefore, $v_{n+1} \in B$

So $v_k \in A$ since k is odd, and $v_1 \in A$ since 1 is odd. So the edge (v_k, v_1) has endpoints both in A. This contradicts the the fact that C is bipartite.

Therefore, the bipartite graph G has no odd cycles.

(b) Suppose the graph G has no simple cycle of odd length.

If G is acyclic, then we are done.

If G has cycles, first assume G is connected.

Then let us perform BFS on G starting from any vertex in G, say s. Then the algorithm gives a spanning tree T of the whole graph with a distance attribute for each vertex.

We group the vertices with odd distance together, call this group A, and group vertices with even distance together, call it B.

Claim: This is the bipartition for G.

Proof by Contradiction: Suppose there exist an edge e = (u, v) in G that has endpoints u and v in the same part of our partition. Without the loss of generality, assume u and v are both in A.

u and v both have either odd distance or even distance. Since they are both in the spanning tree T, there is a path P from u to v contained in T. The length of the path (number of edges) is the difference of the distance of u and distance of v, which is even. Because the difference of two odd numbers (or two even numbers) is even.

Then the cycle, $P \cup \{e\}$ has odd length, which contradicts the fact that G has no cycle of odd length.

Therefore, G is bipartite with the bipartition A, B.

If G is not connected with no cycle with odd length, we can similarly perform BFS on each connected component (which also has no cycle with odd length) C_i with partition A_i, B_i with odd and even distance respectively.

Then by the above proof by contradiction, each C_i is bipartite, so G is bipartite.

(c) **Pseudo Code:**

```
1  BIPARTITE((V, E)):
2      # initilization
3      for v in V:
4          color[v] = white
5          distance[v] = infinity
6      Q = empty queue
7      pick any vertex s in V to start with
8      color[s] = grey
9      distance[s] = 0
10     Q.enqueue(s)
11
12     # main loop
13     while Q not empty:
14         u = Q.dequeue()
15         for all edges (u,v) in E:
16             if color[v] == white:
17                 color[v] = grey
18                 distance[v] = distance[u] +1
19                 Q.enqueue(v)
20
21             # detecting cycles
22             elif color[v] == grey or black:
23                 difference = distance[v] - distance[u]
24                 # check if the cycle has odd length
25                 if (difference + 1) % 2 == 1:
26                     print "not bipartite."
27                     return
28         color[u] = black
29
30     # construct the bipartition
31     V0 = empty set
32     V1 = empty set
33     for v in V:
34         if distance[v] % 2 == 0:
35             add v to V0
36         if distance[v] % 2 == 1:
37             add v to V1
38     return (V0, V1)
```

Explanation: The algorithm is essentially a variation of BFS, where we added an additional if statement in the for loop on line 22 to detect cycles, in other words, to check if we reached a vertex that we already discovered

when we loop over all the edges coming out of the current vertex we are exploring. If we found a cycle, we first take the difference of the distances of the vertices, which gives the length of the path between the two nodes. And we add 1 to this difference, which is the length of the back edge that we found. This gives the length of the entire cycle. We then check if the cycle has odd length, in which case, we print "not bipartite" and terminate. Since by part a) a bipartite graph cannot have simple cycle of odd length. If the cycle is of even length, we continue the algorithm.

Each vertex has a color attribute, initially white, meaning not discovered. Grey means the vertex is discovered but not explored and black means the vertex has been fully explored. We also compute the distance attribute of each node to the starting vertex s in the process of visiting all the vertices.

At the end if the algorithm did not terminate, we know that we have visited all the notes in the graph and the graph is bipartite because it does not contain any odd cycles. So we construct the bipartition by grouping vertices with even distance and those with odd distance respectively similar to what we did in part b).

Runtime: The runtime of BFS when the input is an adjacency list is $O(n+m)$. The addition of the cycle detecting if statement on line 22 has constant runtime. The construction of bipartition at the end takes $O(n)$ time because we loop over the set of vertices. So the overall runtime of the algorithm is still $O(m+n)$

Question 3

(Written by Yueen Ma, Read by Zikun Chen, Yue Li)

a.

(i)

Given a completely connected undirected weighted graph $G = (V, E)$. V is a set of vertices, which represent bike pump stations on a map with coordinates (x, y). E is a set of edges representing the straight path between any two bike pump stations. The weights are the distances between any two vertices, i.e., bike pump stations.(e.g., distance between $v_1 = (x_1, y_1), v_2 = (x_2, y_2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$)

(ii)

Denote the weights of the edges as $w(e)$.
Task: Design an algorithm that takes $O(n^2 logn)$ time to find such a route:
Let such route be R from the starting point v_s to then ending point v_e, which consists of the edges $E_r = \{e_1, e_2, \ldots, e_k\}$.
For all other routes R' from v_s to v_e, which consists of edges $E'_r = \{e'_1, e'_2, \ldots, e'_t\}$, $max\{w(e'_1), w(e'_2), \ldots, w(e'_t)\} \geq max\{w(e_1), w(e_2), \ldots, w(e_k)\}$

b.

- **How to construct the tree and use the tree to find the best path/route:**
  Use Kruskal's algorithm on the graph.
  This algorithm returns an MST of the given graph.
  This MST contains a unique path between any two vertices in the graph since MST's are trees.
  We pick the path from the starting point v_s to the ending point v_e use MST. i.e. we want to use the edges from the MST, to form a path between the two stations/vertices, and this path is the desired path.

- **Proof:**
  Let the MST of the graph be $T = (V, E_T)$.
  Let such route be R from the starting point v_s to then ending point v_e, which consists of the edges $E_r = \{e_1, e_2, \ldots, e_k\}$. For every edge $e \in E_r$, $e \in E_T$

  Proof by contradiction:

  Assume that there exists a route R' in the graph, whose edges are $E'_r = \{e'_1, e'_2, \ldots, e'_t\}$. And for every edge $e' \in E'_r$, $w(e') < max\{w(e_1), w(e_2), \ldots, w(e_k)\}$, i.e., $max\{w(e'_1), w(e'_2), \ldots, w(e'_t)\} < max\{w(e_1), w(e_2), \ldots, w(e_k)\}$

  Let $max\{w(e_1), w(e_2), \ldots, w(e_k)\}$ be $w(e_m)$, i.e., e_m is the edge which has the highest weight.

  Let $e_m = (u, v)$

  Now we consider a cut $(S, V - S)$(this notation of cut is similar to the one in CLRS, page 626) crossing e_m and one edge e'_n in R' such that the T is cut into two halves/components. That is, if we remove e_m, then T becomes a forest with the two components, we let the two components/halves be H_1 and H_2 respectively and the cut $(S, V-S)$ respects both H_1 and H_2.

  Since $max\{w(e'_1), w(e'_2), \ldots, w(e'_t)\} < max\{w(e_1), w(e_2), \ldots, w(e_k)\}$, and every edge in E'_r has a smaller weight than e_m does, then, $w(e'_n) < w(e_m)$.

  Then e_m is not an edge with minimum weight crossing the cut $(S, V-S)$(a light edge). Then it is not an edge in the MST T for the graph, because e_m and e'_n connects H_1 and H_2, and they are in a cycle, and they cross the cut. Then this contradicts with our assumption where we assumed that $e_m \in E_T$

Then we have proven that for all other routes R' from v_s to v_e, which consists of edges $E'_r = \{e'_1, e'_2, \ldots, e'_t\}$, $max\{w(e'_1), w(e'_2), \ldots, w(e'_t)\} \geq max\{w(e_1), w(e_2), \ldots, w(e_k)\}$

Therefore, R is the best route, i.e., the route/path from the MST is the desired path.

c.

- 1) Use the coordinates to build a completely undirected weighted graph $G = (V, E)$ which represents the map of the bike pump stations. The weights are the straight distances between stations. Use **Kruskal's algorithm** on $G = (V, E)$ to find the set of edges for the MST of the graph.
- 2) Use the set of edges returned from step 1) to build an adjacency list of the MST T. Then starting from the starting point/vertex v_s, we do **BFS** one the MST T. Once we discovered the ending point/vertex v_e we stop searching.
- 3) Extract the best route from the BFS tree: start from the ending point/vertex v_e, we initialize a pointer *cur* to point to it, then do the loop that has the following three steps(i. ii. iii.):

  i If $cur == v_s$, break;
  ii. Append the vertex pointed by *cur* to a doubly linked list L;
  iii. $cur = cur.parent$;

- 4) Swap the head and tail of the doubly linked list L, and return L, which is the best route we desire.

d.

Let $|E| = m$, $|V| = n$, in this question, $m = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \in O(n^2)$ since the graph is completely connected.
Time complexity for each step 1), 2), 3) and 4) of our algorithm mentioned in part c.

- 1) Takes $O(mlogn) = O(n^2 logn)$(the time complexity of Kruskal's algorithm)
- 2) Takes $O(m+n) = O(n^2+n) = O(n^2)$ (the time complexity of BFS)(Building the adjacency list might also take some time but it cannot exceed $O(n^2)$)

- 3) Takes $O(n)$ since there are at most $n-1$ edges in the MST.

- 4) Takes $O(1)$

Therefore, the time complexity of the algorithm is $O(n^2 logn) + O(n^2) + O(n) + O(1) = O(n^2 logn)$