Jose Marquez Jaramillo
EN.605.621 Foundations of Algorithms - Spring 2023
April 10, 2023

Homework 5

Question 1.

Consider that the graph G has an edge denoted $e = \{x, y\}$ which does not belong to the tree T. Because T is a Depth First Search (DFS) tree, the condition of one of the two ends being an ancestor of the other needs to be satisfied. Also, since T is also a Bread First Search (BFS) tree, the distance of the two nodes from U in T can differ by at most one.

Now, in the case that x is an ancestor of y, and the distance from u to y in T is at most one greater than the distance from u to x, then x has to be the direct parent of y in T. Because of this logic, it follows that $\{x, y\}$ is an edge of T, contradicting that $\{x, y\}$ did not belong to T. Because of this contradiction $T \neq G$.

Question 2.

(a) By the problem statement we know that there are n people and n nights. We, therefore, need to assign each person p_i to a night, say, d_j; such that p_i is available to cook on the night d_j. It is then possible to conclude that assigning persons available to available nights to cook is a matching problem. In particular, since it is required to assign a person to a particular night, the assignment should be a perfect match. A possible solution to this problem can involve:
  (i) Build a bipartite graph. That is a graph that involves two sets of vertices in which one is the set of people $\{p_1, p_2, p_3, ..., p_i\}$; and the other is the set of nights $\{d_1, d_2, d_3, ..., d_j\}$.
  (ii) An an edge $\{p_i, d_j\}$ to the graph if the person p_i is available to cook on night d_j.
  (iii) To find whether there is a perfect matching or not, we can use maximum flow algorithm.
  (iv) Add two more vertices, source vertex(s) and sink vertex(t) to bipartite graph. Add edges from s to each person. Also, add edges from each night to t.
  (v) Set each edge's capacity to 1.
  (vi) Now, run the maximum flow algorithm. The schedule will be feasible, if and only if the maximum flow is n and the graph has exactly n edges (perfect matching).

If there is a maximum flow with value n and the graph has exactly n edges, each person will be joined to a distinct night. Thus, the matching gives a feasible dinner schedule. If there is no such maximum flow, there will be no feasible dinner schedule.

(b) The problem states that p_i and p_j both are assigned to d_k and nobody is assigned to cook on night d_l. This gives:
  - Whenever there is a feasible schedule under such conditions, the following are feasible scenarios:
    - $d_k \notin S_i$ and $d_l \notin S_j$, or,
    - $d_k \in S_i$ and $d_l \notin S_j$, or,
    - $d_k \in S_i$ and $d_l \notin S_j$, or,
    - $d_k \notin S_i$ and $d_l \in S_j$, or,
    - $d_k \in S_i$ and $d_1 \in S_j$.
  - If $d_k \notin S_i$ and $d_l \not{in}S_j$ then it is enough to add edge $\{p_j, d_l\}$. Otherwise, we need to find an edge $\{p_q, d_r\}$ to assign p_q to either d_l or d_k and need to assign either p_i or p_j to d_r.

Algorithm 1 presents a possible implementation. In terms of the time complexity of the algorithm. Note the following:
  (a) The loop at line 9 takes $O(n)$, since it is in the feasible dinner schedule and therefore will there be only n edges.
  (b) The worst case occurs in line 10. Where checking for membership takes $O(n)$.
  (c) Therefore, the overall worse time complexity corresponds to a linear search nested in a linear loop, or a $O(n^2)$ time complexity.

Algorithm 1. FIXSCHEDULE

if $d_k \notin S_i$ **And** $d_l \notin S_j$ **then**
  Update edges in the graph $\{p_i, d_k\}$ and $\{p_j, d_l\}$ and exit the loop
  for each edge $\{p_i, d_j\}$ **do**
    print(p_i is assigned to night d_j)
  return 1
Flag1=False
Flag2=False
if $d_k in S_i$ **then**
  for each edge $\{p_q, d_r\}$ **do**
    if $d_k \notin S_q$**And**$d_r \notin S_i$ **then**
      Update edges in the graph $\{p_i, d_r\}$ and $\{p_q, d_k\}$
      Flag1=True
      exit the loop
else
Flag1=True
if $d_l in S_j$ **then**
  for each edge $\{p_q, d_r\}$ **do**
    if $d_l \notin S_q$**And**$d_r \notin S_j$ **then**
      Update edges in the graph $\{p_j, d_r\}$ and $\{p_q, d_l\}$
      Flag2=True
      exit the loop
else
Flag2=True
if Flag1==True **And** Flag2==True **then**
  for each edge $\{p_i, d_j\}$ **do**
    print(p_i is assigned to night d_j)
  return 1
else
return 0

Question 3.

(a) A directed graph G can be constructed. In order to build this graph, we can scan through the ordered triples in the trace data, maintaining an array pointed to linked lists associated with each computer C_a. For each (C_i, C_j, t_k) we come across in our scan, we can create nodes (C_i, t_k) and (C_j, t_k), and create directed edges joining these two nodes in both directions. We can also append these nodes to the lists for C_i and C_j respectively. If this is not the first triple involving C_i, then we must include a directed edge from (C_i, t) to (C_i, t_k), where t is the timestamp in the preceding element in the list C_i. The same will be executed for C_j. By explicitly keeping these lists for each node, we are able to construct all these new nodes and edges in constant time per triple.

Given a collection of triples, we need to decide if a virus at computer C_a at time x could have infected computer C_b by time y. Walking through the list for C_a until we get to the last node $(C_a, x'), x' \leq x$. Running a directed Binary First Search (BFS) from (C_a, x') to determine all node that are reachable from it. If a node in the form $(C_b, y'), y' \leq y$ is reachable, then we can definitively declare that C_b is likely to have been infected by time y from C_a. Otherwise, we declare the opposite.

(b) In terms of running time; each triple in the trace data causes us to add a constant number of nodes and edges to the graph. Therefore the graph should have $O(m)$ nodes and edges. Since we build the graph in constant time per node and edge, this takes $O(m)$. Running BFS takes linear time in the size of the graph, so this too takes $O(m)$.

(c) In terms of correctness, the main claim is that is there is a path from (C_a, x') to (C_b, y') then C_b could have been infected by time y. This is rather easily explained by checking the movement of the virus between computers C_i and C_j at time t_k, whenever an edge from (C_i, t_k) to (C_j, t_k) is traversed by the BFS. This is a viable sequence of transmission of the virus that results in the virus leaving initially C_a at some time x or later and infecting C_b by time y.

Question 4.

Consider a directed graph say $G = (V, E)$. In G the set of vertices $X \subset V$ are designated as populated vertices and the set of vertices $S \subset V$ are designated as safe vertices. The two sets X and V need to be disjoint.

A set of evacuation routes from populated vertices to safe vertices is defined as:

- Each vertex in the set X is the tail of one path
- The end vertex in the path should lie in S
- The paths do not share any edges

(a) In the graph G, the evacuation routes should not overlap in order to create "no congestion". For this, we need to make the weight of each edge 1 in order to ensure this condition. Evacuation routes exist if the maximum flow of the flow network is equal to $|X|$. Here, $|X|$ is the capacity of each edge from S to sink t. Therefore, there exists $|X|$ escape routes.

Using the Ford Fulkerson algorithm, the run time is $O(V.E^2)$. We, therefore, use $|X|$ unit capacity edges from the sources s to X. From X to S, there exist unique edges. From S, there exists $|X|$ paths to sink t. Therefore, the maximum flow is $|X|$. The maximum flow of the flow network is equal to $|X|$. Thus, a set of evacuation routes exists.

(b) Under this new case, the third condition for the evacuation route will need to change. The third condition is that "the paths do not share any vertices". In graph G, the evacuation routes should not overlap in order to ensure "no congestion". In order to remedy this situation, we can split each vertex into two vertices except the vertices in the sets S and X. For instance, the vertex a is divided into a_1 and a_2. The vertex a_1 will have incoming edges and the vertex a_2 will have outgoing edges. This ensures "no congestion" in the flow network. The evacuation routes will not share any vertices.

(c) Consider a graph G with nodes x_1, x_2, x_3, s_1, s_2 and edges $(x_1, x_2), (x_2, s_1), (x_2, s_2), (x_3, s_2)$. Under this graph required routes exist in G. However, these can be any number of interconnecting nodes between X and S.