CPSC 320 2018W2: Assignment 2

Please follow the guidelines given in Assignment 1 for submission to Gradescope, and for group collaboration. Remember to provide short justifications for your answers. Submit by the deadline **Friday February 1, 2019 at 10PM**.

If you are using the .tex file to prepare your answers, then for questions where you need to select a circle you can change `\fillinMCmath` to `\fillinMCmathsoln` for your choice of answer.

All of the problems in this assignment concern graphs. Throughout, unless otherwise specified, graphs have n nodes and m edges. A node's *degree* is the number of edges incident on it. Assume that graphs have neither self-loops (edges from a node to itself), nor multiple edges between the same two nodes.

A *path* of length k is a list of $k+1$ nodes, such that there is an edge of the graph between each consecutive pair of nodes. The *shortest path* between two nodes is a path with the minimum number of edges. A *simple path* repeats no node. A *cycle* is a path that starts and ends at the same node. A *simple cycle* repeats no node other than the starting/end node (which only appears twice).

1 Graph Data Structures

Suppose that the edges of undirected graph $G = ([1..n], E)$ are represented using *adjacency lists*: for each i, there is a (not necessarily sorted) linked list $A[i]$ of the nodes j such that $(i, j) \in E$. Suppose furthermore that for each i, the degree of i is also stored, and so can be obtained in $\Theta(1)$ time. With these assumptions, what is the worst-case running time of the most efficient algorithm you can think of to solve each of the following problems, using basic graph algorithms familiar to you?

1. Given two nodes i and j of G, determine if they are adjacent.

   [✓ $O(n)$] ○ $O(nm)$
   ○ $O(n \log n)$ ○ $O(n^2 m)$
   ○ $O(n^2)$ ○ $O(nm^2)$
   ○ $O(n+m)$ ○ None of these

   Simply go through all the adjacency list of i and see if there is the node j, this will take at worst $O(n)$ time because the size of the adjacency list can be at most $n-1$.

2. Determine if G is a star. An undirected graph is a *star* if for some node i (called the star centre), $E = \{(i, j) \mid i \neq j, 1 \leq j \leq n\}$. For example, if $n = 4$ then the graph with edges $\{(1, 2), (2, 3), 2, 4)\}$ is a star with centre 2. (Recall that for undirected graphs, the order of pair of nodes in an edge does not matter.)

   [✓ $O(n)$] ○ $O(nm)$
   ○ $O(n \log n)$ ○ $O(n^2 m)$
   ○ $O(n^2)$ ○ $O(nm^2)$
   ○ $O(n+m)$ ○ None of these

   Go through all nodes and check the degree of all, there must be only one node of degree $n-1$, all the other nodes must have degree 1. This will be $O(n)$ because we are jut looping through the nodes.

3. Determine if G contains a triangle, that is, three nodes i, j, k such that i and j are adjacent, i and k are adjacent, and j and k are adjacent.

○ $O(n)$	○ $O(nm)$
○ $O(n \log n)$	○ $O(n^2m)$
○ $O(n^2)$	○ $O(nm^2)$
○ $O(n+m)$	[✓ None of these]

This can be solve using a DFS algorithm with some modifications. We will go through each node i in G (this will have a running time of $O(n)$). Then with each i do a DFS with a depth limit set to 3. So if we do not find the node i repeated at depth 3 then we know that there is no other nodes j and k that make a triangle between i, j and k. This DFS will have a worst running time of (n+m), so the algorithm will have a complexity $O(n^2 + nm)$

4. Find a node i of G that minimizes $d(i) = \max_j\{dist(i,j)\}$, where $dist(i,j)$ is the length of the shortest path connecting i to j. (Again, it's not obvious for this problem that there might not be a better algorithm, but this seems to be the best possible with simple graph algorithm techniques.) To calculate $dist(i,j)$ we will use BFS that will take complexity $O(n+m)$ and there are only n choose 2 edges with different (i,j) values, this is because (i,j) is the same that (j,i). So wee need to run BFS n choose 2 edges, the complexity would be $O(n^3 + n^2m)$

○ $O(n)$	○ $O(nm)$
○ $O(n \log n)$	○ $O(n^2m)$
○ $O(n^2)$	○ $O(nm^2)$
○ $O(n+m)$	[✓ None of these]

2 Graph Properties

Each of the following problems presents a scenario about a graph and a statement about that scenario. For each one, indicate by filling in the appropriate circle whether:

- The statement is **ALWAYS** true, i.e., true in *every* graph matching the scenario.
- The statement is **SOMETIMES** true, i.e., true in some graph matching the scenario but false in another graph.
- The statement is **NEVER** true, i.e., true in *none* of the graphs matching the scenario.

1. **Scenario:** An undirected graph containing two simple paths that share no edges, each of length k for some integer $k \geq 2$. **Statement:** $n > k + 1$.

   [✓ ALWAYS]
   ○ SOMETIMES
   ○ NEVER

   The smaller instance of the problem would be a graph with 5 nodes all connected in a line. The k in this instance would be 2 so the statement is true $5 > 2 + 1$. If we add one node to one size the n will increase by 1, the statement still true $6 > 2 + 1$. The only way to increase k is by adding two nodes one on each end of the two paths, this will increase the k by 1 and n by two. Which preserves the statement $7 > 3 + 1$.

   There is no way that the k is increased when the n doesn't so by adding more nodes two those two paths or to any part in the graph, the difference between n and k will be larger every time.

2. **Scenario:** A rooted tree found by running DFS from some node of a connected, undirected graph with $n \geq 2$. **Statement:** Every node in the tree has at most two children.

○ ALWAYS

[✓ SOMETIMES]

○ NEVER

This depends on the topology of the graph and which metric the DFS uses to determine the order in which adjacent nodes are visited. An example that proves that the statement is not always true, is with the definition of a star graph from the excercise **1.2**.

If we run DFS on a star graph G starting from the center of the star, then our rooted tree found by our DFS will have the root node with $n-1$ children. So if G has $n <= 3$ the statement will be true the DFS graph will have at most 2 children. But if $n > 3$ then the statement is false, i.e., for $n = 4$ the root node of the DFS graph will have 3 children.

3. **Scenario:** A weakly-connected but **not** strongly-connected, directed graph with $n \geq 2$. **Statement:** A simple cycle of length $n+1$ exists.

SOLUTION

○ ALWAYS

○ SOMETIMES

[✓ NEVER]

There can't be a simple cycle of length $n+1$ in any graph, because a cycle of length $n+1$ needs to be connected with $n+2$ nodes, so two nodes will have to be repeated, on a simple cycle there can't be more repeated nodes rather than the start/end node, so it is impossible that a **simple** cycle of length $n+1$ exists in any directed graph.

3 Graph Application: Tutorial Assignments

We give definitions of two problems here; your task is to reduce the first to the second. (After reading and understanding the two problem definitions, and before continuing on to read more, make some notes for yourself of what is needed to do a reduction and analyze its running time, and then what is needed to show that the reduction is correct. The rest of this problem breaks these parts down for you, but it's good to get practice at breaking the parts down for yourself too.)

- **Tutorial Assignment Problem (TAP)**. An instance of TAP is given by
  - A list of tutorials $t_1, t_2, \ldots, t_k$ and their capacities $c_1, c_2, \ldots, c_k$, such that $\sum_{j=1}^{k} c_j = n$.
  - A set S_i of the tutorials that student i can attend, where the total number of students equals the total capacity n.

  A **valid** solution is an assignment of students to tutorials such that

  - the total number of students assigned to tutorial t_j is at most c_j, $1 \leq j \leq k$,
  - each student i is either unassigned or is assigned to a tutorial in S_i, $1 \leq i \leq n$, and
  - no student is assigned to more than one tutorial.

  We can write a valid solution as a set A of items of the form t_j:$\{i_1, i_2, ...\}$, where i_1, i_2, etc. are the students assigned to tutorial t_j. Not all students need be assigned to tutorials. A **good** solution is a valid solution that maximizes the number of assigned students.

- **Bipartite Matching Problem (BMP).** An instance of BMP is an undirected bipartite graph $G = (X, Y, E)$ where $E \subseteq X \times Y$. That is, the nodes of G are partitioned into two sets X and Y, and all edges of G connect a node of X with a node of Y. A **valid solution** is a matching, i.e., a subset M of E in which each node appears at most once. A **good** solution is a maximum matching.

1. Describe a reduction from TAP to BMP.

   Part (i): Transform an instance I of TAP to an instance I' of BMP.
   To transform the instance I into I' we need to create an undirected bipartite graph G. In this graph, one partition will be filled with n students nodes. The other partition will be n nodes that each one is the representation of a 'seat' in each tutorial, so for every tutorial t_j it will be c_j nodes that we will denote as $seat_{t_j x}, 1 <= x <= c_j$. In this way we ensure that there will be no more than c_j students assigned to the tutorial t_j. So the graph will have n students and n 'seats' of all the tutorials.

   The graph also need to have edges between students and 'seats', this edges will represent that a student i is capable of attend to a tutorial t_j. So to create this edges, we loop through the set S_i of each student i and for every tutorial t_j, that i can attend, we create an edge between i and every 'seat' node of the tutorial t_j in G.
   This process will give us the graph G that we can use to solve BMP.

   Part (ii): Transform a solution M' for instance I' of BMP to a solution A for instance I of TAP.
   The solution M' will be a set containing matches between students and 'seats' on tutorials. To transform M' in to the set A, that will contain a set of students for each tutorial t_j, we need to loop through M' and for each match $(i, seat_{t_j x})$ we will add i to the set t_j in A. This process will give us a valid solution for the instance I on the TAP problem.

2. Give the worst-case running time of part (i) of your reduction, as a big-O function of n.
   The worst-case running time, will be in the case that each student is capable of attending to all the tutorials. This will give us a big-O $O(n^2)$ because for each student i we need to add an edge to all the tutorial seats which in the worst case will be n seats for each i student.

3. Give the worst-case running time of part (ii) of your reduction, as a big-O function of n. We will need to loop through all of the matchings in the solution M' and for each one add each student to each corresponding tutorial, the length of M' will be at worst of length n. So the worst-case running time will be $O(n)$

4. There is a well-known algorithm for solving BMP with running time $O(nm)$. (We won't cover the algorithm, but it can be found in Chapter 7 of K&T's textbook.) Combining this algorithm with your reduction, what is worst-case running time of your TAP algorithm, as a function of n?
   On the worst case the instance generated instance I' will have $2n$ nodes and n^2 edges, so if the algorithm that computes the solution for BMP in $O(nm)$ then the runtime of computing I' would be n^3 since $m = n^2$, if we consider this in the reduction algorithm the whole running time will be $n^2 + n^3 + n$, that will give us $O(n^3)$.

5. Explain why your reduction is correct. Suppose that M'_{good} is a good solution (maximum matching) for I' and let A_{good} be the transformed solution (assignment) obtained by part (ii) of your reduction applied to M_{good}.

   (a) Show that A_{good} is a valid solution for I.
   The instance I' is a completely valid instance for BMP, because we created an undirected bipartite graph with the instance I for the TAP. Therefore M'_{good} must be a valid solution for BMP.
   Then M'_{good} must have a set of matchings between students and tutorial seats, in which each student i appears at most once, every tutorial seat $seat_{t_j x}$ is link to at most 1 student, and there are at most c_j 'seats' of tutorial t_j. So by visiting each matching in the set M'_{good} we can create a set A_{good} and for each student in M'_{good} we add it to the corresponding tutorial t_j in A_{good}. Giving us at the end a valid solution with the instance I for TAP.

   (b) Show that A_{good} is a good solution for I, i.e., maximizes the number of assigned students.
   If M'_{good} is a good solution to BMP, meaning that it have the maximum matching in I' which

represents students and tutorials seats, then A_{good} will be a good solution for TAP, because it will have the most number of students matched for a tutorial, while still being a valid solution for TAP (proved on point 3.5.a).

4 Graph Application: Network Connectivity

(Adapted from Problem 9, Chapter 3 of K&T) Think of a communications network as a connected, undirected graph, where messages from one node s to another node t are sent along paths from s to t. Nodes can sometimes fail. If a node v fails then no messages can be sent along edges incident on v. A network is particularly vulnerable if failure of a single node v can cause two nodes, say s and t, to become disconnected. That is, when v and its incident edges are removed from the network, there is no path from s to t. We call such a node v a *vulnerability*.

1. Suppose that connected, undirected graph G contains two nodes s and t such that the shortest path between s and t is strictly greater than $n/2$. Show that G must contain a vulnerability.
   In order to have a path between two nodes (u and v) that is strictly greater than $n/2$ in G, there must be an articulation node w in the graph. We can see w as a bottleneck where the path must pass to reach v from u.

   Because G is complete if there are no vulnerabilities all the paths from any node to another node are at most $n/2$ in length. The only way to increase the length of the longest path in G is to add an extra node x and connect it to a node which we called w, this will increase the length of the longest path to be greater than $n/2$. But this will introduce a vulnerability to G in the node w.

   So if w is disconnected from the graph it will create at least two separate connected components, in one of them it will be all the first part of the graph and in the other one it will be the node x that we connected to w. So we can't reach x from any other node in the G, hence w will be a vulnerability.

2. Give an algorithm that, given as input G and a node s such that there is some node t for which the distance between s and t is guaranteed to be greater than $n/2$, finds a vulnerability. Your algorithm should have running time $O(m+n)$.

```
Algorithm Find-Vulnerability ($V, E, s, depth = 0$)
    // Initialize queue
    let $n$ be size of $V$
    let $q$ be a queue
    mark $s$ as visited
    set $s$.depth to 0
    $q$.enqueue($s$)
    // Iterate on the queue until is empty
    while $q$ is not empty:
        let $t$ be $q$.dequeue
        // Go through all neighbours of $t$
        For all neighbours $v$ of $t$:
            if $v$ is not marked:
                // If the path from $s$ to $v$ has a length greater than $n/2$ return $t$
                if $t$.depth $+ 1 > n/2$:
                    return $t$
                // If not add $v$ to queue
                mark $v$ as visited
                set $v$.depth to $t$.depth $+ 1$
                $q$.enqueue($v$)
    // No vulnerability found
    return null
```

The algorithm above has a running time $O(m+n)$ because it simply does a BFS from the node s, keeping track of the depth and when the depth reaches a level greater than $n/2$ then the node that creates that path in that depth, must be a vulnerability.