CS161 Homework 4 **Due:** 13 May 2016
Submit on Gradescope **Handed out:** 6 May 2016

Instructions: Please answer the following questions to the best of your ability. If you are asked to design an algorithm, please describe it clearly and concisely, prove its correctness, and analyze its running time. If you are asked to show your work, please include relevant calculations for deriving your answer. If you are asked to explain your answer, give a short ($\sim$ 1 sentence) intuitive description of your answer. If you are asked to prove a result, please write a complete proof at the level of detail and rigor expected in prior CS Theory classes (i.e. 103). When writing proofs, please strive for clarity and brevity (in that order). Cite any sources you reference.

1 (14 points) The streets of Bearville

Barr the bear is the mayor of the newly established city of Bearville near the North Pole. He learns that Po the Penguin is a master civil engineer from the South Pole and hires Po to design the Bearville city streets. Po comes back and in his design all the streets are one-way! Barr demands that Po proves to him very quickly that one can reach any street intersection from any other street intersection. (Note that multiple streets can meet at an intersection.) In this question we will help Po:

(a) Give a graph formulation of the above problem: what are the vertices and edges of the graph? What computational problem does Po need to solve?

Solution: Intersections are the vertices and one-way streets are directed edges between a pair of vertices.

We are trying to answer the following question: Is it true that for every pair of vertices $x, y \in V$, there exists a path from x to y and a path from y to x?

(b) Let m be the number of edges and n be the number of vertices in the graph. Give an algorithm to solve the intersection reachability problem in time $O(m + n)$. Prove why your algorithm is correct and why it runs in $O(m + n)$ time.

Solution: One can call DFS(s) from every node $s \in V$, to check if every other node can be reached from s, but this leads to $n(n + m)$ runtime.

Instead we will choose an arbitrary node $s \in V$, and first run DFS(s); if we cannot reach some node from s, then we output NO. Otherwise, we know that from s we can reach every node in the graph. Now we reverse the direction of edges (which can be done by scanning each vertex and edge once in time $O(n + m)$), and call DFS(s) again. This time, since the edges are reversed, if we can reach some node x from s, it means that in the original graph we can reach s from x. Again, if any node x cannot be reached from s in the reversed graph, we output NO. Otherwise we output YES. Clearly this algorithm runs in $O(n + m)$ time.

(Note that if you use adjacency matrix, the runtime of DFS will not be $O(n + m)$ but $O(n^2)$.)

Correctness:

First suppose that we can reach from every node to every other node. We shall show that our algorithm will output YES. As we pick any node s and run DFS(s) for the first time, by our assumption our algorithm will not output NO since we can reach all nodes from s. Now after we reverse the direction of edges, suppose that we cannot reach some node x from s in the reversed graph. Yet we know that we can reach s from x in the original graph; that is, there is some path P from x to s, whose reversal P' should be a path from s to x in the reversed graph. This is a contradiction. This proves that if we can reach from every node to every other node, then algorithm will output YES. Now suppose our algorithm outputs YES. Then we know that from the first DFS call, we can reach every node from s. From the second DFS call, we know that we can reach s from every node. Hence our algorithm outputs YES if and only if we can reach every node from every other node.

2 (12 points) Touring Parallel Bearville

In a parallel universe, Po-Prime has designed the streets to be one-way and moreover, *for every* intersection i there is *no way* to start at i and get back to i by following streets. This is very disconcerting to Barr-Prime. Po-Prime, however claims that there is an intersection s and a way to tour the city starting from s, traversing every intersection *exactly once*. We call this tour an *exact traversal*. (If we believe Po-Prime, then we can fix his design as follows: find the starting and ending intersections s and t of the exact traversal and add a street going from t to s – now every intersection is reachable from any other intersection.)

In this problem we will check whether Po-Prime is right.

Let n be the number of intersections and m be the number of streets as in the previous problem. Design an $O(m+n)$ time algorithm that determines whether an exact traversal exists, and if so, returns the order in which the intersections are traversed. Prove why your algorithm is correct and why it runs in $O(m+n)$ time.

Solution: Again, intersections are the vertices and streets are the edges. We run topological order in this graph, and let $v_1, v_2, \ldots, v_n$ be the topological order of vertices. We will check whether there is an edge between any two consecutive vertices in the topological order (i.e., $(v_i, v_{i+1}) \in E$?). If yes, then this corresponds to an exact traversal; otherwise, there is none. To see why this algorithm runs in $O(n+m)$ time, we know that topological order can be found in $O(n+m)$ time from class; after that, we simple check the list of n vertices to see if $(v_i, v_{i+1}) \in E$ for each $i = 1, \ldots, n-1$. To check whether $(v_i, v_{i+1}) \in E$, we simply iterate over $N_{\text{out}}(v_i)$ to test membership of v_{i+1}, and in the worst case this iterates over E as we need to look up every out-going edge from every node. Overall this can be done in $O(n+m)$ time.

(Note that if you use adjacency matrix, the runtime of topological order will not be $O(n+m)$ but $O(n^2)$.)

Correctness: If there is an exact tour in the given graph (without loss of generality assume it is $v_1, v_2, \ldots, v_n$), then the topological order is exactly the order of the tour. To see why, v_1 cannot have an incoming edge (or one can start from v_1 and return to v_1, contradicting the assumption in problem statement). All other vertices have at least one incoming edge. This argument applies to any sub-path of the exact tour from v_i to v_n, and thus we get the topological order that is exactly the order of the tour. Now suppose that the topological order of the given graph is a path that goes through all vertices, then the path is the exact tour by definition. This proves correctness of the algorithm.

3 (12 points) Negative Edges

In lecture, we discussed how Dijkstra's Algorithm does not handle graphs with negative edge weights. In this question, we'll explore why this is the case. For this question assume that all edge weights are integers.

(a) Draw a graph G, which contains both positive and negative edges but does not contain negative cycles, and specify some source $s \in V$ where $\mathsf{Dijkstra}(G, s)$ does not correctly compute the shortest paths from s. Your graph should have the minimum number of vertices possible. Show which paths are computed incorrectly and explain why $\mathsf{Dijkstra}$ fails.

Solution: Consider the graph on 3 nodes s, a, b with edges (s, a) of weight 2, (s, b) of weight 1, and (a, b) of weight -3 where the source is s. Then Dijkstra will extract s, set $d[a] = 2, d[b] = 1$, then extract b setting its distance permanently to 1 but it's actual distance is $2 - 3 = -1$. The graph doesn't have cycles so in particular it doesn't have negative cycles. The reason why Dijkstra didn't work was because the shortest path to b started with a heavy weight edge that made it look bad and b was extracted from the heap too early. (There's no way for this to happen in a graph with nonnegative weights since the weight of any subpath is a lower bound for the weight of the path.)

(Note that we posted a clarification note about Dijkstra's algorithm such that it should permanently mark $d[x]$ when x is deleted from F. Yet if your answer did not assume this, we will grade it accordingly, so you would not be penalized for not having read the clarification.)

(b) Consider the algorithm Negative-Dijkstra for computing shortest paths through graphs with negative edge weights (but without negative cycles).

Algorithm 1: Negative-Dijkstra(G, s)

```
$w^* \leftarrow$ minimum edge weight in $G$;
for $e \in E(G)$ do
    $w'(e) \leftarrow w(e) - w^*$;
$T \leftarrow$ Dijkstra$(G', s)$   // Run Dijkstra with updated edge weights to get a SSSP Tree
return weights of $T$ in the original $G$;
```

Note that Negative-Dijkstra shifts all edge weights to be non-negative (by shifting all edge weights by the smallest original value) and runs in $O(m + n \log n)$ time.

Prove or Disprove: Negative-Dijkstra computes single-source shortest paths correctly in graphs with negative edge weights. To prove the algorithm correct, show that for all $u \in V$ the shortest $s - u$ path in the original graph is in T. To disprove, exhibit a graph with negative edges, with no negative cycles where Negative-Dijkstra outputs the wrong "shortest" paths, and explain why the algorithm fails.

Solution: The same graph as before works. The min weight is -3 and when we reweight, the edges have the following weights : (s, a) of weight 5, (s, b) of weight 4, (a, b) of weight 0. Dijkstra will make the same mistake as above. The reason why reweighting by the min edge doesn't work is because longer paths get penalized more; in particular, a path of length l gets a weight increase of $|\text{min weight}| \cdot l$, so that if the shortest path actually has many edges it might still look bad due to the reweighting and will be overlooked.

4 (14 points) Po and Barr party

Po the penguin and Barr the bear have the same set of n friends, $F = \{f_1, f_2, \ldots, f_n\}$, but Po and Barr don't really like each other. Po decides to have a party. After learning this, Barr decides to have a party at the same time as Po. Among their common friends various pairs don't like each other so would prefer not to go to the same party. You are given a set of enemy pairs $E = \{(f_{i_1}, f_{j_1}), \ldots, (f_{i_m}, f_{j_m})\}$ and you need to decide whether the friends in F can be partitioned into two parts B and P (i.e. each friend is in either P or B but not both) so that no pair from E appears together in B or P (and hence the friends in B can go to Barr's party and the friends in P can go to Po's party and both parties will be without conflict).
That is, give an $O(m+n)$ time algorithm that given $F = \{f_1, f_2, \ldots, f_n\}$ and $E = \{(f_{i_1}, f_{j_1}), \ldots, (f_{i_m}, f_{j_m})\}$

- either finds a partition of F into B and P so that for every $(f_{i_r}, f_{j_r}) \in E$, $|B \cap (f_{i_r}, f_{j_r})| = 1$ and $|P \cap (f_{i_r}, f_{j_r})| = 1$,
- or determines that no such partition exists.

Prove the correctness and runtime of your algorithm.
Solution: Form a graph on the friends in F (as vertices) with the pairs in E as edges. Without loss of generality assume the graph is connected (otherwise find the connected components, say with DFS, and then run our algorithm on each component separately; alternatively we can just run our algorithm below from each node that has not been "marked" previously).

Algorithm: Pick an arbitrary node v and run BFS from it. Recall that $v \in L_0$ where L_i is the set of nodes whose distance from v is exactly i. If BFS ever encounters an edge between two nodes in the same level (i.e. the same distance from v), then return FALSE. Otherwise, add to B be the union of all L_i where i is even and add to P be the rest of the nodes. Clearly this algorithm runs in $O(n+m)$ time as we learned in class that BFS runs in time $O(n+m)$. In case there are multiple connected components in the graph, it still takes $O(n+m)$ time as BFS on each connected component runs in time $O(n'+m')$ where n' is the number of nodes in the component and m' the number of edges, and the sum over all connected components would yield $O(n+m)$.

Correctness: We will first prove that if such partition exists, then all nodes whose distance from v is even must be in the same partition as v and those whose distance from v is odd must be in the other partition (v is any node we choose); moreover, if such partition exists, there should be no edge within each partition (by definition).

Without loss of generality we can put v in B. Then all its neighbors must be in the other partition (e.g. P). By induction (on distance from v), every node at odd distance must be in a different partition from v (e.g. in P) and every node at even distance must be in the same partition. The base cases are when the distance is 0 (in which case we put v in B) or 1 (in which case we put the nodes in P). By inductive hypothesis for any node whose distance is less than i from v must be in B (in P) if the distance is even (odd). For any node w whose distance is exactly i from v, there must be a (shortest) path from v to w of length i. In particular, if z is the node on this path just before w, then we know that z is distance $i-1$ away from v. If i is even, then $i-1$ is odd and z is in P (by I.H.), and therefore w must be in B. Likewise if i is odd, then $i-1$ is even and z is in B (by I.H.), and therefore w must be in P. In summary, if we put v in B to begin with, then every node whose distance from v is even (odd) must be in B (in P), and this is exactly what our algorithm does via BFS. Hence, if there exists a valid partition of friends, the algorithm will find it.

Now suppose that no such partition exists. In this case, we can still partition the nodes into B and P by putting all nodes whose distance from v is even in B and odd in P. Since there is no valid partition, there must be at least one edge within B or at least one edge within P or both. In any of these three cases, our algorithm will detect such edge and return FALSE; to see why, suppose that x, y are both in B and $(x, y) \in E$; in particular $d(v, x)$ and $d(v, y)$ are even. If $d(v, x) \neq d(v, y)$ then they differ by at least two (since both are even), but this is a contradiction because there is an edge between (x, y) and therefore the distance to one of them must be odd. Hence, x, y must belong to the same level, $L_{d(v,x)}$. Since our algorithm checks whether there is an edge within each level, it will return FALSE correctly.