CS 124 Homework 2: Spring 2021

Your name: Sean Ty

Collaborators: Tanishq Kumar

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

Homework is due Wednesday 2021-02-17 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style will count in your grades. Be sure to read and know the collaboration policy in the course syllabus. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

For all homework problems where you are asked to design or give an algorithm, you must prove the correctness of your algorithm and prove the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

1. **(10 points)** We saw in lecture that we can find a topological sort of a directed acyclic graph by running DFS and ordering according to the postorder time (that is, we add a vertex to the sorted list *after* we visit its out-neighbors). Suppose we try to build a topological sort by ordering according to the preorder, and not the postorder, time. Give a counterexample to show this doesn't work, and explain why it's a counterexample.

   One counterexample is a directed graph with vertices A, B, C with edges (A, C) and (B, C) (and no other edges). The preorder idea then doesn't work because if we, say, did DFS starting from A, then the orders would be $A : [1, 4], B : [5, 6]$, and $C : [2, 3]$, so pre-order would give A, C, B as the topological sort. But this does not work since we have an edge from B to C.

2. **(15 points)** News from Cambridge, for those of you far away: every night, snow falls and covers all the sidewalks. Every morning, the city's lone snow shoveler, Pat, is tasked with clearing all the sidewalks of snow. Proper snow-shoveling technique requires that sidewalks on opposite sides of the same street be shoveled in opposite directions. (Every street has sidewalks on both sides.) Give an algorithm to find a snow-shoveling path for Pat that doesn't require any more walking than necessary—at most once per sidewalk. (If you have to assume anything about the layout of the city of Cambridge, make it clear!) (Your algorithm should work for any city, not just the Cambridge in which Harvard is.)

   Let G be a graph such that each of its edge is a street and each vertex is an endpoint of a street. Then it suffices to find an algorithm such that Pat traverses each edge in both directions. Note that the graph must be connected, because otherwise traversing all edges would be impossible. Therefore, assume that G is connected.

   Perform a modified DFS starting at any vertex doing the following: at first, we are at the source. Then, pick any unvisited edge incident to the current vertex and mark it as visited. If the other vertex has already been visited, we traverse to that vertex and immediately back, marking the edge we traversed as visited. Otherwise, we recursively search through this vertex. Once all the edges incident to the vertex have already been traversed, we backtrack by traversing the edge we only traversed once to the previous vertex. Then, once this terminates we have traversed through all edges exactly once in each direction.

   First, we know that this algorithm must terminate since all we are doing is performing DFS but with previsit and postvisit actions (and there are finitely many edges). Next, we note that this algorithm creates a DFS tree (since the graph is connected) with non-tree edges which are traversed as soon as they are encountered, and that tree edges are eventually traversed back through the backtracking. This also implies that each edge is traversed exactly once in each direction, so this algorithm works. The time complexity is $O(|V| + |E|)$ since this checks each vertex and the edges incident to it.

3. **(15 points)** The *risk-free currency exchange problem* offers a risk-free way to make money. Suppose we have currencies $c_1, \ldots, c_n$. (For example, c_1 might be dollars, c_2 rubles, c_3 yen, etc.) For various pairs of distinct currencies c_i and c_j (but not necessarily every pair!) there is an exchange rate $r_{i,j}$ such that you can exchange one unit of c_i for $r_{i,j}$ units of c_j. (Note that even if there is an exchange rate $r_{i,j}$, so it is possible to turn currency i into currency j by an exchange, the reverse might not be true— that is, there might not be an exchange rate $r_{j,i}$.) Now if, because of exchange rate strangeness, $r_{i,j} \cdot r_{j,i} > 1$, then you can make money simply by trading units of currency i into units of currency j and back again. (At least, if there are no exchange costs.) This almost never happens, but occasionally (because the updates for exchange rates do not happen quickly enough) for very short periods of time exchange traders can find a sequence of trades that can make risk-free money. That is, if there is a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \ldots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$, then trading one unit of c_{i_1} into c_{i_2} and trading that into c_{i_3} and so on back to c_{i_1} will yield a profit.

   Design an efficient algorithm to detect if a risk-free currency exchange exists. (You need not actually find it.)

   Consider a directed graph G where its vertices v_i represent the currencies c_i, and an edge between $v_i \to v_j$ has weight $w_{ij} = -\log r_{ij}$. Then note that $\prod_{j=1}^{k} r_{i_j i_{j+1}} > 1$ iff $\sum_{j=1}^{k} w_{i_j i_{j+1}} < 0$ (indices taken modulo k, where we got this by taking logs and negating), so the problem is equivalent to finding a negative cycle in G. We can use Bellman Ford to do this, and we know from class that it can be implemented in $O(|V||E|)$ time. Initialization takes $O(|E| + |V|)$ time for the edges and vertices. Using $|E| \leq \binom{n}{2} = O(n^2)$ and $|V| = n$, it follows that the run time complexity is $O(n^3) + O(n) + O(n^2) = O(n^3)$. The space complexity will be $O(|V| + |E|) = O(n^2)$ (here we are assuming adjacency list representation).

4. **(20 points)** Give an algorithm to find the lengths of all shortest paths from a given vertex in a directed graph $G = (V, E)$ where all edge weights are integers between 0 and m, inclusive. Your algorithm should work in time $O(|E| + |V|m)$. (Hint: Modify Dijkstra's algorithm.)

   We perform the following algorithm, where $v_0, v_1, \ldots v_{n-1}$ are the vertices of G (i.e. $|V| = n$. DLL means doubly linked list):

```
# keep track of shortest distance from source so far
dist = [0]*n
# max possible distance (from problem)
maxDist = n*m
# list of queues, each has points distance k away from source
# so DLL for 0, 1, ... maxDist
nodesDist = [DLL]*(maxDist+1)
# DLL of nodes not yet reached from source
nodesDist[infty] = DLL

for v in V-{s}:
    nodesDist[infty].push(v)
    dist[v] = infty

# s is source
dist[s] = 0
iterator = 0

# iterate through vertices closest
# just like dijkstra
while iterator < maxDist:
    while !nodesDist[iterator].is_empty:
        # take top node out and search
        v = nodesDist[iterator].pop()
        for (v,w) in E:
            # if checking makes it smaller, change
            if dist[w] > dist[v] + weight(v, w)
                # remove w from its L and move it
                nodesDist[dist[w]].remove(w)
                dist[w] = dist[v] + weight(v, w)
                nodesDist[dist[w]].push(w)
    iterator += 1
```

   Note that for all other vertices in G, they are either unreachable from the source or can be attained in at most $m \cdot |V|$ distance (since there are at most $|V|$ vertices used in the path to it and each step has weight at most m). Then, by definition, nodesDist[i] keeps track of the nodes that are i away (currently) from the source. The iterator then traverses through nodesDist, emptying the DLL's along the way (so nothing gets counted twice). This works since all edge lengths are nonnegative, so a DLL that has already been emptied will no longer

be filled. Then, dist[v] is updated accordingly so that it becomes the least distance to the v from the source at our current iteration. At the end, all possibilities will have been considered, so this yields the shortest paths from s to all other vertices, as desired.

Initialization of the necessary variables takes $O(|V| * m)$ time. The for loop takes $|V|$ time, and the while loops take $O(\text{maxDist} + |V| + |E|) = O(|E| + m \cdot |V|)$ time. It then follows that the algorithm takes $O(|E| + m \cdot |V|)$ time, as desired. The space complexity here is $O(m \cdot |V| + |E|)$ for the arrays and the DLL's.

5. **(15 points)** Design an efficient algorithm to find the *longest* path in a directed acyclic graph whose edges have real-number weights. (Partial credit will be given for a solution where each edge has weight 1; full credit for solutions that handle general real-valued weights on the edges, including *negative* values.)

   We first topological sort G to get the sequence $v = [v_0, v_1, \ldots v_{n-1}]$ of vertices, where $n = |V|$.

```
# longest path length ending at vertex v_i
dist = [0]*n
# the resulting path
path = (0, -1)
# keep track of starting vertex that yields shortest
prev = [-1]*n
for i in range(n):
    for (i, j) in E:
        # note that i < j always so we only ever update j > i
        if dist[j] < dist[i] + weight(i, j):
            prev[j] = i
            dist[j] = dist[i] + weight(i, j)
```

   Then, we look for the maximum value in dist and its index, using the prev array to determine the path (keep backtracking through prev until we hit -1, since prev is only -1 if the vertex is the first in its longest path. Note that dist[j] does indeed keep track of the longest path length that ends at j, since the topological sort makes us only update dist[j] for $j > i$ when we are currently at vertex i, and it updates based on all the paths that end at j and takes the maximum length. The prev array updates accordingly, so backtracking does indeed yield the longest path. The maximum path length is then the maximum of the path lengths that end at each vertex, which proves correctness.

   For the run time of this algorithm, topological sort takes $O(|V| + |E|)$ time as discussed in class. The for loop loops over all vertices and edges once and all operations performed inside the loops take constant time, so they take $O(|V| + |E|)$ time as well. Thus, the algorithm runs in $O(|V| + |E|)$ time. The space complexity is $O(|V|)$ from the array and the topological sort requirement.

6. **(15 points)** Suppose that you are given a directed graph $G = (V, E)$ along with weights on the edges (you can assume that they are all positive). You are also given a vertex s and a tree T connecting the graph G that is claimed to be the tree of shortest paths from s that you would get using Dijkstra's algorithm. Can you check that T is correct in linear time?

   Yes we can. We need to verify the following (s is the source, $p(v)$ denotes parent of v in tree):

   (a) T is in fact a tree.

   (b) The distances $d(s, v)$ cannot be improved: i.e. for all $(u, v) \in E$, we have $d(s, v) \leq d(s, u) + \text{weight}(u, v)$.

   (c) $d(s, s) = 0$ and $p(s) = \emptyset$.

   (d) $d(s, v) = d(s, p(v)) + \text{weight}(p(v), v)$.

   (e) For unreachable vertices v, we must have $p(v) = \emptyset$ and $d(s, v) = \infty$.

   Note that checking these would be sufficient: everything else is clear, and for (b) this is because of when we do Bellman-Ford: updating the edges again should not change the tree else it is incorrect.

   To check (a), we can just perform DFS on the (undirected) version of T and check for cycles, which can be done in $O(|V| + |E|)$.

   To check (b), can keep track of $d(s, v)$ for all v by traversing through the tree and storing the values. We then look through the edges and check whether each inequality is satisfied or not. This takes $O(|V| + |E|)$ time again. We need to check this because if this was not true, then another iteration of Dijkstra will yield a smaller tree, which means the tree given was not the shortest paths tree.

   To check (c), we can do this in $O(1)$ time (a lookup).

   To check (d), we can also perform a lookup to find the parent, and verify the equality. This would take $O(|V|)$ time.

   To check (e), we look at the vertices we obtained through the DFS in (a) and perform DFS in G (from s). We then have to check that the vertices in (a) are exactly those found in the DFS in G from s. This takes $O(|V| + |E|)$ time.

   This then allows us to verify that the tree is indeed the tree from Dijkstra. The overall run time is then $O(|V| + |E|)$ and the space complexity is $O(|V|)$.

7. **(0 points, optional)**[1] This exercise is based on the 2SAT problem. The input to 2SAT is a logical expression of a specific form: it is the conjunction (AND) of a set of clauses, where each clause is the disjunction (OR) of two literals. (A literal is either a Boolean variable or the negation of a Boolean variable.) For example, the following expression is an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \overline{x_3}) \wedge (x_4 \vee \overline{x_1}).$$

A solution to an instance of a 2SAT formula is an assignment of the variables to the values T (true) and F (false) so that all the clauses are satisfied– that is, there is at least one true literal in each clause. For example, the assingment $x_1 = T, x_2 = F, x_3 = F, x_4 = T$ satisfies the 2SAT formula above.

Derive an algorithm that either finds a solution to a 2SAT formula, or returns that no solution exists. Carefully give a complete description of the entire algorithm and the running time.

(Hint: Reduce to an appropriate problem. It may help to consider the following directed graph, given a formula I in 2SAT: the nodes of the graph are all the variables appearing in I, and their negations. For each clause $(\alpha \vee \beta)$ in I, we add a directed edge from $\overline{\alpha}$ to β and a second directed edge from $\overline{\beta}$ to α. How can this be interpreted?)

8. **(0 points, optional)**[2] Give a complete proof that $\log(n!)$ is $\Theta(n \log n)$. Hint: you should look for ways to bound $(n!)$. Fairly loose bounds will suffice.

[1] We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.
[2] We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.