Colorado CSCI 5454: Algorithms
Homework 2

Instructor: Bo Waggoner
Due: September 12, 2019 at 11:59pm
Turn in electronically via Gradescope.

Problem 1 (8 points)

A *tree* is an undirected graph that (1) is connected, meaning every vertex is reachable from every other vertex, and (2) has no simple cycles. Trees are broadly used in data structures, as we will see in this problem.

Usually, when given a tree, we are also given a *root* r, one of the vertices of the tree. Consider any non-root vertex w and the path from r to w. The second-to-last vertex on this path is the *parent* of w, and w is a *child* of v. A vertex with no children is a *leaf*.

For parts (a) and (b), you do not need to prove correctness or running time of your algorithms (but it should be clear from your algorithm and the course material).

Part a (2 points) The *level* of a node is its distance from the root. Give an $O(n+m)$ time algorithm to output an ordering of the vertices by level, i.e. the root, followed by all level-1 vertices, followed by all level-2 vertices, ...

Solution. We use breadth-first search starting from the root. We print nodes in the order that they are popped from the queue/list and considered in the main loop. It is mentioned in class and shown in DPV Ch 4 that BFS considers vertices in order of distance from the input vertex, exactly as desired.

Part b (2 points) Give a linear-time $O(n+m)$ algorithm to output an ordering of the vertices where each parent appears immediately after all of its descendents. *A descendent is a child, a child of a child, or so on.*

Solution. We use depth-first search starting from the root. When we call the DFS-recurse function on v, at the very end of the function (just before returning, after making all recursive calls), we print v. (Further justification is not necessary since the problem did not require proving correctness and performance.)

Part c (4 points) Suppose that initially, each vertex v has a number $c[v]$ stored in array c. Give a recursive $O(n+m)$ time algorithm to output an array d, where $d[v]$ is the maximum number in the subtree rooted at v. (This is the tree consisting of v as the root, all of v's children, their children, and so on.) For this part, prove correctness and running time of your algorithm. *Hint for correctness: use induction, with the leaves of the tree as the base case.*

Solution. We can modify depth-first search. When we call the DFS recursive function on v, we initially set $d[v] = c[v]$. Then, in the loop, after processing each of v's children u, we update $d[v] = \max\{d[v], d[u]\}$. We then call this function on the root, let it finish, and return d.

Correctness: Suppose v is a leaf of the tree. Then the correct answer is $d[v] = c[v]$. Eventually we call the DFS-recurse function on v, which sets $d[v] = c[v]$, and since v has no children, the function call completes. So the algorithm is correct on all leaves. Now, for induction, suppose the algorithm is correct on all of v's children. Then when processing v, we consider each child and its value $d[u]$, *after* calling the recursive function on u. By inductive assumption, $d[u]$ is set correctly. So $d[v]$ is correctly set to the max of $c[v]$ and $d[u]$ for all children u.

Running time: We do a DFS on the tree and add only some constant-time operations in each function call and loop. So the running time is still $O(n + m)$.

Problem 2 (4 points)

Dasgupta, Papadimitriou, Vazirani Problem 4.8 (Prof. F. Lake problem).

Solution. One counterexample: The graph consists of $v_1, v_2, \ldots, v_{100}$. We have $s = v_1$ and $t = v_{100}$. There is an edge from each v_i to v_{i+1} of length -1. There is also an edge directly from v_1 to v_{100} of length 1, and there is an edge elsewhere in the graph (say from v_{100} to v_{101}) equal to -2. So the algorithm would add at least 2 to every edge, say it adds exactly 2. Now the original path $v_1, v_2, \ldots, v_{100}$ costs 100, but the edge directly from v_1 to v_{100} only costs 3. However, originally the long path cost -100 and the direct edge cost 1.

The problem is that the shortest path had many edges, and when we add a constant to each edge, we make this path much much longer. Compare to a path that initially has higher cost, but only one edge. When we add a constant, this path doesn't get much longer.

Problem 3 (12 points)

The following question was asked by a friend of mine and his young child:

How many rainbow paths are there (red to purple)?

A step in the path must move from one peg to another peg below (directly below or diagonally adjacent). For example, for each orange peg, there happen to be exactly two ways to reach it from a red peg.

In this question, we will design an efficient algorithm to solve this problem, first as pictured, then in general graphs.

Part a (2 points) Suppose we had a board with n colors (so it is n rows high). Argue briefly that for some peg configurations, the number of paths can be very large, for example $\Omega(2^n)$. We conclude that counting the paths one-by-one is not practical.

Solution. Suppose the board is completely full of pegs. We prove a peg in row k has at least 2^{k-1} paths to it. Base case: the first row has 1 path to each peg, since they're the starting points. Inductive step: Each peg in row $k \geq 2$ has at least two pegs above it that can step to it. By inductive assumption, each of those has 2^{k-2} paths to it. So this peg has at least $2^{k-2} + 2^{k-2} = 2^{k-1}$ paths to it. In particular, each peg in the last row has at least $2^{n-1} \in \Omega(2^n)$ paths to it. (Note this undercounts by a lot!)

Part b (1 point) In the image, for each of the three yellow pegs (left to right), how many red $\rightarrow$ orange $\rightarrow$ yellow paths are there ending at that peg?

Solution. 4, 4, 2.

Part c (1 point) In the image, how many red $\rightarrow$ orange $\rightarrow$ yellow $\rightarrow$ green paths are there ending at the leftmost green peg? *Hint: don't count them all; use your answer to Part b!*

Solution. 8.

Part d (4 points) Given a pad of post-it notes and a pen, describe a strategy for computing the number of rainbow paths. Your algorithm should be asymptotically much more efficient that counting the paths one by one. Briefly argue correctness. *Hint: start by posting a "1" on every red peg, then move on to the orange ones...*

Solution. Post a "1" on every red peg. For each orange peg, add the numbers on all the red pegs adjacent to it, and post that total on the orange peg. For each yellow peg, add the numbers on all the orange pegs adjacent to it, and post that total on the yellow peg. Continue downward in this way. When finished, add up all the numbers on all the purple pegs.

This is correct because every path to this yellow peg goes through one of the adjacent orange pegs, then hops to this yellow. So the total number is the sum of the ways to get to each adjacent orange peg, and so on down.

Part e (4 points) We are given a directed acyclic graph (DAG) $G = (V, E)$. We are given a set of *start* vertices $S \subseteq V$ and a set of *end* vertices $T \subseteq V$. Give an efficient algorithm to output the number of paths that start in S and end in T. You should argue correctness and running time. You may assume the graph is represented either as an adjacency list or an adjacency matrix.[1]

Solution.

Algorithm. We use a form of dynamic programming. We keep a count $c[v]$ for each vertex v of how many paths there are to that vertex that start in S. Initially, $c[v] = 0$ for all vertices. Then, set $c[v] = 1$ for all vertices in S.

Key step: Topologically sort the vertices.

Iterate through the vertices in topological sort order. For each v, iterate through each u such that (u, v) is an edge and set $c[v] += c[u]$.

When done, sum up $c[v]$ for all v in T and output the total.

Correctness. By induction over the topological sort order. Base case: after the first vertex considered, it is either in S (so $c[v] = 1$) or not (so $c[v] = 0$), either way correct. Induction case: when we consider v, we have already considered all vertices with an edge to v, because it's a topological sort. The total number of paths from set S to v is equal to the sum, over vertices with an edge to v, of the number of paths from S to that vertex; plus 1 if $v \in S$ because the of the path consisting of just v. This is exactly equal to $c[v]$, using the induction hypothesis.

[1] You may also assume the word size in this RAM model is large enough to hold the total number of paths in the graph.

So after the topological sort iteration, every vertex has $c[v]$ equalling the number of paths starting in S and ending at v; so if we sum over $v \in T$, we get the answer.

Running time. The initialization phase takes $O(n)$ time. Topological sort takes $O(n+m)$. Iterating through the vertices takes $O(n)$ time to consider all the vertices plus $O(m)$ time because we consider all of the edges eventually (for each v, we iterate through its neighbors). The final summation takes $O(n)$ time. In total, we get $O(n+m)$.

(Note: like DFS and BFS, we only touch each edge of the graph once, so even though the "edge loop" is inside the "vertex loop", we only get an additional $O(m)$ time, not $O(n \cdot m)$ for instance.)

Problem 4 (10 points)

This is a programming assignment. Download the undirected, unweighted Texas road network from here: `https://snap.stanford.edu/data/roadNet-TX.html` The format is as follows: the first four lines of the text file are comments and can be ignored. The remaining lines each describe an edge. Each of these lines contains two integers separated by whitespace, representing the two vertices of the edge. The vertex names range from zero to 1393382, inclusive.

Write code in any programming language and use it to solve the following problems. Attach your source code to the end of the assignment (PDF format is fine).

Part a (2 points) Read in and store the graph, e.g. as an adjacency list.[2] What is the largest degree of any vertex? What is the average degree?

Solution. Largest 12, Average 2.758.

For this problem depending on how your read the input, you may get twice this as an answer, which is ok.

Part b (4 points) (Theory problem.) Describe an algorithm that, given an undirected, unweighted graph, a vertex v, and an integer d, counts how many vertices are at distance d or less from v. The time and space complexity should only depend on the final answer, not on n or m. (Example: on input v and $d = 7$, suppose only 20 vertices are at distance 7 or less from v. Then the algorithm should only execute on the order of 20 iterations of any loop, even if n and m are over one million.)
Update: To make the problem easier, you may initially create a length-n array and initialize it, but the rest of the algorithm's running time and should only depend on the final answer.

[2] An adjacency matrix is *not* recommended, unless you have 15 TB free disk space and a lot of free time!

Solution. There are several ways to adapt BFS to solve this. The simplest is to initialize and run the shortest-paths-in-unweighted-graphs version, where we maintain a distance from s for each vertex. Start a count at zero and increase it by one each time we pop and process a vertex in the BFS loop. Because BFS processes vertices in order of distance, once we pop any vertex with distance set to $d+1$, we simply stop the program and return. The total number of iterations of the loop is equal[3] to the number of vertices at distance d.

Note one can avoid using full space by keeping the distance or "marked" variables in a smarter data structure than an array, e.g. a hash table. (This also avoids the initialization time needed to zero an array, although it's operating-system dependent.)

Part c (4 points) Implement your algorithm and run it on the Texas road network, picking 10 random initial vertices. For each, report its index (name) and how many vertices are at distance: 5 or less; 10 or less; and 50 or less. (Can your code handle 500?)

Solution. For example (node, ≤ 5, ≤ 10, ≤ 50):

```
Vertex 1375264: 50, 213, 9085
Vertex 949855: 46, 167, 13112
Vertex 41645: 25, 165, 13288
Vertex 298032: 34, 138, 11498
Vertex 1269367: 52, 252, 11883
Vertex 517652: 19, 136, 13303
Vertex 1097722: 34, 173, 16292
Vertex 366362: 26, 218, 9315
Vertex 893081: 31, 164, 13660
Vertex 257226: 28, 100, 15932
```

Example psuedocode below. In this code, we only process the neighbors of a vertex if it is at distance less than d. If it's at distance d, then we know all of its unprocessed neighbors are at distance $d+1$, so they can be skipped.

[3]There is a slight wrinkle in this problem if there are many vertices at distance $d+1$; we still add them all to the queue which takes time. You can avoid this and get a slightly better solution by also checking if a vertex has distance d and simply not processing its neighbors if this is the case.

Algorithm 1 BFS-Count

Input: Graph $G = (V, E)$, vertex s, distance d
Set dist$[u] = \infty$ for all $u = 1, \ldots, n$
Set dist$[s] = 0$
Set num_found $= 1$
Create queue Q $= \{s\}$
while Q is nonempty **do**
    Pop u from front of Q
    if dist$[u] < d$ **then**
        ProcessNeighbors(u)
    end if
end while
Return num_found

Subroutine 2 ProcessNeighbors(u)

for each neighbor v of u **do**
    if dist$[v] == \infty$ **then**
        Set dist$[v]$ = dist$[u] + 1$
        num_found $+ = 1$
        add v to back of Q
    end if
end for