Instructor:
Eleni Drinea
CSOR W4246–Fall, 2021

Student:
Vishweshwar Tyagi
vt2353@columbia.edu

Homework 2 Theoretical (85 points)

Out: Monday, October 4, 2021
Due: 11:59pm, Monday, October 18, 2021

Homework Problems

1. (20 points)

   (a) (10 points) The mayor of Algotopia gives you an updated map of the city of Algotopia where all streets are one-way. He wants you to determine whether there is a way to drive legally from any intersection in the city to any other intersection. Formulate this problem in graph-theoretic terms and design the most efficient algorithm you can to solve it.

   Solution (a) Here, intersections will be the nodes of our graph $G = (V, E)$ and a directed edge from a node $u \in V$ to a node $v \in V$, denoted as $(u, v) \in E$, would exist if and only if there is a one-way street from intersection u to v. We want to find out if we can drive from any intersection to any other intersection. This problem translates to finding whether or not our graph is strongly connected, i.e., whether or not it has only one strongly connected component (SCC). In case our graph is strongly connected, i.e, it has only one SCC, then this would mean that it is always possible to drive from any intersection to any other, otherwise not. We formulate this as follows:

   Problem Formulation: Given $G = (V, E)$ directed unweighted graph, we have to determine if our graph has only one strongly connected component

   We know that our graph will be strongly connected if and only if the strongly connected component of any node s is equal to the set of nodes V. Hence we choose any node $s \in V$ and calculate the strongly connected of s, denoted as $SCC(s)$.

   To get $SCC(s)$, we run BFS twice starting at s, first on graph G and then on reversed graph G^R. We keep track of nodes visited during each BFS with boolean node attributes $u.G$ and $u.G_R$ where these have usual meanings.

   BFS on G will give us all nodes reachable from s, and running it on G^R will give all nodes that can reach u (in G). We take the intersection of these two sets of nodes to get $SCC(s)$. Note that $SCC(s) \subseteq V$ and so $SCC(s) = V \iff |SCC(s)| = |V|$

   Thus, we return True if $\left|SCC(s)\right| = \left|V\right|$ in which case it is possible to drive from any intersection to any other, otherwise False.

pseudocode

```
1  procedure check-scc(G=(V, E), s):
2      for u in G.V:
3          u.G = False
4          u.G_R = False
5      end for
6
7      Run BFS on G starting at s and mark all visited nodes
8      // O(|V| + |E|)
9      // Now we have for all u in G.V
10     // u.G = True iff s can reach u, else False
11
12     Reverse G to get G_R     // O(|V| + |E|) with
13                              // G as adjacency list
14
15     Run BFS on G_R starting at s and mark all visited nodes
16     // O(|V| + |E|)
17     // Now we have for all u in G.V
18     // u.G_R = True iff u can reach s in G, else False
19
20     for u in G.V:
21         if !(u.G == True and u.G_R == True):
22             return False
23         end if
24     end for
25
26     return True
27
```

Correctness

We know that BFS correctly finds nodes reachable from s when ran on G and nodes that can reach s when ran on G_R. We return True if and only if $|SCC(s)| = |V|$, justification for which is given in idea above pseudocode and $[20 - 26]$ does exactly that.

Time-Complexity

Apart from time-complexities already mentioned in pseudocode, we have both $[2 - 5]$ and $[20 - 24]$ that run in $O(|V|)$ time, hence the time-complexity of our procedure is:

$$T(|V|, |E|) = O(|V| + |E|)$$

(b) (10 points) Suppose your answer to (a) is no. Your new task is to decide if the following statement holds: Starting from the town hall and navigating one-way streets, no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this problem in graph-theoretic terms and design the most efficient algorithm you can to solve it.

Solution (b) Here, our graph will be same as in (a), intersections will be nodes and edge (u, v) will exist if there is a one-way street from u to v. Let s = town hall. Now, it would be possible to legally drive back to town hall no matter where you reach when starting at town hall if and only if s can only reach those nodes that are in its strongly connected component $SCC(s)$, i.e., s is in a sink component.

Problem Formulation: Given $G = (V, E)$ directed unweighted graph and node s, we have to determine if s is in a sink component.

The idea again is to run BFS twice (on G and G_R), but this time starting at s = town hall. This will give us nodes reachable from s and nodes that can reach s. Now, we simply check if all nodes that are reachable from s can also reach s. If this holds, then s is in a sink component as it is not possible to leave this component and we return True, otherwise it is not possible and we return False.

pseudocode

```
procedure check-scc(G=(V, E), s):
    for u in G.V:
        u.G = False
        u.G_R = False
    end for

    Run BFS on G starting at s and mark all visited nodes
    // O(|V| + |E|)
    // Now we have for all u in G.V
    // u.G = True iff s can reach u, else False

    Reverse G to get G_R    // O(|V| + |E|) with
                            // G as adjacency list

```

```
    Run BFS on G_R starting at s and mark all visited nodes
    // O(|V| + |E|)
    // Now we have for all u in G.V
    // u.G_R = True iff u can reach s in G, else False

    for u in G.V:
        if u.G == True and u.G_R == False:
            return False
        end if
    end for

    return True

```

Correctness

We know that BFS correctly finds nodes reachable from s when ran on G and nodes that can reach s when ran on G_R. We return True if there is no node that can't reach s but is reachable from s and False otherwise. $[6-10]$ does exactly that. The justification for this is given in idea above pseudocode. Hence, our procedure is correct.

Time-Complexity

Apart from time-complexities already mentioned in pseudocode, we have both $[2-5]$ and $[6-10]$ that run in $O(|V|)$ time, hence the time-complexity of our procedure is:

$$T(|V|, |E|) = O(|V| + |E|)$$

2. (20 points) Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, give an $O(n + m)$ algorithm that computes the number of shortest $s - t$ paths in G.

Solution: Idea is to run BFS starting at s twice. At first, we mark all the nodes with their shortest distance from s. The second time, we count the number of shortest paths from s for each node while processing them in increasing order of their level in BFS tree.

pseudocode

```
procedure bfs-shortest-paths(G=(V, E), s, t):
    Run bfs on G starting starting at node s
    // Now each node u in G.E contains shortest
    // distance from s, saved in u.d

    for u in G.V:
        u.paths = 0     // no. of shortest paths to
                        // this node from s
        u.visited = False
    end for

    Initialize FIFO queue, call it Q
    s.paths = 1, s.visited = True
    Enqueue(Q, s)

    while Q not empty:
        u = Pop(Q)

        for (u, v) in G.E:
            if v.visited = False:
                v.visited = True
                Enqueue(Q, v)
            end if

            if v.d == u.d + 1:
                v.paths += u.paths
            end if

        end for
    end while

    return t.paths

```

Correctness

The proof of correctness follows from the correctness of BFS and the fact that in our second go, whenever we process a given node u, we increase the number of paths for all of its neighbours that have shortest paths from s with second last vertex as u.

Since each node that is reachable from s is enqueued and processed exactly once, we're only counting distinct shortest paths (from s) for each node. Also, because we process nodes based on their increasing level in BFS tree (due to correctness of BFS), we are not missing any shortest path to a node u from s. Hence, our procedure is correct.

Time-Complexity

We're simply using BFS twice, with some $O(1)$ time additional steps in the second variant. Hence, the time complexity is:

$$T(|V|, |E|) = O(|V| + |E|)$$

3. (20 points) In cases where there are several shortest paths between two nodes (and edges have varying lengths), the most convenient among these paths is often the one with the fewest edges. We define

$$\text{best}[u] = \text{minimum number of edges in a shortest path from } s \text{ to } v$$

Give an efficient algorithm that, on input a directed weighted graph $G = (V, E, w)$ with positive edge weights, and an origin node s, computes best$[u]$ for all $u \in V$.

Solution: The idea is to use Dijkstra's Algorithm again but this time we also compare the number of edges on the shortest path. In Dijkstra's algorithm, we always relax an edge (u, v) with weight $w(u, v)$ whenever $v.d > u.d + w(u, v)$ in which case we set $v.d = u.d + w(u, v)$

Now, in addition to the distance of a current shortest path from source node s to a node u, which is saved in $u.d$, we also store the minimum number of edges on such a path in u.edges

Now, whenever we have $v.d > u.d + w(u, v)$, we also set the minimum number of edges as $v.\text{edges} = u.\text{edges} + 1$

But not only this, if in case we encounter the case where $v.d = u.d + w(u, v)$, we always check if $v.\text{edges} \leq u.\text{edges} + 1$, because if not, then we have found a current shortest path with lesser number of edges, in which case we again set $v.\text{edges} = u.\text{edges} + 1$.

We give the pseudocode below:

pseudocode

```
procedure min-edge-Dijkstra(G=(V, E, w), s):
    for u in G.V:
        u.d = infty          // distance from s
        u.edges = infty      // edges on shortest path
        u.pi = NIL           // parent node

    // Initialize a PriorityQueue that returns node u with
    // minimum u.d and uses u.edges (lesser first) to break ties
    Q = MakePriorityQueue(G.V)          // O(|V|)

    s.d = 0, s.edges = 0
    Enqueue(Q, s)

    while Q not empty:
        u = ExtractMin(Q)        // O(log|V|)
        for (u, v) in G.E:
            if v.d > u.d + w(u, v):
                v.d = u.d + w(u, v)
                v.edges = u.edges + 1
            else if v.d == u.d + w(u, v):
                if v.edges > u.edges + 1:
                    v.edges = u.edges + 1
                end if
            end if
            ChangePriority(Q, v)         // O(log|V|)
        end for
    end while

    return V
```

Correctness

At every iteration of the while loop, we always remove one node from Q and hence it must halt after exactly $|V|$ iterations. At every iteration, we only update edges of a node v when we either find a shorter path to this node or when we find a path of same length but fewer edges and in both cases we set equal v.edges to one plus the edges of the second last node on this path. Because Dijkstra's algorithm is correct, we must, for each node, discover all shortest paths from s and from above argument, we correctly update the edges. Hence, our procedure is correct. A formal proof is given below

Suppose u_i is extracted by ExtractMin(Q) on the i^{th} iteration, then we claim that:

Claim: During i^{th} iteration of the while loop when ExtractMin(Q) outputs u_i, we must have u_i.edges = best$[u_i]$. Note that we already have u_i.d $= d(s, u_i)$, the shortest distance between s and u_i which follows from Dijkstra's algorithm.

We use induction on i to prove our claim:

Base: For $i = 1$, ExtractMin(Q) outputs $u_1 = s$, the only node with finite distance attribute. Clearly, u_1.edges = best$[u] = 0$

Assume our claim holds true when the while loop runs for the first k times (strong induction). We want to show that when it runs for the $(k+1)^{\text{th}}$, i.e., when ExtractMin(Q) outputs u_{k+1}, we must have u_{k+1}.edges = best$[u_{k+1}]$

Suppose otherwise. Then we must have u_{k+1}.edges $>$ best$[u_{k+1}]$. This is because, we relax edges outgoing from a node only after it has been extracted, and our assumption says that such a node $u \in \{u_1, \dots u_k\}$ has u.edges = best$[u]$. Now, since Dijkstra's algorithm is correct, we also have $u.d = d(s, u)$. So, the last time u_{k+1}.edges gets updated, it can only be an overestimate for best$[u_{k+1}]$ if not equal.

Now, u_{k+1}.edges $>$ best$[u_{k+1}]$ can't be possible either at $(k+1)^{\text{th}}$ iteration. This is because, we are always breaking ties between shortest paths from s to u_{k+1} using u_{k+1}.edges and this causes u_{k+1}.edges to be the minimum number of edges on a shortest path from s to u_{k+1}, which is simply best$[u_{k+1}]$.

Once a node is extracted by ExtractMin(Q), its attributes are never altered. Hence, our procedure returns all nodes with their 'best' distances from s saved in 'edges' attribute.

Time-Complexity:

Since we're only appending Dijkstra's algorithm by $O(1)$ time steps, the time complexity of our procedure is same as Dijkstra's algorithm

$$T(|V|, |E|) = O(\log |V| \big(|V| + |E|\big))$$

4. (25 points) You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. Your destination is the final hotel (at distance a_n) and you must stop there.

   You'd like to travel 200 miles a day, but this may not be possible, depending on the spacing of the hotels. If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty —that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the total penalty of an optimal sequence of hotels at which to stop, and returns such an optimal sequence.

Solution: I tried using the natural choice for a save move: Reach the next farthest post which is closest to 200 miles from current post, i.e, we go from post a_i to a_j where $j > i$ such that $|200 - (a_j - a_i)|$ is minimized and a_j is the farthest post where this happens (there can be at most two posts where $|200-(a_j-a_i)|$ is minimum, we choose the farthest one). Unfortunately, a counter-example to this strategy was readily available: $[a_0 = 0, a_1 = 150, a_2 = 190, a_3 = 265]$ where it is better to go directly to take the path $a_0 - a_1 - a_3$ instead of $a_0 - a_2 - a_3$

The next approach is to use recursion and make use of dynamic programming to have constant time access to solutions for overlapping sub-problems.

Let $a_0 = 0$, starting point.

Let $C(i, j)$ be the penalty accrued when we travel directly from a_i to a_j for all $0 \leq i < j \leq n$ We have,

$$C(i, j) = (200 - (a_j - a_i))^2 \tag{1}$$

Sub-problem: We define C_i to be the total penalty to get to post a_i starting at a_0, for all $0 \leq i \leq n$.

Recurrence: Clearly, $C_0 = 0$. Moreover, it is easy to see that

$$C_j = \min_{0 \leq i < j} \{C_i + C(i, j)\} \quad \forall\, 1 \leq j \leq n \tag{2}$$

because the penalty when reaching a_j starting at a_0 is the sum of two penalties: penalty accrued when reaching the last post before reaching a_j and the penalty accrued when travelling directly from that post to a_j. Since we're interested in minimum penalty, we minimize this sum over each prior post that we can visit just before visiting a_j. Since we're minimizing over a finite set of values, C_j exists and is is well defined, but the only thing to take care of is to calculate C_j in order of increasing j.

Boundary Condition: We know for sure that $C_0 = 0$ because no penalty is accrued when we are given $n = 0$ posts and we don't have to travel. This is our boundary condition.

In order to also calculate the optimal sequence of stops, we define

$$\text{stop}_j = \arg \min_{0 \leq i < j} \{C_i + C(i, j)\} \quad \forall\, 1 \leq j \leq n \tag{3}$$

which gives index of the post last visited before visiting a_j. We give the pseudocode below using (1), (2) and (3):

**note: to keep pseudocode clean and understandable, I'm using 0−indexing for this one.

pseudocode

```
1  procedure optimal-cost(S=[a_0, a_1, ... a_n]):
2      C_0 = 0
3
4      for j from 1 to n:
5          set C_j as in (2)                // O(n)
6          set stop_j as in (3)             // O(n)
7      end for
8
9      Initialize FIFO Queue, call it Q
10
11     (post, idx) = (S[n], n)
12     while post != 0:
13         Enqueue(Q, post)
14         post = S[stop_idx]
15         idx = stop_idx
16     end while
17
18     return C_n, Q
```

Correctness:

Claim: After j^{th} iteration, the loop $[4-7]$ correctly stores (in C_j) the minimum penalty accrued when reaching a_j from a_0 as well as it correctly stores (in stop_j) the index of the second-last post on a path with this minimum penalty C_j

We prove this via induction on j

Base: $j = 1$. During the first iteration, it sets $C_1 = C_0 + C(0,1) = C(0,1)$ and $\text{stop}_1 = 0$ which is correct as minimum cost of reaching p_1 from a_0 is $C(0,1)$ and the second-last post on an optimal path would have to be a_0

Assume, the loop satisfies the claim for all iterations $< j$ (Strong Induction). We shall show that it also satisfies the claim for j^{th} iteration.

But there is nothing to prove here since we know that (2) correctly calculates the minimum cost of reaching a_j from a_0 and uses only $\{C_i \mid 0 \leq i < j\}$ which are already computed correctly by assumption and similarly for (3).

Actually, there was no need of formal proof of correctness as (2) and (3) are intuitively correct. The only thing Dynamic Programming does is that it allows us to solve sub-problems only once and we're able to access solutions to these sub-problems in constant time.

Time-Complexity:

Note that $C(i, j)$ in (1) takes $O(1)$ time to calculate, and hence both C_j and stop_j in (1), (2) respectively take $O(n)$ time.

Since $[4 - 7]$ encloses calculations for C_j and stop_j, it takes $O(n^2)$ time.

$[12 - 16]$ is $O(n)$, rest are $O(1)$.

Hence, if $T(n)$ denotes the time-complexity of our procedure, then,

$$T(n) = O(n^2)$$

Extra-Space Complexity:

We need to store $C_0, C_1 \dots C_n$ and $\text{stop}_0, \dots \text{stop}_n$ so that $C_0, \dots C_{k-1}$ and $\text{stop}_0, \dots \text{stop}_{k-1}$ are accessible in $O(1)$ time when computing C_k and stop_k, for $1 \leq k \leq n$. These can be stored in $O(n)$ size array for this purpose.

Additionally, we also need queue Q which is a subset of $\{S[\text{stop}_1] \dots S[\text{stop}_n], a_n\}$ and again requires $O(n)$ extra-space.

Hence, total extra-space required is $O(n)$

Remarks:

- $[x - y]$ or $[x, y]$ refers to code lines starting at x and ending at y for referenced procedure
- $\{x, y, z\}$ refers to code lines x, y and z for referenced procedure and so forth.
- Unless mentioned otherwise, 1-indexing has been used for most data structures.
- Algorithms studied in lectures have been used directly.