CS 170 HW 8

Due **2021-10-25, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer “Yes”, “Yes but anonymously”, or “No”

2 (Max-Flow) A cohort of spies

A cohort of k spies resident in a certain country needs escape routes in case of an emergency. They will be travelling using the railway system which we can think of as a directed graph $G = (V, E)$ with V being the cities. Each spy i has a starting point $s_i \in V$, all s_i's are distinct. Every spy needs to reach the consulate of a friendly nation; these consulates are in a known set of cities $T \subseteq V$. In order to move undetected, the spies agree that at most c of them should ever pass through any one city. Our goal is to find a set of paths, one for each of the spies (or detect that the requirements cannot be met).

Model this problem as a flow network. Specify the vertices, edges and capacities, and show that a maximum flow in your network can be transformed into an optimal solution for the original problem. You do not need to explain how to solve the max-flow instance itself.

Solution: We can think of each spy i as a unit of flow that we want to move from s_i to any vertex $\in T$. To do so, we can model the graph as a flow network by setting the capacity of each edge to ∞, adding a new vertex t and adding an edge (t', t) for each $t' \in T$. We can add a source s and edges of capacity 1 from s to s_i. By doing so, we restrict the maximum flow to be k.

Lastly, we need to ensure that no more than c spies are in a city. We want to add vertex capacities of c to each vertex. To implement it, we do the following: for each vertex v that we want add constraint to, we create 2 vertices v_{in} and v_{out}. v_{in} has all the incoming edges of v and v_{out} has all the outcoming edges. We also put a directed edge from v_{in} to v_{out} with edge capacity constraint c.

If the max flow is indeed k, then as every capacity is an integer, the Ford-Fulkerson algorithm for computing max flow will output an integral flow (one with all flows being integers). Therefore, we can incrementally starting at each spy i follow a path in the flow from s_i to any vertex in T. We iterate through all of spies to reach a solution.

3 (Max-Flow LP) Min Cost Flow

In the max flow problem, we just wanted to see how much flow we could send between a source and a sink. But in general, we would like to model the fact that shipping flow takes money. More precisely, we are given a directed graph G with source s, sink t, costs l_e, capacities

c_e, and a flow value F. We want to find a nonnegative flow f with minimum cost, that is $\sum_e l_e f_e$, that respects the capacities and ships F units of flow from s to t.

(a) Show that the minimum cost flow problem can be solved in polynomial time.

(b) Show that the shortest path problem can be solved using the minimum cost flow problem

(c) Show that the maximum flow problem can be solved using the minimum cost flow problem.

Solution:

(a) Write min-cost flow as a linear program. One formulation is as follows:

$$\begin{aligned} &\min \sum_e l_e f_e \\ &\text{subject to } 0 \le f_e \le c_e && \forall e \in E \\ &\sum_{e \text{ incoming to } v} f_e = \sum_{e \text{ outgoing from } v} f_e && \forall v \in V, v \neq s,t \\ F + &\sum_{e \text{ incoming to } s} f_e = \sum_{e \text{ outgoing from } s} f_e \end{aligned}$$

(b) Consider a shortest path instance on a graph G with start s, end t, and edge weights l_e. Let $F = 1$ and let $c_e = 1$ for all edges e. We will now show that in the min cost flow f, the subgraph H of G consisting of edges with nonzero flow, all directed paths from s to t have length equal to the shortest path from s to t. Therefore, to find a shortest path from s to t, it is enough to use a DFS or BFS to find any path between s to t in H.

Suppose that each $l_e > 0$ (we can contract any edge with $l_e = 0$). First, we show that H is a DAG. If H is not a DAG, it must contain a cycle. Let $\delta > 0$ be the minimum flow along any edge of this cycle. Subtracting δ from the flow of all edges in this cycle results in a flow that still ships 1 unit of flow from s to t. This flow, though, has smaller cost than before, a contradiction to the fact that f is a min-cost flow. Therefore, H is a DAG.

Consider any path P between s and t in H. Let δ be the minimum flow on any edge of this path. One can write the flow f as a linear combination of paths including P by writing

$$f = \delta p + \delta_1 q_1 + \ldots + \delta_k q_k$$

where p and q_i are the unit flows along the paths P and Q_i respectively, δ_i is the minimum amount of flow on some arbitrary path Q_i after subtracting flow from $P, Q_1, Q_2, \ldots, Q_{i-1}$. Since subtracting δ_i units of flow decreased the flow from s to t by δ_i units, $\delta + \sum_i \delta_i = 1$. Since the objective function for min-cost flow is linear, we can write

$$\sum_e l_e f_e = \delta \sum_e l_e p_e + \sum_i \delta_i \sum_e l_e q_{ie} = \delta \sum_{e \in P} l_e + \sum_i \delta_i \sum_{e \in P_i} l_e$$

In particular, the cost of the flow f is the average length of the paths $P, Q_1, \ldots, Q_k$ according to the weights $\delta, \delta_1, \ldots, \delta_k$. Therefore, if P or some Q_i is not a shortest path from s to t, one can decrease the cost of f by reassigning the flow along that path to a shortest path between s and t. This means that if f has minimum cost, P and all Q_is must be shortest paths between s and t. Since P was chosen to be an arbitrary path from s to t in H, all paths in H must be shortest paths between s and t.

(c) We can use binary search to find the true max flow value. Consider a max flow instance on a graph G with capacities c_e where we wish to ship flow from s to t. Create a min-cost flow instance as follows. Set the capacities to be equal to c_e and let the lengths be arbitrary (say $l_e = 1$ for all edges). We know that the max flow value $F_{max} \leq \sum_e c_e$. If the capacities are integers, F_{max} is also an integer. Therefore, we can binary search to find its true value. For an arbitrary F, we can find out if there is a flow that ships more than F units of flow by querying our min-cost flow instance with value at least F. If there is a flow with value at least F, it will return a flow with finite cost. Otherwise, the program is infeasible.

4 Faster Maximum Flow

In the class, we see that the Ford-Fulkerson algorithm computes the maximum flow in $O(mF)$ time, where F is the max flow value. The run-time can be very high for large F. In this question, we explore a polynomial-time algorithm whose run-time does not the depend on the flow value. Recall that Ford-Fulkerson provides a general recipe of designing max flow algorithm, based on the idea of an *augmenting path* in a residual graph:

Algorithm 1 Ford-Fulkerson

```
while there exists an augmenting path in $G_f$ do
    Find an arbitrary augmenting path $P$ from $s$ to $t$
    Augment flow $f$ along $P$
    Update $G_f$
```

This problem asks you to consider a specific implementation of the algorithm above, where each iteration we find the augmenting path with the smallest number of edges and augment along it.

Algorithm 2 Fast Max Flow

```
while there exists an augmenting path in $G_f$ do
    Find the augmenting path $P$ from $s$ to $t$ with the smallest number of edges
    Augment flow $f$ along $P$
    Update $G_f$
```

We first show that each iteration can be implemented efficiently. Then we analyze the number of iterations required to terminate. Throughout, we define the shortest path from u to v as the path from u to v with the smallest number of edges (instead of the smallest sum of edge capacities). Consequently, we define distance $d(u, v)$ from u to v as the number of edges in the shortest path from u to v.

(a) Show that given G_f, the augmenting path P from s to t with the smallest number of edges can be found in $O(m + n)$ time.
Hint: Use the most basic graph algorithm you know.

(b) Show that the distance from s to v in G_f never decreases throughout the algorithm, for any v (including t).
Hint: Consider what augmentation does to the "forward edges" going from u to u', where $d(s, u') = d(s, u) + 1$.

(c) Show that with every m flow augmentations, the distance from s to t must increase by (at least) 1.
Hint: Observe that each flow augmentation removes at least one forward edge.

(d) Conclude by proving that the total number of flow augmentations this algorithm performs is at most $O(mn)$. Analyze the total run-time of the algorithm.
Hint: You may observe that the distance from s to t can increase at most $O(n)$ times throughout the algorithm. If you use this fact, explain why it holds.

Solution: *Bibliographical note*: This algorithm is due to Jack Edmonds and Richard Karp in 1972. In the same paper, they also considered the "fattest augmenting path" algorithm, namely, pushing along the augmenting path that allows the most increase in the flow value. The last excercise of the textbook asks you to analyze it.

(a) Recall that in unweighted graph, BFS suffices to find the shortest path, and it runs in time $O(m + n)$. Specifically, run BFS starting from s. The level where t appears in the BFS tree is the distance from s to t, and the shortest path is simply the corresponding tree path.

(b) We consider what a flow augmentation does to the edges in G_f. Consider the residual graph before the augmentation. Define the *rth layer* of the graph as $L_r = \{v : d(s, v) = r\}$, the set of nodes of distance r from s. We call an edge (u, v) forward if $u \in L_r$ and $v \in L_{r+1}$ for some r, and backward otherwise. Observe that no edge can skip a layer (check!).

Since we push along the path found by BFS, an elementary property of BFS is that its traversal contains only forward edges. Therefore, the augmentation would only create reverse edges going from L_{r+1} to L_r. In particular, it does not create any shortcut edge that skips at least a layer—namely, edge going from L_r to $L_{r'}$, for $r' \geq r + 2$. Therefore, $d(s, v)$ does not decrease.

(c) Note that each flow augmentation removes at least one forward edge. Since there are only m edges overall, this means that after k augmentations for some $k \leq m$ we must either reach a point when t is disconnected from s or a point where the path from s to t

has a back edge. In the first case, we are done. In the second case, this path has to be longer than the path in the beginning because it has to have at least one edge more than the shortest path of forward edges we had initially. Therefore, every m augmentations increase the distance from s to t by at least 1.

(d) Before the algorithm terminates, s, t are connected in G_f; that is, $d(s,t) < \infty$. (If it's not the case, we know that there is no more augmenting path from s to t, and the algorithm would terminate by simply outputting the current flow f.) Note that the maximum distance is $n-1$, since there are n nodes in total. Therefore, before the algorithm terminates, $d(s,t)$ can only increment (by 1) for $O(n)$ times. Each incrementation may take at most $O(m)$ augmentations by (c), and each augmentation costs $O(m)$ time via BFS, as shown in (a). Hence, the total runtime of the algorithm is $O(m^2n)$.

5 Jumbled Sequences

You are measuring temperatures in a few different places, and receiving temperatures remotely. You have received a sequence of n temperatures, from a total of k thermometers; unfortunately, they are all jumbled together, and you don't know which temperatures are from which thermometer.

Thankfully, you don't really care, and so are content to simply extract k disjoint 'plausible' subsequences from your original sequence of measurements (disjoint meaning each element in the original sequence belongs to at most 1 subsequence). A subsequence is 'plausible' if it satisfies 3 conditions:

1. it consists of elements from the original sequence in their original order
2. any two consecutive elements in the subsequence have no more than $2k$ elements between them
3. the first element in the subsequence has no more than $2k$ elements before it in the original sequence, and the last element has no more than $2k$ elements after it
4. any two consecutive elements in the subsequence differ in value by no more than 2

As an example, if you receive the sequence $(10, 8, 3, 9, 4, 9, 5, 6)$, you can extract 2 disjoint plausible subsequences from it: $(10, 8, 9, 9)$ and $(3, 4, 5, 6)$. Note that $(10, 8, 6)$ is not a plausible subsequence, because there are 5 elements between 8 and 6 in the original sequence.

Give an algorithm which, given a *specific* k, determines in $O(nk^2)$ time whether it is possible to find k disjoint plausible subsequences.

Describe your algorithm and give a runtime analysis; no proof of correctness is necessary.

Solution: Algorithm description

Represent the original sequence as a graph, with one 'in' node and one 'out' for each element, plus one 'start' node s and one 'end' node t. For every element, add an edge from its 'in' node to its 'out' node. For every two elements, add an edge from the 'out' node of the earlier element to the 'in' node of the later element when those elements satisfy the 'consecutive element' conditions for subsequences. Add an edge from the 'start' node to the 'in' node of every element within $2k$ elements of the beginning of the sequence, and from the 'out' node of every element within $2k$ elements of the end of the sequence to the 'end' node.

Assign every node in this graph a capacity of 1, and compute the maximum flow from s to t; it is possible to find k disjoint plausible subsequences if and only if the value of the maximum flow is k or greater. We can use Ford-Fulkerson, and stop once our flow reaches k.

Runtime

Since each element can be followed in a subsequence by one of at most $2k$ others, there are at most $O(nk)$ edges in the graph we create. Since each edge has capacity 1, we can run the Ford-Fulkerson algorithm for k iterations and it will augment the flow by 1 if it has not already found the maximum flow. Each of these iterations takes $O(|E|) = O(nk)$ time, and there are k iterations, so our algorithm as a whole takes at most $O(nk^2)$ time. If it is not possible to find k disjoint plausible subsequences, the algorithm will stop earlier, taking less time.