Compsci 330 Design and Analysis of Algorithms
Assignment 6, Spring 2023 Duke University

William Huang

Due Date: Tuesday, April 4, 2023

Directions

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. If you use another editor for your solutions (such as Microsoft Word), you should convert the final document to a pdf, confirm that the symbolic math from the equation editor is properly formatted, and submit the pdf. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English and pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work; do not skip details but do not write paragraphs where a sentence suffices.

Collaboration. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on gradescope. See the course policies webpage for more details on the collaboration and homework policies.

Problems

The first problem is an example with a solution provided. Problem 2 is a theory problem for you to solve, and problem 3 is an applied problem. We only have one theory problem, since problem 2 is closely related to problem 3. *All problems will be included in your final grade for this assignment.*

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1(Example). Given a directed flow network $G = (V, E)$ and any pair of vertices u and v, let $\mathsf{bottleneck}_G(u, v)$ denote the maximum, over all paths π in G from u to v, of the minimum-capacity edge along π. Describe an algorithm that computes $\mathsf{bottleneck}_G(s, t)$ in $O(|E| \log |V|)$ time. Prove the correctness of your algorithm and analyze its running time.

Solution 1 (Example). Let c be the capacity function, *i.e.* the capacity of edge e is $c(e)$.

Algorithm. The algorithm computes a maximum spanning tree of G, with capacity function as the weight of an edge, using the algorithm described in the class (Prim's algorithm). We grow a directed tree T rooted at s, one vertex at a time, by repeatedly adding the highest-capacity edge leaving T , until T contains a path from s to t. Rigorously, let V_T be the current set of vertices in T and E_T be the current set of edges in T. We initialize V_T by $\{s\}$ and initialize E_T by $\emptyset$. In each iteration, we find the edge $e = (u, v)$ with the highest capacity such that $u \in V_T$ and $v \in V \setminus V_T$. Then, include v into V_T and include e into E_T. In the following, we argue that $\mathsf{bottleneck}_G(s, t) = \mathsf{bottleneck}_T(s, t)$, whereas $\mathsf{bottleneck}_T(s, t)$ can be computed easily since there is only one path from s to t in T.

Proof of Correctness. We argue that $\mathsf{bottleneck}_G(s, v) = \mathsf{bottleneck}_T(s, v)$ for all $v \in V_T$. Suppose the sequence in which the vertices are included into V_T is $v_1, \ldots, v_k$, where $k = |V_T|$, $v_1 = s$, and $v_k = t$, *i.e.* in the i^{th} iteration, v_i is included into V_T. We apply induction on i.

The base case is when $i = 1$, in which case we have $V_T = \{s\}$ and $E_T = \emptyset$, so clearly $\mathsf{bottleneck}_G(s, v) = \mathsf{bottleneck}_T(s, v)$ for all $v \in V_T$. For the inductive step, suppose the claim holds for all $j \leq i$, and we show that the claim holds for $j = i + 1$ as well. Let T be the constructed tree after i iterations, and T' be the constructed tree after $i + 1$ iterations. Along the path from s to v_{i+1} in T, suppose the edge with the minimum capacity is $a \to b$. We will show that $\mathsf{bottleneck}_G(s, v_{i+1}) = \mathsf{bottleneck}_{T'}(s, v_{i+1})$. Suppose for the sake of contradiction that there exists a path from s to v_{i+1} in G whose minimum-capacity edge has higher capacity than $\mathsf{bottleneck}_{T'}(s, v_{i+1}) = c(a \to b)$. Take a prefix of this path $x_1 \to x_2 \to \cdots \to x_\ell$ such that x_ℓ is the first vertex along this path that is not in V_T. We consider the following cases:

- If $b = v_{i+1}$, then we clearly have contradiction, because we are taking the highest-capacity edge leaving T in each iteration, but $c(x_{\ell-1} \to x_\ell) > c(a \to b)$.
- If $b \neq v_{i+1}$, then we have $a, b \in V_T$. We consider what happens after a is added into the constructed tree: since for each $i \in \{1, \ldots, \ell - 1\}$, $c(x_i \to x_{i+1}) > c(a \to b)$, so $x_1, \ldots, x_\ell$ would be added into the tree before b. However, $x_\ell \notin V_T$ by assumption, which leads to contradiction.

Therefore, we have $\mathsf{bottleneck}_G(s, v_{i+1}) = \mathsf{bottleneck}_T(s, v_{i+1})$ as well, finishing the proof.

Analysis of Running Time. We use a binary heap to extract the heaviest edge leaving T at each step. We initially store all the edges in a heap, which takes $O(|E| \log |E|)$ time. In each iteration, finding the highest-capacity edge (u, v) with $v \notin V_T$ leaving T takes $O(\log |E|)$ time, and we delete all edges incident on v whose other end point is already in T, in $O(\log |E|)$ time per edge. The algorithm runs for at most $\min(|V| - 1, |E|)$ iterations (since each iteration adds in a new vertex and a new edge to T) and deletes at most $|E|$ edges. As $O(\log |E|) = O(\log |V|)$, the total running time is $O(|E| \log |V|)$.

Remark. One could also modify Dijkstra's algorithm to solve this problem in the same running time.

Problem 2. Given a directed network $G = (V, E)$ with integer capacity function $c : E \to \mathbb{Z}$ and maximum (s, t)-flow $f^* : E \to \mathbb{R}^+$ for some $s, t \in V$. Describe an algorithm for the following:

(a) Find an edge e in G, if there exists one, such that increasing the capacity of e by 1 increases the value of the maximum (s, t)-flow in G.

(b) Find an edge e in G, if there exists one, such that decreasing the capacity of e by 1 reduces the value of the maximum (s, t)-flow in G.

Prove the correctness of your algorithm and analyze its running time. Your algorithm should run in $O(|V| + |E|)$ time. You can assume you have direct access to the residual graph.

Algorithm a: Since we have direct access to the residual graph we can assume we can find a min-cut of the graph by taking all the nodes reachable on the residual graph from s to create set S and all other nodes to create set T, and creating a min-cut from all the edges that connect the two sets together. Because we are given the maximum (s, t)-flow $f*$, we can assume the residual network doesn't have any more augmenting paths, because if there did exist an augmenting path it would increase the max flow and change the residual network. All of those edges create the cut set, all of which act as the bottleneck of the graph. Where we can possibly increase the max-flow by 1. We can find this cut set with BFS. An increase in any of these edges would increase the max flow if an augmenting path exists, so Ford-Fulkerson must be run to see if there is an augmenting path that exists once one of the edges is increased. If Ford-Fulkerson finds an augmenting path, it will increase the edge weight to capacity, and we will have found the edge, if not, it does not exist.

Correctness: We can directly proof this with what we know about max-flow min-cut theorem and the Ford-Fulkerson algorithm. We know we have been given the max flow $f*$, so the residual graph, that we have direct access to, does not have any augmenting paths. And we know that there also exists a min-cut/cut set that connects the set of nodes reachable from s on the residual graph, and all other nodes. We know that by the max-flow min-cut theorem that this cut set contains the maximum flow of the graph, which makes it the bottleneck of the graph. Meaning increasing capacity in one of these edges can possibly increase the maximum flow of the graph, as long as an augmenting path exists after the increase. Using what we know about the Ford-Fulkerson algorithm if there exists an augmenting path, it will be found and increased on that single iteration. Otherwise, such an edge does not exist.

Runtime: We find the cut set with BFS which is $O(|V| + |E|)$. We then check for an augmenting path with Ford-Fulkerson, which worst-case is bounded by $O(|E|)$. So our overall runtime is $O(|V| + |E|)$.

Algorithm b: Using the residual graph, which has no augmenting path, because we've already been given the max flow $f*$, we can quickly identify the edges that are at max capacity. If there is an edge below capacity, decreasing it by 1 will not change the max flow, so we search the residual graph we have direct access to. We can use BFS on the residual graph to find a path from s to t. Decrease the edge weight at each edge in the path by 1, which will decrease the total flow by 1. If we run a single iteration of Ford-Fulkerson, if there exists an augmenting path on the path we decreased, the Ford-Fulkerson algorithm increases that path to the maximum, which means that we can decrease any edge on that path with maximized capacity to decrease maximum flow. If it does not find an augmenting path, then such an edge does not exist.

Correctness: Similar direct proof to part a, we are already given a residual graph with no augmenting paths (see part a), and we know trivially, decreasing the capacity of an edge that is not at full capacity will not change flow, so we only need to search through residual graph where all the flow is maxed. We use the graph to find a path from s to t and decrease each edge on that path by one, to decrease the total flow on that path by 1. If we run one iteration Ford-Fulkerson, and there exists an augmenting path, Ford-Fulkerson will find it, and return the total flow on that path increased by 1. We now can decrease the capacity of any edge on that path with maximized capacity and guarantee it will decrease flow. If such a path doesn't exist, then such an edge does not exist.

Runtime: We use BFS to search for the path s to t which is $O(|V|+|E|)$ runtime. We then use one iteration of Ford-Fulkerson which is bounded by the number of edges, so worst-case is $O(|E|)$. This gives an overall runtime of $O(|V| + |E|)$.

Problem 3 (Applied). You are a city planner trying to optimize traffic flow in the city's transportation network. Imagine a city with a complex network of roads and highways but with only one entry point s and one exit point t. You are asked to increase the traffic capacity driving from s to t but are only allotted with the money to widen one road such that its capacity would increase.

Assume now the city road map is converted into a directed graph (not necessarily acyclic) with nodes labeled as integers and with non-negative integer capacities on the edges. Your function will take in two parallel lists edges and capacities, where each edge is a tuple. For example, $(2, 4)$ is an edge that goes from starting node 2 to destination node 4. You are also given the label of a source vertex and a target vertex.

We say that an edge is a *priority edge* for routing traffic flow from source s to target t if increasing the capacity on that edge by 1 (with no other changes to the graph) would increase the value of the maximum flow from s to t.

Given the above inputs, **you should design and implement an algorithm that returns a list of *all* priority edges in the graph (or an empty list if there are none)**. The list can be in any order. For full credit, your solution will need to have an empirical runtime that is within constant factors of an $\mathcal{O}(nm^2)$ reference solution where n is the number of vertices and m is the number of edges.

Hint: recall the correspondence between the maximum value of the (s,t)-flow and the minimum value of the (s,t)-cut.

Language-specific details follow. You can use whichever of Python or Java you prefer. Your submission will be automatically graded against test cases for correctness and efficiency. You can resubmit as many times as you like up to the deadline.

- **Python.** You should submit a file called `flow.py` to the Gradescope item "Assignment 6 - Applied (Python)." The file should define (at least) a top level function `find_edges` that looks like:
  - `def find_edges(edges:[(u:int,v:int)], capacities:[int], s:int, t:int)`

  and returns a list of tuples `(u,v)` that are priority edges or an empty list `[]`
- **Java.** You should submit a file called `Flow.java` to the Gradescope item "Assignment 6 - Applied (Java)." The file should define (at least) a top level function `findEdges` that looks like:
  - `public List<int[]> findEdges(int[][] edges, int[] capacities, int s, int t)`

  where `edges` is a 2D array where `edges[i]` is an edge from `edges[i][0]` to `edges[i][1]` and `capacities[i]` is the capacitiy of `edges[i]`. Return either a list of edges or an empty list `[]`