Compsci 330 Design and Analysis of Algorithms
Assignment 8, Spring 2024 Duke University

Example Solutions

Due Date: Thursday, March 28, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

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. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any 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 or 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 and Internet. 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. You can use the internet, but looking up solutions or using large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[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 (Grid Games). Suppose we are given an $n \times n$ grid of squares. Each square is black or white. The input is represented as a two dimensional array W, where $W[i][j] = 1$ if the row i column j square is white, or 0 otherwise.

Describe an algorithm to determine whether tokens can be placed on white squares of the grid so that every row and every column contains exactly one token. Analyze the runtime complexity of the algorithm and briefly explain its correctness (at most a few sentences, not a formal proof).

You may use and assume the correctness of the Ford-Fulkerson algorithm as described in lecture, including using BFS to identify augmenting paths as in the Edmonds-Karp implementation, without restatement.

Hint. Try to reduce this problem to computing the maximum flow on a graph you define.

Solution 1. Construct a graph as follows:

- Add a source vertex s and a sink vertex t
- Add a row vertex r_i for every i from 1 to n. For each r_i, add a directed edge with capacity 1 from s to r_i.
- Add a column vertex c_j for every j from 1 to n. For each c_j, add a directed edge with capacity 1 from c_j to t.
- Add a vertex v_{ij} for every row i and column j such that $W[i][j] = 1$. For each v_{ij} add a directed edge from r_i to v_{ij} and another directed edge from v_{ij} to c_j, each with capacity 1.

Compute a maximum flow f^* on this graph using the Ford-Fulkerson algorithm (finding augmenting paths in the residual network with, for example, breadth-first search). If $|f^*| < n$, report that no feasible assignment exists. Else, a valid solution places a token on every row i column j entry such that a unit of flow passes through v_{ij} in f^*.

Correctness. First, note that because all capacities are 1, the Ford-Fulkerson algorithm will compute a flow that assigns either 1 or 0 units of flow to every edge. Also, observe that $|f^*| \leq n$ given that $\{s\}$ is an example of a cut with capacity n.

If $|f^*| = n$, then 1 unit of flow must pass from s to every r_i to some v_{ij}. Placing tokens on cells corresponding to all such v_{ij} thus covers every row. Similarly, from the v_{ij} nodes, 1 unit of flow must pass to every c_j, so placing the same tokens covers every column. Also, at most one unit of flow can pass from a given r_i to any v_{ij} and to c_j from any v_{ij}, due to the capacity constraints from s to r_i and from c_j to t, so we place a token on at most one cell per row or column. Conversely, if there is a feasible placement of tokens, the following corresponds to a flow of value n: For each row i, column j on which a token is placed, push 1 unit of flow from s to r_i to v_{ij} to c_j to t.

Runtime. The graph we construct has at most $n^2 + 2n + 2$ is $O(n^2)$ vertices and at most $2n^2 + 2n$ is $O(n^2)$ edges, so we can construct the graph in $O(n^2)$ time. Straightforwardly applying the runtime bound of the Edmunds-Karp algorithm (that is, Ford-Fulkerson using BFS to identify augmenting paths) would lead to a runtime bound of $O(n^2(n^2)^2)$ or $O(n^6)$.

However, we can give a tighter bound. We can find an augmenting path in $O(n^2)$ time with, for example, a single BFS in the residual network. As each augmentation in this graph adds 1 flow and a max flow has value at most n (note that the cut $\{s\}$, for example, has capacity n), this can happen at most n times. So the overall runtime will actually be $O(n(n^2))$ or $O(n^3)$.

Problem 2 (Flow Cycles). Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges. Let s, t be two distinct vertices in G, and let $c : E \to \mathbb{Z}_{\geq 0}$ be an edge capacity function such that each edge capacity is a non-negative integer, i.e., $c(u \to v) \geq 0$ is the (integer) capacity of the edge $u \to v$. Let $f : E \to \mathbb{Z}_{\geq 0}$ be a (s, t)-flow function in G in which one of the edges $v \to s \in E$ entering the source vertex s has $f(v \to s) = 1$.

(a) Prove that there must exist another (s, t)-flow $f' : E \to \mathbb{Z}_{\geq 0}$ with $f'(v \to s) = 0$ and $|f| = |f'|$ (that is, having the same (s, t)-flow value).

Solution 2a. We argue that there must be a cycle $C = s \to u_1, u_1 \to u_2, \ldots, u_k \to v, v \to s$ where $v \to s$ is the edge with $f(v \to s) = 1$ such that $f(e) \geq 1$ for every edge in C. Since $f(v \to s) = 1$ and flows are nonnegative integers, the flow conservation constraint on v implies that there must be an edge $u_k \to v$ with $f(u_k \to v) \geq 1$. The same reasoning holds for u_k, then for u_{k-1}, and so forth inductively, until we reach the start vertex s that does not have a flow conservation constraint.

Given that C exists, consider the flow f' defined as

$$f'(e) = \begin{cases} f(e) - 1 & \text{if } e \in C \\ f(e) & \text{otherwise.} \end{cases}$$

Note that f' has $f'(v \to s) = 0$ since $f(v \to s) = 1$ and $v \to s \in C$. We also argue that f' is also a valid flow. $f'(e) \leq f(e)\ \forall e \in E$ so f' cannot violate capacity constraints if f was a valid flow. Furthermore, for any $u \in C$, the net flow into and out of u are both reduced by 1, whereas for $u \notin C$ neither in flow nor out flow are changed. In either case, the flow conservation constraints at f' are still satisfied since they were satisfied for f. Finally, note that by the same logic the net flow out of s has not changed, so $|f'| = |f|$.

(b) Given f, describe an $O(m)$ runtime algorithm to compute f'. Briefly explain why the algorithm is correct, referencing the proof of existence from the previous part. Analyze the runtime of your algorithm.

Solution 2b. Run a single depth-first search (DFS) starting from s and only considering edges $u \to v$ that have $f(u \to v) \geq 1$. Record the DFS tree. As argued previously, there must be a cycle C beginning from s, which can be identified by a back edge to s in the DFS tree rooted at s. Compute f' by enumerating the edges and setting $f'(e) = f(e) - 1$ for all $e \in C$, and $f'(e) = f(e)$ otherwise.

The runtime complexity for the DFS on a subset of the edges is $O(|E|)$. The backedge and cycle can be identified from the DFS tree in $O(|E|)$ time. Finally, computing f' from f requires enumerating the edges, $O(|E|)$ time.

You may use and assume the correctness of the Ford-Fulkerson algorithm as described in lecture, including using BFS to identify augmenting paths as in the Edmonds-Karp implementation, without restatement.

Problem 3 (Disconnected). Let $G = (V, E)$ be an undirected connected graph with $n = |V|$ vertices and $m = |E|$ edges. Describe an $O(n^2 m)$ runtime algorithm to compute the minimum number of edges that must be removed from G in order to make it disconnected, i.e., after the removal of the edges, there exist two vertices in G with no path between them. Briefly explain why the algorithm is correct. Analyze the runtime complexity of the algorithm.

You may use and assume the correctness of the Ford-Fulkerson algorithm as described in lecture, including using BFS to identify augmenting paths as in the Edmonds-Karp implementation, without restatement.

Hint. What is the relationship between disconnecting the graph in this problem and the concept of a minimum cut?

Solution 3. First direct the graph by, for example, adding two directed edges $u \to v$ and $v \to u$ for every undirected edge $(u, v) \in E$. Then run the following algorithm that fixes an arbitrary $s \in V$ and then computes the max (s, t)-flow for every $t \in V \setminus \{s\}$ with the capacity of each edge in the graph set to 1. It then returns the minimum of all of those max flows.

```
1: procedure CUT(G = (V, E))
2:     Set the capacity of each edge to 1
3:     Fix an arbitrary s ∈ V
4:     c = ∞
5:     for t in V − {s} do
6:         Compute the max (s, t)-flow f* in G with Edmonds-Karp
7:         c = min{c, |f*|}
8:     return c
```

Correctness. Let c be the value returned by the algorithm, and let E^* be a minimum set of edges whose removal disconnects G.

Since c is the value of a minimum (s, t)-cut for some $t \in V \setminus \{s\}$, $c \geq |E^*|$, so it suffices to prove that $c \leq |E^*|$.

Let $A \subset V$ be the set of vertices in the connected component of $G' = (V, E \setminus E^*)$ that contains s. Since G' is disconnected, $V \setminus A$ is nonempty.

Let t be a vertex in $V \setminus A$. Then E^* is an (s, t)-cut. Since the algorithm computes the maximum (s, t')-flow for every $t' \in V \setminus \{s\}$ and returns the minimum value among them, c is bounded by the value of the minimum (s, t)-cut and thus $c \leq |E^*|$. Hence, we conclude that the algorithm returns the minimum number of edges that need to be deleted to disconnect G.

Runtime. The algorithm runs Edmonds-Karp (Ford-Fulkerson identifying augmenting paths with a breadth-first search) $O(n)$ times. Each such run takes $O(m|f^*|)$ time, where $|f^*|$ is the max flow between the specified vertices. In this case, G is a simple graph and all edge capacities are 1, so the flow is upper-bounded by $n - 1$, the maximum number of edges that can extend from the source. Thus, $|f^*|$ is $O(n)$, and the cumulative runtime is $O(n^2 m)$.

Problem 4 (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. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline. **NOTE:** Unlike the theory problems, the applied problem grade **is the raw score shown on Gradescope**. See the course assignments webpage for more details.

- **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 `[]`

Solution 4. We do not provide exact code implementations, but we sketch the idea of the algorithm below.

Initialize $E^* \leftarrow \emptyset$. Then:

- Compute the maximum $s - t$ flow f in G once using the Edmonds-Karp algorithm (Ford-Fulkerson with BFS to find augmenting paths). Let the resulting residual graph computed

as part of the algorithm be G_f.

- For every saturated edge e independently, increase the capacity of e by one, add the forward edge e in G_f with residual capacity 1, and run a BFS from s to determine whether G_f contains an augmenting path. If incrementing the capacity of e creates an augmenting path, add e to E^*. After each search, decrement the capacity of e again and remove the forward edge from G_f, continuing this procedure for every saturated edge. Finally, return E^*.

The runtime of Edmonds-Karp is $O(|V||E|^2)$, and the runtime of the BFS after incrementing the capacity of every saturated edge is $O(|E|(|V| + |E|))$. Thus, the total runtime is dominated by the initial computation of the max flow, which is $O(|V||E|^2)$.