E0 225 Design & Analysis of Algorithms **Assignment - 3**

Sai Phanindra Korlepara — **24142** — M.Tech CSA Dept.

Note : Hand in has a slight printing error, was unable to get a better print. Please excuse

---

Problem 1. *KT Book, Chapter 7, Problem 5.*
Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

Let G be an arbitrary flow network, with a source s, a sink t, and a positive integer capacity c_e on every edge e; and let (A, B) be a minimum s-t cut with respect to these capacities $\{c_e : e \in E\}$. Now suppose we add 1 to every capacity; then (A, B) is still a minimum s-t cut with respect to these new capacities $\{1 + c_e : e \in E\}$.

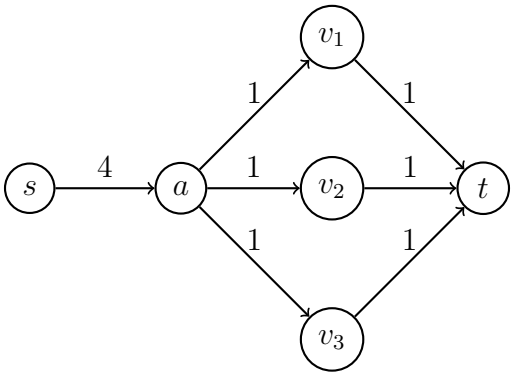
Solution. We are asked whether increasing each edge capacity by 1 in a flow network G will preserve the minimality of a given minimum s-t cut (A, B). Specifically, does (A, B) remain a minimum s-t cut after the capacities are increased to $c'_e = c_e + 1$ for all $e \in E$?

Answer:

The statement is **false**. Increasing each capacity by 1 can change the minimum s-t cut; thus, (A, B) may no longer be a minimum cut in the modified network.

Counterexample:

Consider the following flow network:



Original Minimum Cut:

The minimum s-t cut in the original network is (A, B), where $A = \{s, a, v_1, v_2, v_3\}$ and $B = \{t\}$. The capacity of this cut is:

$$C(A, B) = c(v_1, t) + c(v_2, t) + c(v_3, t) = 3.$$

Another Cut:

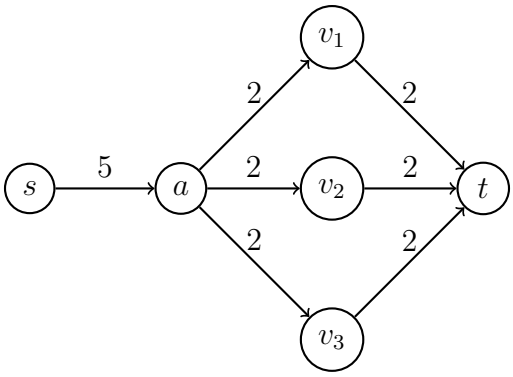
Consider another cut (A', B') where $A' = \{s\}$ and $B' = \{a, v_1, v_2, v_3, t\}$. The capacity of this cut is:

$$C(A', B') = c(s, a) = 4.$$

Thus, (A, B) is a minimum s-t cut and (A', B') is not a minimum s-t cut.

After Increasing Capacities:

Now, increase each capacity by 1:



Capacities of Cuts After Increase:

- Capacity of cut (A, B):

$$C'(A, B) = c(v_1, t) + c(v_2, t) + c(v_3, t) = 6.$$

- Capacity of cut (A', B'):

$$C'(A', B') = c(s, a) = 5.$$

As we can clearly see from the example above, by chaning weight of each of the edges by 1, The minimum cut has changed from (A, B) to (A', B').

□

Problem 2. *KT Book, Chapter 7, Problem 12.*
Consider the following problem. You are given a flow network with unit capacity edges: It consists of a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$; and $c_e = 1$ for every $e \in E$. You are also given a parameter k.

The goal is to delete k edges so as to reduce the maximum s-t flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum s-t flow in $G' = (V, E - F)$ is as small as possible subject to this.

Give a polynomial-time algorithm to solve this problem.

Solution. We are given a flow network with unit capacities on the edges and a task to delete k edges such that the maximum flow from s to t is reduced as much as possible. The key idea is to leverage the Max-Flow Min-Cut Theorem, which tells us that the maximum flow is equal to the capacity of the minimum cut in the network. If the minimum cut has fewer than k edges, we can reduce the flow to zero by removing all edges in the cut. If the minimum cut is larger than k, removing k edges will reduce the flow by exactly k units.
Algorithm

- Use any standard max-flow algorithm, such as Ford-Fulkerson or Edmonds-Karp, to compute the maximum flow from s to t and identify the corresponding minimum cut in G.

- Let C be the set of edges in the minimum cut. If $|C| \leq k$, return the set C (removing all edges in C reduces the flow to zero). If $|C| > k$, proceed to the next step.

- Remove any k edges from the set C. Since all edges have unit capacity, removing k edges reduces the maximum flow by exactly k units.

Complexity Analysis

Time Complexity: - Finding the max flow takes $O(|V| \cdot |E|^2)$ using the Edmonds-Karp algorithm. - Removing k edges takes $O(k)$. Thus, the overall time complexity is $O(|V| \cdot |E|^2)$.

Space Complexity: The space complexity is $O(|V| + |E|)$ to store the graph.

Proof of Correctness

By the Max-Flow Min-Cut Theorem, the maximum flow is equal to the minimum cut's capacity. Removing k edges from the minimum cut reduces the capacity and hence the maximum flow by exactly k units, as all edges have unit capacity. If $|C| \leq k$, removing all cut edges reduces the flow to zero. □

Problem 3. *JE Book, Chapter 10, Problem 13(a) and (b).*
For any flow network G and any vertices u and v, let bottleneck$_G(u, v)$ denote the maximum, over all paths in G from u to v, of the minimum-capacity edge along the path.

(a) Describe and analyze an algorithm to compute bottleneck$_G(s, t)$ in $O(E \log V)$ time. This is the amount of flow that the Edmonds-Karp fattest-augmenting-paths algorithm pushes in the first iteration.

(b) Now suppose the flow network G is undirected; equivalently, suppose $c(u, v) = c(v, u)$ for every pair of vertices u and v. Describe and analyze an algorithm to compute bottleneck$_G(s, t)$ in $O(V + E)$ time. [Hint: Find the median edge capacity.] Why doesn't this speedup work for directed graphs?

Solution.

a. We want to find bottleneck$_G(s, t)$, which is the maximum, over all paths from s to t, of the minimum-capacity edge along the path. This can be efficiently computed using a modified Dijkstra's algorithm with a priority queue, where the edge weights are treated as capacities, and we aim to maximize the minimum edge weight on each path.

Algorithm:

1. Initialize a priority queue with the source vertex s and set the bottleneck capacity of s to infinity.
2. For each vertex u, initialize bottleneck[u] to be 0, except for s
3. While the priority queue is not empty:
   (a) Extract the vertex u with the maximum bottleneck[u] from the priority queue.
   (b) For each neighbor v of u, compute the bottleneck capacity of v as min(bottleneck[u], $c(u, v)$), where $c(u, v)$ is the capacity of the edge from u to v.
   (c) If the computed bottleneck capacity is larger than the current bottleneck[v], update bottleneck[v] and insert v into the priority queue.

4. Return bottleneck[t] when t is extracted from the queue.

Time Complexity: Using a binary heap for the priority queue, the algorithm processes each edge in $O(\log V)$ time. Thus, the overall time complexity is $O(E \log V)$.

Correctness: The algorithm is similar to Dijkstra's algorithm, but instead of minimizing distances, we maximize the bottleneck capacity. Each vertex is processed in order of decreasing bottleneck capacity, ensuring that the maximum bottleneck capacity is found for each vertex.

b. Now, we consider the case where the flow network is undirected, i.e., $c(u, v) = c(v, u)$ for all vertices u and v. Here, we can exploit the fact that the edges are undirected to speed up the computation of the bottleneck capacity.

Algorithm:

1. Sort all edges by their capacities.
2. Perform a breadth-first search (BFS) to find a path from s to t. If no path is found, terminate and return 0.
3. Perform binary search on the sorted edge capacities to find the largest capacity for which there exists a path from s to t using only edges with capacity greater than or equal to the median capacity.
4. Return this capacity as $\text{bottleneck}_G(s, t)$.

Time Complexity:

Sorting the edges takes $O(E \log E)$ time. Each BFS takes $O(V + E)$ time.. The binary search on the edge capacities takes $O(\log E)$ iterations, and each iteration involves a BFS. Hence, the overall time complexity is $O((V + E) \log E)$,

□

Problem 4. *JE Book, Chapter 10, Problem 14.*
Suppose you are given a flow network G with integer edge capacities and an integer maximum flow f in G. Describe algorithms for the following operations:

(a) ***Increment(e)****: Increase the capacity of edge e by 1 and update the maximum flow.*

(b) ***Decrement(e)****: Decrease the capacity of edge e by 1 and update the maximum flow.*

Both algorithms should modify f so that it is still a maximum flow, more quickly than recomputing a maximum flow from scratch.

Solution. We are given a flow network G with integer edge capacities and a known maximum flow f. The problem asks us to describe algorithms for efficiently updating the maximum flow when the capacity of an edge is either incremented or decremented by 1, without recomputing the maximum flow from scratch.

a. Increment(e): Increase the capacity of edge e by 1

Algorithm:

1. Increase the capacity of edge $e = (u, v)$ by 1.
2. Check if there is an augmenting path from the source s to the sink t in the residual graph G_f after the increment. To do this, perform a breadth-first search (BFS) or depth-first search (DFS) on the residual graph to look for a path from s to t.
3. If an augmenting path exists, augment the flow along this path by the bottleneck capacity (which will be at least 1).
4. If no augmenting path is found, the maximum flow remains unchanged.

Time Complexity:

Finding an augmenting path takes $O(V + E)$ time using BFS or DFS. Augmenting the flow along the path takes $O(V + E)$ time. Thus, the total time complexity for the **Increment(e)** operation is $O(V + E)$.

b. Decrement(e): Decrease the capacity of edge e by 1

Algorithm:

1. Check if edge $e = (u, v)$ is currently fully utilized in the flow (i.e., if the flow along e is equal to its capacity). If not, simply decrease the capacity of edge e by 1 in the original graph and reduce the residual capacity of edge e in the residual graph by 1. No flow adjustment is needed.
2. If edge e is fully utilized:
   (a) Decrease the capacity of edge e by 1.
   (b) Reduce the flow along edge e by 1 and adjust the flow along the residual graph G_f to maintain flow conservation.
   (c) Find an alternate path from u to v in the residual graph G_f that can accommodate the lost flow by performing a BFS or DFS. If such a path exists:
   - Augment the flow along this path by 1.
   - Update the residual capacities along the path, adjusting both the forward and backward capacities in the residual graph to reflect the change.

   (d) If no alternate path is found:
   - Reduce the total maximum flow by 1.
   - Find a valid path from the source s to the sink t that includes edge $e = (u, v)$. Perform a BFS or DFS in the original flow graph to identify all such paths.
   - Choose the path with the smallest bottleneck capacity. Decrease the flow along this path by 1, and update both the flow graph and the residual graph accordingly to ensure flow conservation is maintained.

Time Complexity: Checking if an edge is fully utilized takes $O(1)$ time. Searching for an alternate path using BFS or DFS takes $O(V + E)$ time. Updating the residual graph takes $O(V + E)$ time as well. Thus, the overall time complexity for the **Decrement(e)** operation is $O(V + E)$.

□
