15-451/651 Algorithm Design & Analysis, Spring 2026

Homework #4 Solutions

1. (*) (min bottleneck spanning tree) Define the bottleneck cost of a spanning tree to be the maximum weight of an edge in the tree.

   (a) Show that a minimum spanning tree also minimizes the bottleneck cost among all spanning trees.

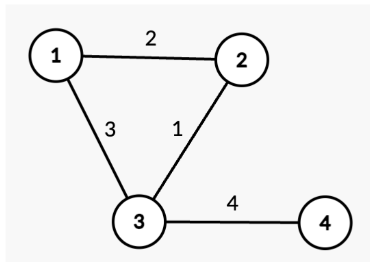
   (b) Give an example graph on at most 4 vertices where a tree that minimizes the bottleneck cost is not a minimum spanning tree (aka. does not minimize total cost).

   Solution: (a) Let A be an MST and let e^* be the maximum-weight edge in A. Suppose for contradiction there exists a spanning tree B whose bottleneck cost is strictly smaller than $w(e^*)$.

   Order edges increasingly by weight and run Kruskal's algorithm. Since every edge of B has weight $< w(e^*)$, all edges of B are considered before e^*. Because B is a spanning tree, its edges connect the graph. Thus Kruskal would already have formed a spanning tree before reaching e^*, so e^* would not be added — contradiction.

   Hence every MST minimizes the bottleneck cost.

   (b) Consider the following graph where there are two different spanning trees: one consisting of edges with weights $1,3,4$ and another with weights $1,2,4$.



   Both trees minimize the bottleneck cost. However, their total weights are different:

   $$1+3+4=8 \quad \text{and} \quad 1+2+4=7.$$

   Thus, a tree that minimizes the bottleneck cost need not minimize the total weight.

2. (**)

   n foxes are each trying to acquire a burrow and a smart phone, of which there are also n of each. However, each fox has a list of burrows and phones that they can use, and no phones or burrows should belong to multiple foxes.

   We want to maximize the number of happy foxes, where a happy fox has both a smartphone and burrow they can use.

   Formulate this problem graph theoretically, and give an $O(n^{10})$ time algorithm that solves that version.

   The technical description of your solution, once you create your graph, should not mention foxes, burrows, phones, or happiness :-(.

   Solution: We create a flow network with 3 "layers" (secretly 4 layers though). First, we create s our source vertex and t our sink vertex. Then our first "layer" will be nodes representing burrows, lets call them b_i. We create one node per burrow, and all will be connected to s with an edge of capacity 1. Our third layer (yes, I am describing it out of order) will represent phones, lets call each node p_i. Every phone will be connected to t with an edge of capacity 1.

Then our middle layer will represent foxes, lets call each node f_i. Note, that if we have two burrows b_i and b_j and 2 phones p_i and p_j that fox f_i can use, and we naively connect burrows to foxes that can use them and foxes to phones they can use, then a maximum flow can send multiple flow through just 1 fox node! 2 edges in of some capacity and two edges out of some capacity. We have to stop this as a fox should only use 1 burrow and 1 phone. What we use is a strategy called node splitting. For each fox we create two nodes. f_i and f_i'. We connect b_is to f_is and we connect f_i's to p_is. Then for all i we connect f_i to f_i' with an edge of capacity 1. Now, only 1 flow can pass through each fox node regardless of how many burrow edges send flow and how many phone edges can receive flow.

Complexity: We construct the flow network, which takes time $O(n^2)$ as we look through all pairs. Then, we actually compute the maximum flow. We can just use standard Ford-Fulkerson, which takes time $O(mF)$, where m is the number of edges in the graph, and F is the maximum flow. The number of edges is $O(n^2)$, and the maximum flow is every fox is satisfied, which is at most n. Thus, we have a runtime of $O(n^3)$

Correctness: We can show each assignment of foxes corresponds to a valid flow (*result of value k $\Rightarrow$ flow of value k*. This is straightforward, how we have constructed our network any assignment that can be made corresponds to a path in the graph from s to t.

We can then show each flow corresponds to a valid assignment (*flow of value k $\Rightarrow$ result of valuek*. This is the slightly more complicated case, but what we know is that a flow can be broken down into a list of unit-capacity paths from s to t. Now observe what we have done with the node-splitting. This guarantees that only 1 flow enters and leaves the middle layer. By conservation this implies that each unit-capacity path is disjoint, and can correspond to exactly 1 fox and a unique burrow and phone from any other path. This means our flow is valid.

Knowing that for any result a valid flow exists and that for any flow a corresponding result exists implies that the optimal result also corresponds to the optimal flow. Thus, the max flow is our answer.

3. (***) Consider undirected graphs where edges have **positive** costs c_e, and are numbered from 1 to m. For a sequence of m vertices $x_1 \ldots x_m$, we want to compute for each i, the minimum cost of reaching vertex 1 starting from x_i if:

   (a) first walk along any sequence edges numbered $1 \ldots i$ for free,

   (b) then take the rest of the edges, with costs given by c_e.

   Give an algorithm that takes such a graph with n vertices, $m > n$ edges, and computes these values in $O(m \log n)$ time.

   hint: process the edges in order using a variant of the union-find tree.

   Solution: First, run Dijkstra's algorithm to compute

   $$d_v := \text{the shortest-path distance from 1 to } v$$

   with respect to the original edge costs c_e.

   Then for each query vertex x_i, the answer is

   $$\min\{d_y \mid y \text{ is reachable from } x_i \text{ using only edges with index } \le i\}.$$

   To compute this efficiently, process the edges in increasing order $1, 2, \ldots, m$. Suppose edges $1, \ldots, i-1$ have already been processed, and edge i connects vertices u and v. Since edges with index $\le i$ are now allowed, we can treat u and v as connected via a zero-cost edge. Thus, we merge their connected components.

   We maintain a Union-Find data structure. For each connected component, we store the minimum value of d_v among all vertices in that component. When two components are merged, we update this minimum accordingly.

   Therefore, when processing x_i, the answer is simply the stored minimum d_v in the connected component containing x_i.