Homework 3

Winston (Hanting) Zhang

Friday, September 23, by 23:00

General Instructions. The following assignment is meant to be challenging, and we anticipate that it will take most of you at least 10–15 hours to complete, so please allow yourself plenty of time to work on it. (This assignment is probably somewhere between Homework 1 and Homework 2 in difficulty.) We highly recommend reading the entire assignment right away — you never know when inspiration will strike. Please provide a formal mathematical proof for all your claims, and present runtime guarantees for your algorithms using asymptotic (big-$O/\Omega/\Theta$) notation, unless stated otherwise. You may assume that all basic arithmetic operations (multiplication, subtraction, division, comparison, etc.) take constant time.

Collaboration. Please carefully check the collaboration policy on the course website. When in doubt, ask an instructor.

Consulting outside sources. Please carefully check the policy regarding outside sources on the course website. Again, when in doubt, ask an instructor.

Submission. Homework submission will be through the Gradescope system. Instructions and links have been provided through the course website and Piazza. The only accepted format is PDF. Starting with this homework, only typed solutions will be accepted, and we highly recommend producing your solutions using LaTeX (the text markup language we are also using for this assignment).

Recommended practice problems (do not hand in): KT Problems 4.1, 4.2, 4.19, 4.20, 4.22, 4.26 (for extra challenge).

Problem 1. [8+7=15]

Consider an undirected connected graph $G = (V, E)$ with edge costs $c_e > 0$ for $e \in E$ which are all distinct.

1. Let $E' \subseteq E$ be defined as the following set of edges: for each node v, E' contains the cheapest of all edges incident on v, i.e., the cheapest edge that has v as one of its endpoints. Is the graph (V, E') connected? Is it acyclic? For both questions, provide a proof or a counter-example with explanations.

   Proposition 1. *(V, E') is not always connected.*

   Proof. Counterexample: Intuitively, we can have a super high weight edge that no one picks, which splits the graph. Consider $V = \{w, x, y, z\}$ and $E = \{(w, x), (x, y), (y, z)\}$ with weights $\{1, 100, 1\}$, respectively. Then $\{w, x, y, z\}$ will all be incident to an edge of weight 1, making $E' = \{(w, x), (y, z)\}$, which is not connected. □

   Proposition 2. *If (V, E') has no self loops, then it is acyclic.*

   Proof. Assume for the sake of contradiction that E' has some cycle. Let $e = (v, w)$ be the edge in that cycle with the highest weight. Since e is in a cycle, v and w are incident to edges e_v and e_w in the cycle. Note that $w \neq w$, and $e_v, e_w \neq e$, since (V, E') has no self loops.

   We assumed that e has the highest weight, but this is contradictary as v both e_v and e_w are cheaper edges incident to v and w. Hence e could never be in E', contradiction. □

2. Consider the following outline for an algorithm, which starts with an empty set T of edges: Let E' contain the cheapest edge out of each connected component of (V, T). Add E' to T, and repeat until (V, T) is connected. Show that this algorithm outputs a minimum spanning tree of G, and can be implemented in time $O(m \log n)$.

Algorithm 1 MST Algorithm

Require: graph $G = (V, E)$
  for each vertex v in G **do**
    $k_v = v$, (k_v is the connected component of v)
  while (V, T) is not connected **do**
    let $E' \leftarrow$ set of minimum edges out of each connected component
    let $T \leftarrow T \cup E'$
    compute new connected components of (V, T)
  return (V, T)

Proposition 3. *Algorithm 1 terminates and produces a MST of G.*

Proof. First let's establish two facts we will need to use.

(a) Since G is a connected graph with distinct edge weights, there is a unique MST (V, T') of G.

(b) **Cut property**: Let $S \subseteq E$. The minimum edge between S and $E \setminus S$ is in T'.

At each iteration, the number of connected components of (V, T) strictly decreases. Hence we can have at most $|V|$ iterations before (V, T) has only one connected component, i.e. (V, T) is connected, and hence the algorithm must terminate.

Let (V, T) be the output of Algorithm 1; let (V, T') be the unique MST of G. At each iteration k, we produce some edge set E'_k. By design, each edge $e \in E'_k$ is the minimum edge between its connected component C and $E \setminus C$. By the cut property, $e \in T'$, thus $E'_k \subseteq T'$. This holds for each iteration k. We know that $T = \bigcup_k E'_k$, i.e. T is the total of all edges in each iteration, so we conclude that $T \subseteq T'$.

Furthermore, (V, T) is a connected subgraph of G. Hence $|T| \geq |V| - 1 = |T'|$, since (V, T') is a MST. Combining $T \subseteq T'$ and $|T| \geq |T'|$, it must be that $T = T'$. Hence (V, T) is indeed the MST of G. □

Proposition 4. *Algorithm 1 runs in $O(m \log n)$ time complexity.*

Proof. We claim that the outer while loop iterates at most $O(\log n)$ times, and that the inner body runs in $O(m)$, giving a total of $O(m \log n)$ runtime.

Outer while loop: At the end of the iteration, we end with new connected components which each have at least 2 of the connected components from the start of the iteration inside them. That is to say, at each iteration we at least halve the number of connected components. We begin with a total of n connected compoents, giving an upper bound of $O(\log n)$ iterations before (V, T) is connected.

Inner body: We claim that each of the 3 steps in the body can be implemented in $O(m)$, for a total runtime of $O(3m) = O(m)$.

1. Compute E' as follows: Initialize empty map m where each key-value pair (k, e) represents the current best edge e coming out of the connected component k. For each $e = (v, w) \in E$, let k_v and k_w be their components. (This can be implemented as a field of the node class for $O(1)$ access.) If $k_v = k_w$, then e is an internal edge, so discard e. Otherwise, update $m[k_v]$ and $m[k_w]$ if e is lower cost than the current best (again $O(1)$ if m vector or hashset). In the end, m maps each connected component to its best outgoing edge. Collect m into E'. This runs a for loop for $O(n)$, with $O(1)$ operations within each iteration, for a total of $O(n)$ to compute E'.
2. Implement T as a vector. Then $T \leftarrow T \cup E'$ is $O(|E'|) = O(m)$.
3. Compute the new connected components by running BFS/floodfill on (V, T). BFS runs in $O(m + n) = O(m)$.

Thus we have shown that each step runs in $O(m)$. This completes the proof. □

Hints:

- Each iteration of this algorithm can be viewed as applying the operation from part (a) on a "contracted graph", where each connected component of (V, T) corresponds to a node.
- If you want, you are welcome to use the efficient implementation of the Union-Find data structure which supports Find and Union in logarithmic time, as described in Chapter 4.6 of the KT book and briefly outlined in class. However, this implementation is not necessary, and focusing on it may mislead you from the basic idea.
- We recommend thinking about how many iterations it will take until (V, T) is connected.

Problem 2. [15]

We frequently motivate the shortest path problem, and Dijkstra's Algorithm for solving it, by considering transportation networks, such as driving from one place to another. Other transportation networks are rail or flight networks. Those are a little different from road networks, in that the edges (e.g., trains) are only available at certain times, rather than all the time with a given length.

Specifically, you are given a directed graph $G = (V, E)$ in which each directed edge $e = (u, v, t_u, t_v)$ corresponds to a train connection that goes from u to v, leaves at time t_u, and arrives at time t_v. Of course, there can now be multiple edges from u to v. You know that for each such edge, $t_v > t_u$, since the train cannot arrive before it leaves.[1] You know nothing else about the arrival and departure times; in particular, there could be two trains from u to v with $t'_u > t_u$ and $t'_v < t_v$. For example, these could be an express train and a scenic local train, where the express train leaves later and arrives earlier. You are also given a start node s, start time T, and destination node d. Your goal is to compute the earliest that you can arrive at d when starting at s at time T. You can assume that changing trains takes 0 time, so if you arrive at node u at time t, you can board a train that leaves u at time t.

Give and analyze (i.e., prove correct and analyze the running time) an algorithm that solves this problem in time $O(m \log n)$.

[1] We are talking about trains here, not DeLoreans.

Algorithm 2 Modified Dijkstra's Algorithm

Require: graph G, vertex *start*, vertex *target*, start time T
  let Q be a min-heap of vertices prioritized by earliest time
  for each vertex $v \neq$ *start* in G **do**
    earliest$[v] = \infty$
    Add v to Q
  Add *start* to Q
  let earliest[*start*] $\leftarrow T$
  while Q is not empty **do**
    let $v \leftarrow$ popmin Q
    if $v =$ target **then**
      return earliest[*target*]
    for each incident edge (v, w) with w still in Q **do**
      if $t_v \geq$ earliest$[v]$ **then**
        earliest$[w] \leftarrow \min(t_w,$ earliest$[w])$
  return $-\infty$

Proof. We do induction on the following invariant: At each iteration, when we pop v, dist$[v]$ is the shortest distance from *start* to v.

In the base case, all nodes have priority infinity except for *start*, so we must pop *start*. This trivially satisfies the invariant since earliest[*start*] $= 0$.

Now, assume the invariant is true till the k^{th} iteration. We set $v \leftarrow$ popmin Q and must prove the invariant is true on the $k + 1^{\text{th}}$ iteration. Split the neighbours of w into two cases: those not in the Q, N, and those still in Q, M. Consider the shortest path from *start* to w and let v be the node just before w.

If $v \in M$, then we know that earliest$[v] >$ earliest$[w]$, because w had the minimum priority when we popped it. Hence going *start* $\rightarrow v \rightarrow w$ is clearly worst than not going through v. Thus it cannot be the case that $vinM$.

If $v \in N$, let m_v be the minimum time it takes to get to v. Then the earliest you can get to w is the minimum of t_w over all edges (v, w) such that $t_v > m_v$. Note here m_v is the theorical best. However, by our induction hypothesis, earliest$[v] = m_v$! Since each $v \in N$ was already popped, the line 'earliest$[w] \leftarrow \min(t_w,$ earliest$[w])$' was computed for each v. In particular, this means exactly that earliest$[w]$ is the minimum of t_w over all edges (v, w) such that $t_v > m_v$! Hence our induction step is complete.

Thus, when we pop *end*, we must have earliest[*end*] be the optimal time.

□

Hint: Obviously, this question is closely related to Dijkstra's Algorithm, which we did not cover in class. We highly recommend that you review the algorithm and — more importantly — its analysis in Section 4.4 of the textbook before attempting this question.

Problem 3. [0]

Chocolate Problem: 2 chocolate bars

Reminder: If you solve a chocolate problem (which you can do in groups of size up to 3), please e-mail David with the solution — do not submit it on Gradescope. Also, feel free to list preferences or dietary restrictions for/against particular types of chocolate.

Exercise 4.31 in the textbook. Notice that Part (b) is really the interesting thing here — Part (a) is basically a slightly harder regular problem.