Problem Set 6 Solutions

Reading: Chapters 22, 24, and 25.

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered in the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct algorithms which are *which are described clearly*. Convoluted and obtuse descriptions will receive low marks.

Exercise 6-1. Do Exercise 22.2-5 on page 539 in CLRS.

Exercise 6-2. Do Exercise 22.4-3 on page 552 in CLRS.

Exercise 6-3. Do Exercise 22.5-7 on page 557 in CLRS.

Exercise 6-4. Do Exercise 24.1-3 on page 591 in CLRS.

Exercise 6-5. Do Exercise 24.3-2 on page 600 in CLRS.

Exercise 6-6. Do Exercise 24.4-8 on page 606 in CLRS.

Exercise 6-7. Do Exercise 25.2-6 on page 635 in CLRS.

Exercise 6-8. Do Exercise 25.3-5 on page 640 in CLRS.

Problem 6-1. Truckin'

Professor Almanac is consulting for a trucking company. Highways are modeled as a directed graph $G = (V, E)$ in which vertices represent cities and edges represent roads. The company is planning new routes from San Diego (vertex s) to Toledo (vertex t).

(a) It is very costly when a shipment is delayed en route. The company has calculated the probability $p(e) \in [0, 1]$ that a given road $e \in E$ will close without warning. Give an efficient algorithm for finding a route with the minimum probability of encountering a closed road. You should assume that all road closings are independent.

Solution:

To simplify the solution, we use the probability $q(e) = 1 - p(e)$ that a road will be open. Further, we remove from the graph roads with $p(e) = 1$, as they are guaranteed to be closed and will never be included in a meaningful solution. (Following this transformation, we can use depth first search to ensure that some path from s to t has a positive probability of being open.) By eliminating $p(e) = 1$, we now have $0 < q(e) \leq 1$ for all edges $e \in E$. It is important to have eliminated the possibility of $q(e) = 0$, because we will be taking the logarithm of this quantity later.

Because the road closings are independent, the probability that a given path will be open is the product of the probabilities of the edges being open. That is, for each path $r = e_1, e_2, \ldots, e_n$, the probability $Q(r)$ of the path being open is:

$$Q(r) = \prod_{i=1}^{n} q(e_i)$$

Our goal is to find the path r, beginning at s and ending at t, that maximizes $Q(r)$. Taking the negative logarithm of both sides yields:

$$\begin{aligned} -\lg Q(r) &= -\lg \prod_{i=1}^{n} q(e_i) \\ &= \sum_{i=1}^{n} -\lg q(e_i) \end{aligned}$$

Using $w(e_i)$ to denote the quantity $-\lg q(e_i)$, this becomes:

$$-\lg Q(r) = \sum_{i=1}^{n} w(e_i)$$

The right hand side is a sum of edge weights $w(e)$ along the path from s to t. We can minimize this quantity using Dijkstra's algorithm for single-source shortest paths. Doing so will yield the path r that minimizes $-\lg Q(r)$, thereby maximizing $Q(r)$. This path will have the maximum probability of being open, and thus the minimum probability of being closed.

The running time is $O(E+V \lg V)$. We only spend $\Theta(E)$ to remove edges with $p(e) = 1$ and to update the edge weights; Dijkstra's algorithm dominates with a runtime of $O(E + V \lg V)$.

Alternate Solution:

It is also possible to modify Dijkstra's algorithm to directly compute the path with the highest probability of being open. As above, let $q(e) = 1-p(e)$ denote the probability that a given road $e \in E$ will be open, and let $Q(r)$ denote the probability that a given path r will be open. For a given vertex v, let $o[v]$ denote the maximum value of $Q(r)$ over all paths r from s to v. Then, make the following modifications to Dijkstra's algorithm:

1. Change INITIALIZE-SINGLE-SOURCE to assign $o[s] = 1$ for the source vertex and $o[v] = 0$ for all other vertices:

   ```
   INITIALIZE-SINGLE-SOURCE$(G, s)$
   1  for each vertex $v \in V[G]$
   2       do $o[v] \leftarrow 0$
   3          $\pi[v] \leftarrow$ NIL
   4  $o[s] \leftarrow 1$
   ```

   That is, we can reach the source vertex with probability 1, and the probability of reaching all other vertices will increase monotonically from 0 using RELAX.

2. Instead of EXTRACT-MIN, use EXTRACT-MAX (and a supporting data structure) to see which vertex to visit. That is, first explore paths with the highest probability of being open.

3. Rewrite the RELAX step as follows:

   ```
   RELAX$(u, v, q)$
   1  if $o[v] < o[u] * q(e)$    $\triangleright$ where $e = (u, v)$
   2     then $o[v] \leftarrow o[u] * q(e)$
   3          $\pi[v] \leftarrow u$
   ```

   That is, if a vertex v can be reached with a higher probability than before along the edge under consideration, then increase the probability $o[v]$. Because the probabilities of roads being open are independent, the probability of a path being open is the product of the probabilities of each edge being open.

The argument for correctness parallels that of Dijkstra's algorithm, as presented in lecture. The proof relies on the following properties:

1. *Optimal substructure.* A sub-path of a path with the highest probability of being open must also be a path with the highest probability of being open. Otherwise we could increase the overall probability of being open by increasing the probability along this sub-path (cut-and-paste).

2. *Triangle inequality.* Let $\rho(u, v)$ denote the highest probability of a path from u to v being open. Then for all $u, v, x \in V$, we have $\rho(u, v) \geq \rho(u, x) \cdot \rho(x, v)$. Otherwise $\rho(u, v)$ could be increased if we chose the path through x.
3. *Well-definedness of shortest paths.* Since $q(e) \in [0, 1]$, the probability of a path being open can only decrease as extra edges are added to a path. Since we are exploring the path with the highest probability of being open, this ensures that there are no analogues of negative-weight cycles.

Properties (1) and (2) are true whenever an associative operator is used to combine edge weights into a path weight. In Dijkstra's shortest path algorithm, the operator is addition; here it is multiplication.

The running time is $O(E + V \lg V)$. We spend $\Theta(E)$ to calculate $q(e) = 1 - p(e)$, and Dijkstra's algorithm runs in $O(E + V \lg V)$.

(b) Many highways are off-limits for trucks that weigh more than a given threshold. For a given highway $e \in E$, let $w(e) \in \mathbb{R}^+$ denote the weight limit and let $l(e) \in \mathbb{R}^+$ denote the highway's length. Give an efficient algorithm that calculates: 1) the heaviest truck that can be sent from s to t, and 2) the shortest path this truck can take.

Solution:

First, we modify Dijkstra's algorithm to find the heaviest truck that can be sent from s to t. The weight limit $w(e)$ is used as the edge weight for e. There are three modifications to the algorithm:

1. In INITIALIZE-SINGLE-SOURCE, assign a value of ∞ to the source vertex and a value of 0 to all other vertices.
2. Instead of EXTRACT-MIN, use EXTRACT-MAX (and a supporting data structure) to see which vertex to visit. That is, first explore those paths which support the heaviest trucks.
3. In the RELAX step, use $\min$ in place of addition. That is, maintain the minimum weight limit encountered on a given path instead of the total path length from the source.

As in Part (a), the proof of correctness follows that of Dijkstra's algorithm. Since the $\min$ operator is associative, the optimal paths exhibit optimal substructure and support the triangle inequality. There are no analogues of negative-weight cycles because the weight supported by a path can only decrease as the path becomes longer (and we are searching for the heaviest weight possible).

Given the weight of the heaviest truck that can pass from s to t, we can find the shortest path as follows. Simply remove all edges from the graph that are less than the weight of the heaviest truck. Then, run Dijkstra's algorithm (unmodified) to find the shortest path.

The overall runtime of our algorithms is that of Dijkstra's algorithm: $O(E + V \lg V)$.

(c) Consider a variant of (b) in which trucks must make strictly eastward progress with each city they visit. Adjust your algorithm to exploit this property and analyze the runtime.

Solution:

Remove from the graph any edges that do not make eastward progress. Because we always go eastward, there are no cycles in this graph. Thus, we can use DAG-SHORTEST-PATHS to solve the problem in $\Theta(V + E)$ time. We need only modify the INITIALIZE-SINGLE-SOURCE and RELAX procedures as in (b).

Problem 6-2. Constructing Construction Schedules

Consider a set of n jobs to be completed during the construction of a new office building. For each $i \in \{1, 2, \ldots, n\}$, a ***schedule*** assigns a time $x_i \geq 0$ for job i to be started. There are some constraints on the schedule:

1. For each $i, j \in \{1, 2, \ldots, n\}$, we denote by $A[i, j] \in \mathbb{R}$ the ***minimum latency*** from the start of job i to the start of job j. For example, since it takes a day for concrete to dry, construction of the walls must begin at least one day after pouring the foundation. The constraint on the schedule is:

$$\forall\, i, j \in \{1, 2, \ldots, n\} : \quad x_i + A[i, j] \quad \leq \quad x_j \tag{1}$$

   If there is no minimum latency between jobs i and j, then $A[i, j] = -\infty$.

2. For each $i, j \in \{1, 2, \ldots, n\}$, we denote by $B[i, j] \in \mathbb{R}$ the ***maximum latency*** from the start of job i to the start of job j. For example, weatherproofing must be added no later than one week after an exterior wall is erected. The constraint on the schedule is:

$$\forall\, i, j \in \{1, 2, \ldots, n\} : \quad x_i + B[i, j] \quad \geq \quad x_j \tag{2}$$

   If there is no maximum latency between jobs i and j, then $B[i, j] = \infty$.

(a) Show how to model the latency constraints as a set of linear difference equations. That is, given $A[1\,..\,n, 1\,..\,n]$ and $B[1\,..\,n, 1\,..\,n]$, construct a matrix $C[1\,..\,n, 1\,..\,n]$ such that the following constraints are equivalent to Equations (1) and (2):

$$\forall\, i, j \in \{1, 2, \ldots n\} : \quad x_i - x_j \quad \leq \quad C[i, j] \tag{3}$$

Solution:

Re-arranging Equation (1) yields:

$$\forall\, i, j \in \{1, 2, \ldots, n\} : \quad x_i - x_j \quad \leq \quad -A[i, j] \tag{4}$$

Re-arranging Equation (2) yields:

$$\begin{aligned} \forall\, i,j \in \{1,2,\ldots,n\} : \quad x_i - x_j &\geq -B[i,j] \\ \forall\, i,j \in \{1,2,\ldots,n\} : \quad x_j - x_i &\leq B[i,j] \\ \forall\, i,j \in \{1,2,\ldots,n\} : \quad x_i - x_j &\leq B[j,i] \end{aligned} \tag{5}$$

Equations (4) and (5) are equivalent to Equation (3) if we set:

$$\forall\, i,j \in \{1,2,\ldots,n\} : \;\; C[i,j] = \min(-A[i,j], B[j,i])$$

(b) Show that the Bellman-Ford algorithm, when run on the constraint graph corresponding to Equation (3), minimizes the quantity $(\max\{x_i\} - \min\{x_i\})$ subject to Equation (3) and the constraint $x_i \leq 0$ for all x_i.

Solution:

Recall that the Bellman-Ford algorithm operates on a graph in which each constraint $x_j - x_i \leq C[i,j]$ is translated to an edge from vertex v_i to vertex v_j with weight $w_{ij} = C[i,j]$. Also, an extra vertex s is added with a 0-weight edge from s to each vertex $v \in V$. If Bellman-Ford detects a negative-weight cycle in this graph, then the constraints are unsatisfiable. We thus focus on the case in which there are no negative-weight cycles. The proof takes the form of a lemma and a theorem.

Lemma 1 *When Bellman-Ford is run on the constraint graph,* $\max\{x_i\} = 0$.

Proof. Let $p = \langle s, v_1, \ldots, v_k\rangle$ be the shortest path from vertex s to vertex v_k as reported by Bellman-Ford when run over the constraint graph. By the optimal substructure of shortest paths, $\langle s, v_1\rangle$ must be the shortest path from s to v_1. By construction, the edge from s to v_1 has a weight of 0. Thus $x_1 = \delta(s, v_1) = w(\langle s, v_1\rangle) = 0$. In combination with the constraint $x_i \leq 0$ for all x_i, this implies that $\max_i x_i = 0$. □

Theorem 2 *When Bellman-Ford is run on the constraint graph, it minimizes the quantity* $(\max\{x_i\} - \min\{x_i\})$

Proof. Since $\max\{x_i\} = 0$ (by Lemma 1), it suffices to show that Bellman-Ford maximizes $\min\{x_i\}$. Let $x_k = \min\{x_i\}$ in the solution produced by Bellman-Ford, and consider the shortest path $p = \langle v_0, v_1, \ldots, v_k\rangle$ from $s = v_0$ to v_k. The weight of this path is $w(p) = \sum_{i=0}^{k-1} w_{i(i+1)} = w(\langle v_0, v_1\rangle) + \sum_{i=1}^{k-1} C[i, i+1] = \sum_{i=1}^{k-1} C[i, i+1]$. The path corresponds to the following set of constraints:

$$\begin{aligned} x_1 - x_0 &\leq 0 \\ x_2 - x_1 &\leq C[1,2] \\ x_3 - x_2 &\leq C[2,3] \\ &\cdots \\ x_k - x_{k-1} &\leq C[k-1,k] \end{aligned}$$

Summing the constraints, we obtain:

$$\begin{aligned} x_k &\le \sum_{i=1}^{k-1} C[i, i+1] \\ &= w(p) \end{aligned}$$

That is, in any solution that satisfies the constraints, x_k cannot be greater than $w(p)$, the weight of the shortest path from s to v_k. As Bellman-Ford sets x_k to the shortest path value, this implies that x_k is as large as possible. $\square$

(c) Give an efficient algorithm for minimizing the overall duration of the construction schedule. That is, given $A[1\,..\,n, 1\,..\,n]$ and $B[1\,..\,n, 1\,..\,n]$, choose $\{x_1, x_2, \ldots, x_n\}$ so as to minimize $\max\{x_i\}$ subject to the latency constraints and the constraint $x_i \ge 0$ for all x_i. Assume that an unlimited number of jobs can be performed in parallel.

Solution:

The algorithm is as follows:

1. Construct a constraint graph from the constraints in Equation (3). However, if $C[i, j] = \infty$, then **do not add the edge** (v_j, v_i) to the graph.
2. Run Bellman-Ford on the constraint graph to obtain a solution $\{x_1, x_2, \ldots, x_n\}$.
3. Set $y = \min\{x_i\}$ and calculate a new solution $x'_i = x_i - y$ for all x_i. Output $\{x'_1, x'_2, \ldots, x'_n\}$.

This algorithm differs from Part (b) in two ways. First, to incorporate the constraint $x_i \ge 0$, we simply shift the solution x_i (in Step 3) so that all values are non-negative. This linear shift will not affect the feasibility of the difference constraints, as the difference between each pair of variables remains unchanged. Also, $\min\{x'_i\} = \min\{x_i\} - y = 0$, which (in combination with Theorem 2) implies that the algorithm minimizes $\max\{x_i\}$ subject to the constraints.

The second difference between the algorithm and Part (b) is in the construction of the constraint graph. In order to improve the runtime, we omit edges from the constraint graph that correspond to an infinite weight $C[i, j]$. Because these edge weights are infinite, they will not impact the shortest path found by Bellman-Ford.

The running time is $\Theta(V^2)$ for Step 1, $\Theta(VE)$ for Step 2, and $\Theta(V)$ for Step 3; the overall runtime is $\Theta(V^2 + VE)$. In terms of the scheduling problem, the running time is $\Theta(n^2 + nk)$, where k represents the number of (non-infinite) constraints between jobs. If each job has at least one constraint, then the runtime is $\Theta(nk)$. Though $k = \Theta(n^2)$ in the worst case, we would expect the number of constraints to be sparse in practice. Thus, the algorithm above is a significant improvement over the $\Theta(n^3)$ algorithm that would result from a naive construction of the constraint graph.

(d) If the constraints are infeasible, we'd like to supply the user with information to help in diagnosing the problem. Extend your algorithm from (c) so that, if the constraints are infeasible, your algorithm prints out a set S of conflicting constraints that is minimal—that is, if any constraint is dropped from S, the remaining constraints in S would be feasible.

Solution:

A simple algorithm is to run depth-first search on the constraint graph, starting from vertex s. Upon encountering a back edge, construct a cycle in the graph by tracing back through the predecessor matrix to the target of the back edge. Then print out the constraint corresponding to each edge in the cycle; these constraints form a set S as specified.

A set of linear difference constraints are conflicting if and only if they form a cycle in the constraint graph. Thus, identifying a single cycle in the constraint graph will indicate a set of conflicting constraints S. This set is minimal in that, with the removal of any edge, the cycle is broken and the remaining constraints in S are feasible.

The running time is $\Theta(n + k)$, where k is the number of (non-infinite) constraints between jobs. This runtime follows directly from the $\Theta(V + E)$ runtime of depth-first search on the constraint graph.

Alternate Solution:

It is also possible to utilize the predecessor matrix that is constructed as part of the Bellman-Ford algorithm. If the last pass of Bellman-Ford is modified to update the predecessor matrix for the vertices under consideration, then the infeasible constraints will be manifested as cycles in the predecessor graph. A cycle can be detected in the predecessor graph using depth first search. In this case, the runtime for the search is $\Theta(V) = \Theta(n)$, since each vertex has at most one incoming edge ($E = \Theta(V)$).

While this is arguably a more elegant solution, the asymptotic runtime is no better than the first solution when considered as an extension of Bellman-Ford. Bellman-Ford is $\Theta(nk)$, which dominates the overall runtime.

Note:

It is **not** a correct solution to simply trace back in the predecessor graph from the first edge that fails the test in Bellman-Ford. If there are multiple cycles in the graph, the first edge found might not be part of a cycle in the predecessor graph.

Problem 6-3. Honeymoon Hiking

Alice and Bob (after years of communicating in private) decide to get married and go hiking for their honeymoon. They obtain a map, which they naturally regard as an undirected graph $G = (V, E)$; vertices represent locations and edges represent trails. Some of the locations are bus stops, denoted by $S \subset V$.

Alice and Bob consider a hike to be ***romantic*** if it satisfies two criteria:

1. It begins and ends at different bus stops.
2. All of the uphill segments come at the beginning (and all of the downhill segments come at the end).[1]

Let $w(e) \in \mathbb{R}^+$ denote the length of a trail $e \in E$, and let $h(v) \in \mathbb{R}$ denote the elevation (height) of a location $v \in V$. You may assume that no two locations have exactly the same elevation.

(a) Give an efficient algorithm to find the *shortest* romantic hike for Alice and Bob (if any romantic hike exists).

Solution:

The algorithm is as follows:

1. Construct a directed acyclic graph $G_U = (V_U, E_U)$ that represents the uphill trail segments. The vertices in this graph are the same as the original: $V_U = V$. For each undirected edge in G, there is an edge in G_U that is directed uphill:

$$E_U = \{(u, v) : (u, v) \in E \text{ and } h(u) < h(v)\}$$

   Because the elevation of each vertex is distinct, one direction of each trail segment must be strictly uphill. This implies that G_U is acyclic, as it is impossible to form a loop while hiking strictly uphill.

2. For each bus stop $s \in S$, compute the shortest uphill path to all other vertices $v \in V$. As G_U is acyclic, this can be done by running DAG-SHORTEST-PATHS from each $s \in S$ over the graph G_U.
3. For each vertex $v \in V_U$, compute the two closest bus stops $s_1[v]$ and $s_2[v]$ from which one can reach v by hiking strictly uphill. This can be done by taking the two minima, over all vertices $s \in S$, of the shortest uphill path from s to v (as computed in (2)). (Note that if v is a bus stop, then one of these paths has zero length—that is, if $v \in S$ then $s_1[v] = v$ or $s_2[v] = v$.)
4. Observe that for every vertex v, there is a romantic hike $s_1[v] \rightsquigarrow v \rightsquigarrow s_2[v]$. Further, this is the shortest romantic hike that has v as its high point (i.e., where the hike transitions from uphill to downhill). Thus, the shortest romantic hike overall can be found by taking the minimum over all high points $v \in V$. Using $l[v]$ to denote the length of the path $s_1[v] \rightsquigarrow v \rightsquigarrow s_2[v]$ in G, the shortest hike is:

$$\text{Shortest} = \min_{v \in V} l[v]$$

[1]After all, what is more frustrating than going uphill after you have started going downhill?

The actual path of the shortest romantic hike can be recovered by keeping track of the vertex v that is selected as the minimum.

Correctness relies on the observation that every romantic hike will have a high point, where the hike turns from uphill to downhill (we assume that all elevations are distinct, so every trail is either strictly uphill or strictly downhill). The shortest romantic hike for a given high point will correspond to the two closest bus stops for which there are uphill paths from the bus stops to the high point. The hike itself will start at one of these stops, go uphill to the high point, and then downhill to the second stop. We compute the shortest overall romantic hike as the minimum over all possible high points.

Step 1 runs in $\Theta(V + E)$ time, as all edges and vertices must be copied over into the new graph. Step 2 runs in $\Theta(S(V + E))$ time, as DAG-SHORTEST-PATHS takes $\Theta(V + E)$ time to run on each vertex $s \in S$. Step 3 runs in $\Theta(SV)$ time, as it selects the smallest and second-smallest distances out of S bus stops, for each of V vertices. Step 4 runs in $\Theta(V)$ time, as it selects the minimum length over V points.

The overall runtime is thus $\Theta(SV + SE)$, as Step 2 dominates.

(b) Give an efficient algorithm to find the *longest* romantic hike for Alice and Bob (if any romantic hike exists).

Solution:

The same algorithm as in Part (a) applies, with minor modifications to compute the longest path instead of the shortest:

1. Each $\min$ operation from Part (a) is replaced with $\max$.
2. DAG-SHORTEST-PATHS is modified to compute longest paths instead of shortest paths. This involves modifying INITIALIZE-SINGLE-SOURCE to invert the roles of 0 and ∞ and modifying RELAX to use $\max$ instead of $\min$.

Correctness follows the same argument as in Part (a). The running time is identical.

Note that while this problem is very similar to (a), it does introduce the possibility of traversing an edge twice in the undirected graph. Some other approaches to (a) break down given this possibility.

Alternate Solution:

Use the same algorithm as in Part (a), but invert all of the edge weights on the original graph. That is, use edge weights $w'(e) = -w(e)$ for all edges $e \in E$.

As the algorithm in Part (a) breaks the graph into DAGs, there cannot be any negative-weight cycles (there are no cycles at all). Thus, DAG-SHORTEST-PATHS still gives the correct answer. Steps 3 and 4 remain unchanged in selecting the shortest composition of paths from G_D and G_U.

The runtime is identical to Part (a).