Sample Solutions to Problem set 5

1. (5 points) Chokepoints

Let $G = (V, E)$ be a directed graph and let $w : E \to \mathbb{R}$ be a weight function. For a path P from u to v we define the *chokepoint* of P to be the weight of an edge with smallest weight $\min_{e \in P}\{w(e)\}$. For two vertices u and v we define the *chokepoint* from u to v to be the maximum, over all paths P from u to v, of the chokepoint of P. (If there is no path from u to v, then we can define the chokepoint from u to v to be the undefined symbol $\perp$.)

Design an algorithm that takes as input G, w, and a source s, and computes the chokepoint from s to v for every vertex $v \in V$. Prove the correctness of your algorithm and analyze its running time.

Answer: We can modify Dijkstra's algorithm by mainly replacing the "min" operation by "max", the "+" operation by "min", and the ">" operation by "<". These entail three kinds of changes in the algorithm.

First, consider the initialization routine. We will set $d[v]$ to 0 for all $v \neq s$ and we set $d[s]$ to ∞.

Second, the operation in which we explore a given edge (u, v) will change to the following code.

1. **if** $d[v] < \min\{d[u], w(u, v)\}$
2. $\quad d[v] = \min\{d[u], w(u, v)\}$
3. $\quad \pi[v] = u$

Finally, in DIJKSTRA(G, w, s), instead of extracting the vertex with the minimum d value, we extract the vertex with the maximum d value.

The proof of Dijkstra's algorithm for shortest paths proceeds by showing two claims: (a) $d[u] \geq \delta(s, u)$ for all u at all times, and (b) at the time u is extracted from Q, $d[u] = \delta(s, u)$. For establishing these two claims, the two main properties that we need are (i) if the last edge on the shortest path from s to v is (u, v), then $\delta(s, v) = \delta(s, u) + w(u, v)$, and (ii) for any edge (u, v), $\delta(s, v) \leq \delta(s, u) + w(u, v)$ (triangle inequality).

In our present situation, we are seeking the path with the largest chokepoint – not the shortest path. Let $\Delta(s, u)$ denote the chokepoint of the path from s to u with the largest chokepoint. Claims (a) and (b) translate to the following two claims: (A) $d[u] \leq \Delta(s, u)$ for all u at all times, and (b) at the time u is extracted from Q, $d[u] = \Delta(s, u)$. We can establish these two claims by showing properties (I) and (II) analogous to the properties (i) and (ii) above: (I) if the last edge on the path with the largest chokepoint from s to v is (u, v), then $\Delta(s, v) = \min\{\Delta(s, u), w(u, v)\}$, and (II) for any edge (u, v), $\Delta(s, v) \geq \min\{\delta(s, u), w(u, v)\}$. Note that all we have done is to replace + by min and $\leq$ by $\geq$. Both claims (I) and (II) can be proved in a manner very similar to the proofs of (i) and (ii) for the original Dijkstra's algorithm.

2. (5 points) Shortest paths with few tolls

We have focused our attention on shortest path problems where there is a single weight for each edge, and we need to find the shortest paths under these weights. In real life, often we have multiple criteria for determining good routes; for example, the cost of a route and the time taken. It turns out that many such problems are actually much harder than the standard shortest paths problems; but not all such problems are hard.

You are given a directed graph $G = (V, E)$ with weights on edges $w : E \to \mathbb{R}$. A subset T of the vertices in V are marked as *toll nodes*.

Design an algorithm that takes as input G, w, T, a source s, a destination t, and an integer k, and determines a path from s to t that contains at most k toll nodes and has shortest weight among all such paths. State the running time of your algorithm. In your submission, you need not prove the correctness of your algorithm; but please note that you need to be able to prove it if asked.

Answer: There are several ways to solve this problem. One approach is to calculate, for each v, for each $0 \leq i \leq k$, the shortest-weight path from s to v using at most i intermediate toll nodes. Let $d(v, i)$ denote the weight of shortest-weight path from s to v using at most i intermediate toll nodes.

We first compute $\delta(u, v)$ for all u, v by running any standard shortest paths algorithm on the graph G with all outgoing edges from toll nodes removed. That is, $\delta(u, v)$ is the weight of a shortest path from u to v using no intermediate toll nodes. So $d(v, 0) = \delta(s, v)$.

To compute $d(\cdot, i)$ from $d(\cdot, i-1)$, we can compute:

$$d(v, i) = \min\{d(v, i-1), \min_{\text{toll node } x}(d(x, i-1) + \delta(x, v))\}.$$

That is, the shortest path from s to v using at most i intermediate toll nodes is either (a) the shortest path from s to v using at most $i-1$ intermediate toll nodes, or a shortest path from s to a toll node x using at most $i-1$ intermediate toll nodes (i.e., not counting x) followed by the shortest path from x to v using no toll nodes. This iteration takes $O(n^2k)$ time since there are nk entries to be filled, and each entry takes time $O(n)$.

For positive weights, the above algorithm (using Dijkstra multiple times for the first phase) will take $O(mn \log n) + O(n^2k)$. For negative weights, assuming no negative weight cycles, the above algorithm (using Floyd-Warshall) for the first all-pairs, will take $O(n^3 + n^2k)$.

Also, we have only shown how to calculate the weight of the path, not the actual path. We can do so by maintaining the intermediate nodes that yield the shortest paths.

3. (1 + 4 = 5 points) Determining period with most rainfall deficit

Being the chief climatologist of your city, you have been given the rainfall data for each day for the last 100 years in your city. So you have an array $R[1..n]$ where $R[i]$ lists the amount of rain measured in your city on the ith day (from some starting point).

Let $\mu = (\sum_{1 \leq i \leq n} R[i])/n$ denote the average rainfall in R. For an interval $I = [\ell, r]$, $1 \leq \ell, r \leq n$ of days, we define the *deficit* of I to be $\mu(r - \ell + 1) - (\sum_{\ell \leq i \leq r} R[i])$. That is, the deficit of I is the difference between the total amount of rain expected in I and the total amount of rain that actually fell in I. (Note that the deficit of an interval can be negative.)

For a new climate study, you have been asked to find an interval I that has the largest deficit.

(a) Design a brute-force $\Theta(n^2)$ time algorithm for your problem.

Answer: There are $n(n-1)/2$ intervals. We compute the deficit of each interval, and select the one that has the largest deficit.

(b) Having just completed CS5800, you think that you can do much better. Design a much more efficient $\Theta(n)$ time algorithm for your problem.

You need not prove the correctness of your algorithms, but please note that you need to be able to prove it if asked.

Answer: First notice that we can replace the given list by a new list D, where $D[i]$ is the deficit on day i (could be negative). What we are seeking is an interval that has the largest contiguous sum in D. We give an $O(n)$ time algorithm below.

Let $M[i]$ denote the ending index of the interval starting at i that has the largest deficit among all intervals starting at i. And let $S[i]$ denote the sum of the deficits in the interval $[i, M[i]]$. We give the recurrences for $M[i]$ and $S[i]$. For the base case, $M[n]$ is simply n and $S[n]$ is simply $D[n]$. For $0 \leq i < n$, we have

$$M[i], S[i] = \begin{cases} i, D[i] & \text{if } S[i+1] < 0 \\ M[i+1], D[i] + S[i+1] & \text{otherwise} \end{cases}$$

We compute M by an iterating over i from $n-1$ down to 1. To calculate the interval with the largest deficit, we simply determine the i such that $S[i]$ is maximized and return the interval $[i, M[i]]$. The number of iterations is at most n and each iteration takes $O(1)$ time. The last set of steps also take $O(n)$ time. So the overall running time is $O(n)$.

4. (5 points) Longest common subsequence of three sequences

Design an efficient algorithm that takes as input three sequences X, Y, and Z of length m, n, and p, respectively and returns the longest common subsequence of X, Y, and Z. Prove the correctness of your algorithm and analyze its worst-case running time.

Note that W is a common subsequence of X, Y, and Z if and only if W is a subsequence of X, W is a subsequence of Y, and W is a subsequence of Z.

Answer: It may be tempting to solve this by first finding the longest common subsequence, say S, of X and Y, and then finding the longest common subsequence of S and Z. This strategy does not always work.

- for $i = 0$ to m, for $j = 0$ to n: $L[i, j, 0] = 0$
- for $i = 0$ to m, for $k = 0$ to p: $L[i, 0, k] = 0$
- for $j = 0$ to n, for $k = 0$ to p: $L[0, j, k] = 0$
- for $i = 1$ to m:
  - for $j = 1$ to n:
    - for $k = 1$ to p:
      - If $X[i] = Y[j] = Z[k]$ then $L[i, j, k] = L[i-1, j-1, k-1] + 1$.

· else $L[i,j,k] = \max\{L[i-1,j,k], L[i,j-1,k], L[i,j,k-1]$.

Running time is $\Theta(nmp)$.

5. (5 points) Taking turns

Alice and Bob play the following game: Given a sequence of numbers:

$$a_1, a_2, ..., a_n$$

each player at her/his turn, takes either the first or last number of the sequence. (So if Alice takes a_1 at the first turn, then Bob can take either a_2 or a_n at his turn. If Bob takes a_n then Alice can take either a_2 or a_{n-1} and so on)

The game ends when there are no numbers left. At the end of the game the score of each player is the sum of the numbers she/he took, minus the sum of the numbers of the opponent.

Describe a dynamic programming algorithm that computes the maximal score that the first player can guarantee. Prove the correctness of your algorithm. What is the complexity of the algorithm that you suggest?

Answer: Let $S(i,j)$ where $1 \le i \le j \le n$ denote the best score a player can guarantee by having first pick at the subsequence $a_i, ..., a_j$. Clearly we have:

$$S(i,j) = \begin{cases} a_i, & i = j; \\ \max\{a_i - S(i+1,j),\ a_j - S(i,j-1)\}, & j > i. \end{cases}$$

Using the above formula we can easily calculate $S(1,n)$ in $O(n^2)$ time.

6. (5 points) Picking entries from a matrix

Let A be an $n \times m$ matrix of positive integers; A has n rows and m columns. You are asked to pick a total of k entries from the matrix that yield the maximum sum, under the following constraint: the entries that you pick in any row have to start from the first column and be contiguous. That is, your problem is to determine k_1, k_2, ..., k_n such that

$$k_1 + k_2 + \cdots + k_n = k, \text{ and}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{k_i} A[i,j] \text{ is maximized.}$$

Design an efficient algorithm to solve the above problem. State the worst-case running time of your algorithm. You need not prove the correctness of your algorithm, but please note that you need to be able to prove it if asked.

Answer: Let $C[i,j]$ denote the total sum obtained in the optimal selection of j entries among rows 1 through i. Let $B[i,j]$ denote the number of entries selected in the ith row in this optimal solution. We can establish the following recurrences for $C[i,j]$ and $B[i,j]$.

We set $C[i,0]$ to 0 for all i. Furthermore, for all j, $C[1,j]$ equals $A[1,1] + \ldots + A[1,j]$ and $B[1,j]$ equals j.

For $i > 1$ and $j > 0$, we have the following.

$$C[i, j] = \max_{0 \leq \ell \leq j} \{(A[i, 1] + A[i, 2] + \ldots + A[i, \ell]) + C[i - 1, j - \ell]\}.$$

We set $B[i, j]$ to be the ℓ that gives the maximum value in the above recurrence.

The algorithm proceeds by iteratively computing $C[i, j]$ and $B[i, j]$, i going from 1 through n and j going from 1 through k. The final assignment can be computed using the B array. In each iteration, we need to take the maximum of at most k entries. Therefore, there are at most nk iterations, each taking $O(k)$ time, leading to a runtime of $O(k^2 n)$.