CMSC 27200 Assignment 4

Rachel Hwang

February 7, 2014

Collaborated with: Andrew Ding and James Porter.

- **Exercise 6.5**
  The following function $Opt(j)$ defines the maximum possible quality value obtainable by segmenting the string from index 1 though j.

$$Opt(0) = 0$$
$$Opt(j) = max_{1 \leq i \leq j}(W_{i,j} + OPT(i-1))$$

  At character j in the string, we find the maximum possible value of the sum of the weight of a segment from an index i to j plus the maximum value at i, $Opt(i)$. We try this for every i from 1 to j. In essense, this part of the algorithm finds the best value obtainable by segmenting the word that j is a part of at every possible segmentation point.

  Step 1: Iterate over the list calculating Opt at every character and saving each result (memoization).

  Step 2: In order to find the indices of the segmentation points, we begin iterating through the string from the end to the front. Begin with the last character, index n, as our current character, which we can call j. Now we begin to iterate through the rest of the string, call the second current character i, beginning with $i = n-1$ then index $i = n-2 \ldots$. If ever $Opt(j) - Opt(i) = W_{i,j}$, we know that the string from i to j must be a segment in the optimal segmentation, so we add index $i-1$ to a list of segment boundary indices and run step 2 again beginning from $i-1$.

  Claim: This algorithm returns an optimal segmentation.
  Proof: Let $opt(i)$ return the value of the optimal segmentation at index i. By definition $opt(0) = 0$, since an empty string cannot be segmented. By way of induction, suppose $opt(i)$ finds $optimal(i)$ for all $i < j$ where $j > 0$ and $optimal()$ provides the truly optimal solution. By induction hypothesis, we know that $opt(i) = optimal(i)$ for all $0 \leq i < j$. In $optimal()$, the jth letter is in a word that begins with the ith letter for some $i \leq j$. Since value is calculated by taking the maximum possible quailty sum, it follows that

$$optimal(j) = max_{1 \leq i \leq j}(W_{i,j} + opt(i-1)) = opt(j)$$

  Complexity: With memoization, step one requires the calucation of Opt n times, and then a constant ammount of work at each index trying segments at every other index. This ammounts to $O(n^2)$ work. The final step makes only a single pass over the list wih a fixed ammount of work at each iteration, so this runs in $O(n)$ time. Thus, the algorithm runs in $O(n + n^{@}) = O(n^2)$ time.

- **Exercise 6.7**
  The following function $Opt(j)$ and variable min define the maximum possible profit at any day $1 \leq j \leq n$:

$$Opt(1) = 0, \qquad min_1 = P_1$$
$$Opt(j) = max(P_j - min_{j-1}, \ \ Opt(j-1)), \qquad min_j = min(P_j, \ \ min_{j-1})$$

  As we move recursively through the entire list, we must save several pieces of additional imformation. 1) We need to keep track of the minimum value that we've seen (min_i) and we need to record the index i of the day which first produced the current maximum. For instance, if at day x, the maximum possible profit increases, we record x as the index of the sell day until a higher profit is discovered. This means if $Opt(j) \neq Opt(j-1)$, record j as the current sell day, call it s.

  Step 1: iterate through the list of days, for each day i between day 1 to day n, calculating $Opt(i)$, min_i

and s, storing these values for each day. $Opt(k)$ will be the maximum possible profit and s will be the sell day.

Step 2: in order to find the buy day, we simply iterate through the days in order once more. At each day i, we check to see if $P_s - P_i$ gives us the optimum value we obtained. If so, return i as the buy day and s as the sell day.

Proof: By induction, as in the above exercise, on the recurrence defined above. The base case is trival: on day 1, the maximum possible profit is 0. Now assume $opt(i)$ returns the max profit for all $1 \leq i < j$. On any given day, we can either sell or not sell. On day j, if the difference of P_j and the minimum value we've seen so far is greater than the $opt(j-1)$, we know that it is optimal to sell on day j. Otherwise, we keep the same *opt* value from before.

Complexity: Using memoization, each iteration in step one takes a constant ammount of time since all values are known (having saved all previous $Opt()$ values) and involve only comparisons. Step 1 therefore runs in $O(n)$ time. Step 2 again involves a constant ammount of work, a comparison at each of n days, so step 2 runs in $O(n)$ time. Thus, the overall algorithm runtime is linear, $O(2n) = O(n)$.

- **Exercise 6.12**
Similar to problem 1, we are looking for the minimum cost by looking at the cost incurred by adding the file at every possible index ahead of the current index (analogous to segmenting). We can define the function *opt*, which returns the minimum total configuration cost at a given index as follows:

$$opt(0) = 0$$
$$opt(j) = C_j + \min_{1 \leq i \leq j} \left(opt(i) + \sum_{x=i+1}^{j} A_x \right)$$

The algorithm is thus:

1. As defined above, calculate the *opt* value for every index.
2. In order to find the indices of servers which should get a copy of the file, we simply iterate through the servers in reverse order once more, starting with n. At each server i, we check to see if $opt(n) - opt(i) = C_n + \sum_{x=i+1}^{n} A_x$. If so, add index i to a list of servers which should have the file on them (n is on there by default), and run step 2 again beginning with i instead of n. This should repeat until all indices are finished.

Proof: Again, by induction over the defined recurrence. The base case is simple: at index 0, there is no server with the file on it, hence cost is 0. Assume *opt* yields the proper, optimum result for all $1 \leq i < j$. The cost of a configuration at the jth server is the sum of C_j (since we must access at the last server) + the access time between i and j, where i is the index of the last server with the file, + $opt(i)$. Since we want a minimum cost configuration, we simply take the minimum of this sum for all i values less than j. It follows that *opt* gives the optimal, minimized cost.

Complexity: For each index j we perform a constant ammount of work another $j-1$ times maximum, since we must check $j-1$ other servers. For a list of n, this gives us $O(n \cdot n) = O(n^2)$ time.

- **Exercise 6.22**
We can first take advantage of the Bellman-Ford algorithm (memoizing) to calculate the minimum value of a path from v to t. Once we have that *opt* value for every vertex in the graph, we can calculate the number of minimum paths from any node to t using the following definitions:

$opt(i, v)$ = min cost from v to t, len $\leq i$
$num(i, v)$ = number of minimal paths from v to t, len $\leq i$
$num(i, t) = 1$, for any i

To calculate $num(i, v)$,

1. Find (for w st. $\exists e(v, w)$), $M = min_{all\ w}(C_e + opt(n - i - 1, w))$
2. Define S to be a subset of w such that all elements of S achieve M

3. $num(i, v) = \sum_{w \in S} num(i-1, w)$

Using these definitions, the solution to this problem is given by $num(n-1, s)$, since we know that all cycles have a positive weight, the minimum path length must be no greater than $n-1$. $num(n-1, s)$ gives the number of minimal paths from s to t. To calculate this value, the algorithm is as follows:

1. Run the Bellman-Ford Algorithm, storing the *opt* value for each node v and integer i less than n.
2. Calculate $num(n-1, s)$, using the above recursive definition of num and the stored *opt* values.

Proof: By induction. The base case is simple: when $i = 0$, $num(0, v)$ is 0, unless we are at t, in which case it is 1. Assume $num(k, v)$ does indeed return the correct value for all $1 \leq k < j$. Now, for $i = j$, we know that the number of paths from any vertex v to t of at most i edges will be the sum of $num(i-1, w)$, where w is a neighbor with an edge to v if w is in S defined above. It follows that recurrence defined above counts the correct number of paths.
Complexity: The complexity of the first step is given in the textbook to be $O(n^3)$. As for the second step, consider that we need to calculate num for all i values and all v, both of which are simply n. This is n^2 calculations of num. However, since each calculation involves finding the minimum of the defined value using all neighbors of n, a max value of $n-1$. So the second step runs in $O(n \cdot n \cdot (n-1)) = O(n^3)$.

- **Extra Credit** Consider a variant of the Sequence Alignment Problem with a different cost function for gaps. Instead of charging delta for each gap, when we have a sequence of gaps the cost will be a linear function of the length, that is the cost of j successive gaps will be a+bj (where a will be comparable to delta but b will be a lot smaller. This represents more accurately the biological process where in the process of DNA copying a strand was cut, and a small sequence inserted.) Derive a formula similar to 6.16 (page 282 of the text), and sketch a dynamic programming algorithm to find the minimum cost alignment.

$$opt(i, j) = \min_{k \leq i, L \leq j} [opt(i, j-L), opt(i-k, j), opt(i-1, j-1) + \alpha]$$
$$opt(i, 0) = a + ib$$

The above function defines $opt(i, j)$ as the minimum cost alignment at sequence indices i and j. The algorithm simply finds $opt(i, j)$ for all $i < n$ and $j < m$, where n and m are respective sequence lengths. After storing the values for all *opts*, We simply use the same kind of backtracing techinique as before to derive the actual string alignment.

Consider an optimal solution *Optimal* and characters x_i and y_j. If in *Optimal*, they are matched, then in *opt*, we choose to use $opt(i-1, j-1) + \alpha$ in calculating $opt(x_i, y_j)$. If they are not matched, as shown in the book, there must be a gap. Assume the optimal solution has a gap of length w, so w blanks matched with w characters in either y or x. This must cost at least $a + bw + Opt(i, j-w)$ or $a + bw + Opt(i-w, j)$. Since *opt* returns the minimum of the three options, match, gap in y, gap n x, it is guarenteed to return a solution that is as good or better than the *optimal*.