CS 577 Homework 6

Alex Pizzuto

Question One

We are given two strings, A=$a_1a_2 \dots a_m$ and B=$b_1b_2 \dots b_j$, and we seek to convert A to B using the fewest number of deletions, insertions, and changes of letters in A to some different letter using dynamic programming. First, we recognize that this is very similar to the problem on sequence alignment that we have covered, but now instead of adding gaps to either string we can add gaps or remove letters, and now our mismatch cost and δ parameter are both of equal magnitude. Thus, we have:

$$opt(i,j) = \min\Big((1-\delta_{i,j}) + opt(i-1,j-1), 1 + opt(i-1,j), 1 + opt(i,j-1)\Big)$$

where $opt(i,j)$ denotes the minimum cost of an alignment between $a_1a_2 \dots a_i$ and $b_1b_2 \dots b_j$ and δ_{ij} is the usual Kronecker delta used in applied mathematics, and is equal to 1 when the indices are equal, and 0 otherwise. Here, however, if we fall into the first option, we change a_i to be equal to b_j. In the second option, we would add a gap to A, and in the third case we would delete from A. Thus, our algorithm is:

```
Array M[0...m, 0...n];
Initialize M[i, 0] = i for each i;
Initialize M[0, j] = j for each j;
for j in 1 to n do
    for i in 1 to m do
        Use recurrence above to compute M[i,j];
    end
end
Return M[m, n];
```

As with other dynamic programming algorithms, we can trace back through our array to figure out the way in which the optimal alignment was generated, starting from M[0,0] and proceeding to M[m,n]. When it is clear how to proceed from one set of indices to the next, we can use the following to decide what to print:

```
if first option was the minimal option then
    print replace A[i] with B[j] if the two letters are different;
end
if second option was the minimal option then
    print add gap to A at location i;
end
if third option was the minimal option then
    print delete A[i];
end
```

Question Two: 6.14

a First, we try and find the shortest path working under the assumption that there exists an s-t path in all graphs, $G_0, \ldots, G_b$. In order to do this, we create a new graph, call it G'. Start by setting $G' = G_0$, and then remove all edges that do not appear in all other graphs, $G_1, \ldots, G_b$ (in other words, G' will have only the edges that appear in every G_i). Then, use Dijkstra's algorithm on G' to find the shortest s-t path.

b Now, we provide a polynomial time algorithm to find a sequence of paths, $P_0, \ldots, P_b$ of minimum cost, where cost is defined as:

$$\text{cost}\left(P_0, P_1, \ldots, P_b\right) = \sum_{i=0}^{b} \ell\left(P_i\right) + K \cdot \text{changes}\left(P_0, P_1, \ldots, P_b\right)$$

In order to accomplish this, we will use subproblems to find $opt(i)$, where $opt(i)$ denotes the minimum cost of the solution just considering graphs $G_0, \ldots, G_i$. Then, for each subproblem, we must figure out if it is optimal to keep our path the same throughout all iterations, or to switch over at some other point, giving us the following recurrence relation. If there is one path throughout all graphs, we can find it as we did in part (a), and we will denote this path as $P'_{i,j}$ to represent the shortest s-t path that is present in all graphs, $G_i, \ldots, G_j$. Then, we get the following recurrence:

$$opt(b) = \min\left((b+1)\ell(P'_{0,b}) \, , \min_{1 \le i \le b}(opt(i) + (b-i)\ell(P'_{i+1,b}) + K)\right)$$

where the first term represents having one path throughout all the graphs, and the nested minimization is to fix what graph it might be optimal to change paths at. We can implement this with an array to store the optimal values, $M[0 \ldots b]$, and write out our algorithm as:

for *i from 0 to b* **do**
  $M[i] = \min\left((i+1)\ell(P'_{0,i}) \, , \min_{1 \le j \le i}(M[j] + (i-j)\ell(P'_{j+1,i}) + K)\right)$
end

And finding each P' path requires us to construct the proper graphs ($\mathcal{O}(n^2 b)$), and we could consider up to $\mathcal{O}(b^2)$ of these combinations, so we are still overall polynomial ($\mathcal{O}(n^2 b^3)$).

Question Three: 6.25

Our goal is to design an algorithm that takes the prices, $p_1, \ldots, p_n$ and the function that determines the penalty for selling a certain number of shares, $f(\cdot)$, and returns the maximum profit. Here, our problem is underdetermined if we just try to create subproblems based off of the number of days, so we pass an extra parameter (number of shares remaining) to our optimization recurrence relation, as well as the penalty that we have incurred up to that day. We thus have the following recurrence relation:

$$opt(i, s, k) = \max_{t \le s}\left((p_i - f(t) - k) \cdot t + opt(i+1, s-t, k+f(t))\right)$$

where $opt(i, s, k)$ denotes the maximum profit attainable from selling a remaining s stocks from day i to n, already with a penalty of k. We can then put this into an algorithm, creating a 3-dimensional tensor:

M[0 ... n, 0 ... x, 0... p_1];
for *a in* p_1 *to 0* **do**
  for *i in n to 0* **do**
    for *j in 0 to x* **do**
      M[i,j] = $\max_{t \leq j} \left((p_i - f(t) - a) \cdot t + M[i+1, j-t, a+f(t)] \right)$
    end
  end
end

This will return the maximum profit by looking at M[0,x,0]. To figure out the actual sequence of selling stocks, you can trace back in the tensor from this position to M[n,0,p_1].