CMPT 441-711: Homework 2

Instructor: Prof. Cenk Sahinalp

Fall 2014

Question 6.9

Denote by S_k the set of floors that can be reached from floor 32 in k, but not in $k-1$, steps. Then, obviously, $S_0 = 32$. Now we give description of the algorithm for computing S_k, for any $k > 0$, assuming that all $S_0, S_1, \ldots, S_{k-1}$ have been computed.

From the definition of S_k it follows that any element x of S_k was reached from at least one of $x-11$, where $x - 11 \in S_{k-1}$ and $x+6$, where $x + 6 \in S_{k-1}$. Also, if some element $y \in \{1, \ldots, 50\}$ belongs to any of $S_0, S_1, \ldots, S_{k-1}$, then $y \notin S_k$ (as shorter path from 32 to y has already been found). Therefore, in order to compute S_k, we iterate through all $y \in \{1, \ldots, 50\}$ and for each of the y that does not belong to any of $S_0, S_1, \ldots, S_{k-1}$ we check if any of $y - 11$ and $y + 6$ belongs to S_{k-1}. If it does, then we add y to S_k. In the case that some of S_j remains empty after this procedure then either the shortest paths to all of the floors have been found or there are some floors that can not be reached using the given operations. In this problem this can not happen as 11 and 6 are relatively prime (so path from i to j exists for any pair of floors), but for example in the case when we can only go 10 floors up and 5 floors down it is impossible to reach floor 33 starting from floor 32. It can be easily computed that S_{14} is the S_k containing 33, hence the answer to the first question is: we have to press an elevator's button at least 14 times in order to get from floor 32 to floor 33. Using simple backtracking it can be found that, in total, we have to go 5 times *up* and 9 times *down* in order to reach floor 33.

For the second part of the question, we first prove that in this case the answer to the first question (where $5 \cdot 11 + 9 \cdot 6 = 109$ floors are visited) gives the answer to the second question as well. Assume on contrary that there exists another path where lower number of floors is visited. It implies that we have to take less than 9 button presses *down* or less than 5 button presses *up* (if none of these two is true than we end up with solution requiring at least $5 \cdot 11 + 9 \cdot 6 = 109$ floors to be passed on our way). But, if we take less than 9 button presses *down* then, in total, we will pass by less than $9 \cdot 6 = 54$ floors *down*, implying that we will pass by less than 55 floors *up*. This further implies that we have less than $\frac{55}{11} = 5$ *up* button presses. This gives us solution where less than $9 + 5 = 14$ button presses have been used, contradicting first part where we showed that 14 is the least number of button presses we need. The case when less than 5 button presses *up* are used is handled in analogous way. Therefore, our assumption is wrong and we conclude that the shortest time to go from 32 to 33 requires passing 109 floors.

Question 6.22

Optimal overlap alignment between $\mathbf{v}$ and $\mathbf{w}$ (with length m and n, respectively) is in fact the same as global alignment between these two strings when we do not penalize "ending" gaps in $\mathbf{v}$ and "starting" gaps in $\mathbf{w}$. For example, here:

```
v:  T  A  T  A  T  A  -  -
w:  -  -  -  A  A  A  T  T
```

We will not penalize "–" since they belong to the ending gap of $\mathbf{v}$ / starting gap of $\mathbf{w}$. Let $s(i,j)$ be the global alignment score between $\mathbf{v}[1 \ldots i]$ and $\mathbf{w}[1 \ldots j]$. In order to discard penalty for starting gaps in $\mathbf{w}$, it is enough to set $s(i, 0) = 0$ for each i, as aligning beginning of the first string with a blank string of an arbitrary length is equivalent to the discarding of the starting gaps of the second string. In order to discard ending gaps in **v**, it is enough to observe that ending gaps of size l will be aligned with $\mathbf{w}[n-l+1\dots n]$. So, we can in fact perform global alignment with $n-l+1$-prefix of **w**. So, in order to find a best overlap alignment score, it is enough to iterate through each prefix of **w** and check its global alignment score. As we keep already these scores in our edit matrix T, combining these observations will lead us to the following algorithm:

```
Overlap-Align(v,w): (of lengths $m$ and $n$)
   Initialize matrix $T[m,n]$
   Set $T[0\dots m,0]=0$
   Global-Align(v,w,T)

   Set $M=\max_{0\le i\le n} T[m,i]$
   Output $M$
```

Complexity will be dominated by the complexity of the global alignment algorithm (preprocessing and finding maximum is $O(n)$) – thus the complexity $O(mn)$. We can reconstruct the path using the same method as in global alignment problem (we just need to start from the row which contains maximum value).

Question 6.24

Optimal semiglobal alignment between **v** and **w** (with length m and n, respectively) is the same as global alignment between these strings, except that we do not penalize starting and ending gaps in **v** and **w**. For example, here:

```
v:  T  A  T  A  T  A  C  C
w:  -  -  -  A  A  A  -  -
```

We will not penalize "-" because they belong to the staring and ending gap of **w**. Let $s(i,j)$ be the global alignment score between $\mathbf{v}[1\dots i]$ and $\mathbf{w}[1\dots j]$. As in the previous task, in order to discard penalty for starting gaps in **v** and **w**, it is enough to set $s(0,j)=s(i,0)=0$ for each i,j. In order to discard ending gaps in **v**, it is enough to observe that ending gaps of size l will be aligned with $\mathbf{w}[n-l+1\dots n]$. So, we can in fact perform global alignment with $n-l+1$-prefix of **w**. Thus, in order to find a best semiglobal alignment score, it is enough to iterate through each prefix of **w** and check its global alignment score. We will do the same to the prefixes of **v** in order to discard the ending gap penalty in **w**. As we keep already these scores in our edit matrix T, combining these observations will lead us to the following algorithm:

```
Semiglobal-Align(v,w): (of lengths $m$ and $n$)
   Initialize matrix $T[m,n]$
   Set $T[0,0\dots n]=T[0\dots m,0]=0$
   Global-Align(v,w,T)

   Set $M=\max(\max_{0\le i\le n} T[m,i], \max_{0\le i\le m} T[i,n])$
   Output $M$
```

Complexity is still dominated by complexity of global alignment algorithm (preprocessing and finding maximum is $O(2n)$) – thus the complexity $O(mn)$. We can reconstruct the path using the same method as in global alignment problem (we just need to start from the row and the column which contains maximal value).

Question 6.25

We are given **v** of size m and **w** of size n where $m>n$ and $k=m-n$. It is obvious that problem reduces to insertion of k gaps within **w** so that we minimize number of mismatches.

Suppose that we already processed j characters of $\mathbf{w}$ and that we added i gaps. Then, we processed $i+j$ characters inside $\mathbf{v}$. In order to process next character we can either match $j+1$-th character of $\mathbf{w}$ with $i+j+1$-th character of $\mathbf{v}$, or we can decide to add a gap (of course, if we can, i.e. if $i<k$). So our recurrence looks like:

$$s(i,j) = \max\begin{cases} s(i,j-1) & +\Delta(i+j,j) \\ s(i-1,j) & +\Delta(i+j,-) \end{cases}$$

Pseudocode is:

```
NoDelete-Align(v,w): (of lengths $m$ and $n$)
   Initialize matrix $T[k,n]$
   Set $T[0,0]=0$
   For $i=0$ to $k$:
      For $j=0$ to $n$:
         $T[i,j] = \max(T[i,j-1]+\Delta(i+j,j), T[i-1,j]+\Delta(i+j,-))$
            (n.b. we do $j-1$ and $i-1$ steps only if $j$ or $i$ are greater than $0$
             otherwise we skip calculation of that argument)
   Output $T[k,n]$
```

As we have k gaps and n characters of $\mathbf{w}$, complexity is obviously $O(kn)$.

Question 6.26

We first declare 4 global variables i_{opt}, i'_{opt}, k_{opt} and $score_{opt}$ with obvious meanings. For each pair (i,i') such that $i \geq i'$ we find global alignment between $v_i v_{i+1} \ldots v_n$ and $v_{i'} v_{i'+1} \ldots v_{(n-i)+i'}$ (note that $n-i+i' \leq n$ and we do not have to align the whole $v_{i'} v_{i'+1} \ldots v_n$ against $v_i v_{i+1} \ldots v_n$ as we are looking only for substrings of equal lengths). We keep the optimal alignment between $v_i v_{i+1} \ldots v_p$ and $v_{i'} v_{i'+1} \ldots v_q$ in $V[p][q]$. Then, for all k satisfying the inequality $i'-i+k > MinGap$ we compare $V[i+k][i'+k]$ with $score_{opt}$ and adjust values of i_{opt}, i'_{opt}, k_{opt} and $score_{opt}$ if $V[i+k][i'+k] > score_{opt}$. In order to reduce complexity the last step can be done at the same time when $V[i+k][i'+k]$ entry is filled during the run of global alignment algorithm. We do the analogous for pairs (i,i') such that $i<i'$.

Since we have $O(n^2)$ pairs (i,i') and finding the global alignment takes, on average, $O(n^2)$ time per pair, the running time of this algorithm is $O(n^4)$.

Question 6.30

Let $\bar{\mathbf{v}}_1$ and $\bar{\mathbf{v}}_2$ be the strings constructed from $\mathbf{v}_1$ and $\mathbf{v}_2$ after the optimal alignment (i.e. strings with dash signs for deletion and insertion). It is obvious that $|\bar{\mathbf{v}}_1| = |\bar{\mathbf{v}}_2| = q$. Let's define the anomaly as insertion, deletion or mismatch. Take some prefix of $\bar{\mathbf{v}}_\mathbf{1}$ such that it contains k anomalies (compared with $\bar{\mathbf{v}}_\mathbf{2}$), and some suffix of $\bar{\mathbf{v}}_\mathbf{2}$ such that it contains $d(\mathbf{v_1},\mathbf{v_2})-k$ anomalies (we obviously have $d(\mathbf{v_1},\mathbf{v_2})$ anomalies in total). Concatenate these strings and remove all dash characters from the newly constructed string. Let's prove that for this string, say $\mathbf{w}$, we have $d(\mathbf{v}_1,\mathbf{w})+d(\mathbf{v}_2,\mathbf{w}) = d(\mathbf{v}_1,\mathbf{v}_2)$. It is obvious that $d(\mathbf{v}_1,\mathbf{w})$ will be $d(\mathbf{v_1},\mathbf{v_2})-k$, because chosen prefix of $\mathbf{w}$ will be aligned with $\mathbf{v_1}$ without penalty, and rest will be aligned with $d(\mathbf{v_1},\mathbf{v_2})-k$ (because of our choice of second part of the $\mathbf{w}$). In the same way we get that $d(\mathbf{v}_2,\mathbf{w}) = k$, so $d(\mathbf{v}_1,\mathbf{w})+d(\mathbf{v}_2,\mathbf{w}) = d(\mathbf{v_1},\mathbf{v_2})+k-k = d(\mathbf{v}_1,\mathbf{v_2})$.

Now, we need to minimize $|d(\mathbf{v}_1,\mathbf{v_2})-k-k| = |d(\mathbf{v}_1,\mathbf{v_2})-2k|$. In order to minimize given expression, we must find k such that k and $d(\mathbf{v}_1,\mathbf{v_2})-2k$ are as close as possible. Obviously, such condition yields $k = \lfloor d(\mathbf{v}_1,\mathbf{v_2})/2 \rfloor$. If $d(\mathbf{v}_1,\mathbf{v}_2)$ is even, our expression is 0, and we have found our desired string. Suppose that k is not even. Then $|d(\mathbf{v}_1,\mathbf{w})-d(\mathbf{v}_2,\mathbf{w})| = \lceil d(\mathbf{v}_1,\mathbf{v_2})/2 \rceil - \lfloor d(\mathbf{v}_1,\mathbf{v_2})/2 \rfloor = 1$. If there is some other string $\hat{\mathbf{w}}$ for which $|d(\mathbf{v}_1,\hat{\mathbf{w}})-d(\mathbf{v}_2,\hat{\mathbf{w}})| = 0$ and $d(\mathbf{v}_1,\hat{\mathbf{w}})+d(\mathbf{v}_2,\hat{\mathbf{w}}) = d(\mathbf{v}_1,\mathbf{v}_2)$, then we would get $2d(\mathbf{v}_1,\hat{\mathbf{w}}) = d(\mathbf{v}_1,\mathbf{v}_2)$, which is not possible because $|d(\mathbf{v_1},\mathbf{v_2})-2k| = 1$ (which implies that $d(\mathbf{v_1},\mathbf{v_2})$ is odd number).

Summing this up, we end with the following algorithm:

```
Mini-W($v_1$,$v_2$):
    Set $k = d(v_1, v_2)/2$
    Set $\bar{v}_1, \bar{v}_2$ as alignment results of strings $v_1, v_2$
    Set $w$ as prefix($\bar{v}_1$) with $\lfloor k \rfloor$ anomalities +
                suffix($\bar{v}_2$) with $\lceil k \rceil$ anomalities
    CleanSpaces($w$)
    Return $w$
```

Complexity is obviously $O(mn)$ for construction of the alignment. Post-processing takes at most $O(m+n)$, so total complexity is $O(mn)$.

Question 6.36

Assume that the length of first string is m and the length of second string is n. For this question, we give trivial solution where each of the strings is cut at m (resp. n) possible positions and global (resp. local) alignment algorithm is then used to find the best alignment for each possible cut-pair. The optimal solution is the one having the highest score among all of these pairs.

There are m possible ways to cut first and n possible ways to cut second string, giving in total mn possible ways to cut both strings. Each of (local, global) alignment algorithms takes $O(mn)$ time, so the overall running time of this algorithm is $O(m^2n^2)$.

Although these are very difficult problems, there exist more efficient algorithms for solving them. One of such algorithms is explained in the lecture 13. at `http://web.cs.ucdavis.edu/~gusfield/cs224f11/`.

Question 6.40

This is a fairly straightforward modification of the semi-global alignment problem, where we have following changes:

1. We cannot continue from diagonal unless $v(i) = w(j)$, i.e. there is no option for mismatch (we can simply say $\Delta(i, j) = -\infty$ if $v(i) \neq w(j)$)
2. $\Delta(i, -)$ will be zero (unpenalized) if $v(i-1) = v(i)$
3. $\Delta(-, j)$ will be zero (unpenalized) if $w(j-1) = w(j)$

Pseudocode follows:

```
Homoalign(v,w): (of length $m, n$)
    Initialize matrix $S[m, n]$
    Set $T[0, 0] = 0$
    For $i = 1$ to $m$:
        For $j = 1$ to $n$:
            If $v[i] \neq w[j]$:
                $\Delta(i, j) = -\infty$
            If $v[i] = v[i-1]$
                $\Delta(i, -) = 0$
            If $w[j] = w[j-1]$
                $\Delta(-, j) = 0$
            $S[i, j] = \max(S[i-1, j-1] + \Delta(i, j), S[i-1, j] + \Delta(i, -), S[i, j-1] + \Delta(-, j))$
            (n.b. we do $j-1$ and $i-1$ steps only if $j$ or $i$ are greater than $0$
                 otherwise we skip calculation of that argument)
    Output $S[m, n]$
```

Complexity is obviously unchanged – $O(mn)$ (as all additional operations are $O(1)$).

Question 6.44

Observation: We have multiple optimal alignments iff we have two or more values with the same maximum value in the recurrence

$$s(i,j) = \max \begin{cases} s(i-1,j-1) & +\Delta(i,j) \\ s(i-1,j) & +\Delta(i,-) \\ s(i,j-1) & +\Delta(-,j) \end{cases},$$

This means that it does not matter which path you choose to find the best alignment. During the reconstruction on any "cell" which has this property, we can choose whatever parent we want. This will lead into a different alignment with the same optimal score. This gives us general ides: try to follow each feasible direction on any "cell" that has this property and count all distinct paths.

Number of those paths could be calculated neatly with the dynamic programming. We calculate number of distinct paths $q(i,j)$ during the calculation of $s(i,j)$, where we set $q(i,0) = q(0,j) = 1$ (only one way to align empty string with other string) and

$$q(i,j) = \sum_{k,l=\text{argmax}(s(i,j))} q(k,l).$$

Intuitively, we will add number of found paths for each feasible direction we can extend. The proof of soundness is straightforward using induction, where crucial remark is that during the extending of all paths of neighbours of (i,j), each two paths coming from different directions will be different (as each different direction will yield different alignment), so we can safely add all of them and be sure that they are all distinct.

Pseudocode is:

```
Opt-Inexact(v,w): (of length $m, n$)
    Initialize matrix $S[m,n]$
    Set $T[0,0] = 0$, $Q[0,0\dots n] = Q[0\dots m,0] = 1$
    For $i = 0$ to $m$:
        For $j = 0$ to $n$:
            $S[i,j] = \max(S[i-1,j-1] + \Delta(i,j), S[i-1,j] + \Delta(i,-), S[i,j-1] + \Delta(-,j))$
            $Q[i,j] = \sum_{k,l=\text{argmax}(s(i,j))} Q(k,l)$
                (again if we can use $i-1$ and $j-1$, otherwise skip the argument)
    Output $Q[m,n]$
```

Obviously, as calculation of $q(i,j)$ is sum of three values (thus $O(1)$), total complexity will not exceed $O(mn)$.