Problem Set 5 Solution

All parts are due Thursday, December 8, 2016 at 11:59PM.

Name: Jisoo Min
Collaborators:

Part A

Problem 5-1. [35 points] Fantastic Beasts

(a) [2 points] Solution:
Summary: Score of a particular alignment of two genes is defined as the sum of the character-match scores for each index of the alignment. We need to find the \( \text{optimal\_score}(a, b) \), or the smallest possible \( \text{score}(a^+, b^+) \).
Algorithm: dynamic programming with suffixes
- Subproblem: minimum score for the alignment of suffixes \( a[i:] \) and \( b[j:] \)
- Guessing: whether to
  - insert ‘_’ to \( a \) (analogous to ‘insert’ of ‘Edit Distance DP’ from lecture)
  - insert ‘_’ to \( b \) (analogous to ‘delete’ of ‘Edit Distance DP’ from lecture)
  - leave as is (analogous to ‘replace’ of ‘Edit Distance DP’ from lecture)
- Recurrence:
  - \( dp[i, j] = \min\{4 + dp[i, j + 1], 4 + dp[i + 1, j], w(a[i], b[j]) + dp[i + 1, j + 1]\} \)
  - \( dp[|a|, |b|] = 0, \, dp[i, |b|] = 4i, \, dp[|a|, j] = 4j \)
- memoize
- Order: for \( i = |a|, |a| - 1, ..., 1, 0 \) and \( j = |b|, |b| - 1, ..., 1, 0 \)
- Solve original: \( dp[0, 0] \)
Correctness: Right to left ordering guarantees no cycle to be in the recurrence. Keeping \( dp[i, j] \) for all \( 0 \leq i \leq |a|, 0 \leq j \leq |b| \) \((|a|, |b| = \Theta(n))\) requires \( O(n^2) \) space.
Run Time: \( O(n^2) \) subproblems \( \times O(1) \) time/subproblem = \( O(n^2) \) time

(b) [3 points] Solution:
We only need to keep last and current rows (or columns) each of size \( O(n) \). Based on our recurrence, updates only come from adjacent rows (or columns). In 2D table, update values right to left in each row bottom to up. Now we use \( O(n) \) space.
(c) [5 points] **Solution:**
Create a separate data structure $P$ (list of lists) that holds parent pointers for all $0 \leq i \leq |a|$, $0 \leq j \leq |b|$ ($|a|, |b| = \Theta(n)$). As we update $dp[i, j]$, we keep track of the parent pointers and the nucleotide or the underscore. Then we backtrack and output the modified genes. ($a^+$ and $b^+$ for optimal alignment will be modifications of $a$ and $b$ with added underscores based on the parent pointer information.) With constant number of $O(n^2)$ size tables, run time is still $O(n^2)$ as analyzed in part (a).

(d) [15 points] **Solution:**
Summary: We need to compute the optimal alignment in $O(n^2)$ time with $O(n)$ space.
Algorithm: Use $optimal\_score(a, b)$ notation defined above. We only keep last and current rows (or columns) each of size $O(n)$ in a global variable as in part (b) to save space. Find $0 \leq j_1 \leq |b|$ that minimize $optimal\_score(a[^{|a|/2}], b[^{j_1}]) + optimal\_score(a[^{|a|/2}], b[^{j_1}])$.
We use recursion for both subproblems. In other words, find $0 \leq j_2 \leq j_1$ and $j_1 \leq j_3 \leq |b|$ that each minimize $optimal\_score(a[^{|a|/2}], b[^{j_2}]) + optimal\_score(a[^{|a|/2}], b[^{j_2}])$ and $optimal\_score(a[^{|a|/2}], b[^{j_2}]) + optimal\_score(a[^{|a|/2}], b[^{j_2}])$. Base case is handled as in part (a). Output answers from left to right subproblems recursively, using parent pointers as in part (c).
Correctness: On each recursive call, the updates don’t depend on other recursive calls. And we are sure that if we minimize scores for all subproblems, we will get the smallest score for our original problem.
Runtime: $T(p) = 2T(p/4) + O(p)$ where $p$ is the total number of entries in the matrix (i.e., $p = |a||b| = O(n^2)$). The final runtime is $O(n^2)$.

Figure 1: A few iterations of recursion (star: next call, yellow cell: value chosen for the column)
(e) [10 points] **Solution:**
Now we reduce the number of subproblems. In part (c), we updated $dp$ and $P$ dictionaries each of size $O(n^2)$. Now for part (e), for all $0 \leq i \leq |a|$, we say $j = i - u_1 + u_2$ ($u_1$: # of nucleotides in $a[i:]$ aligned with underscores, $u_2$: # of nucleotides in $b$ aligned with underscores). Our subproblem is now defined as $dp[i, u_1, u_2]$ which stores the minimum score for aligning $a[i:]$ with $b[j - u_1 + u_2:]$ under $u_1, u_2$ value constraints. Because $0 \leq u_1, u_2 \leq 100$, we now have $O(100^2 n)$, or $O(n)$ subproblems. For subproblems that have $u_1 = 100$ or $u_2 = 100$, we will calculate the $dp$ of the current alignment (and construct parent pointers to adjacent gene sequence) without adding more underscores (i.e., without checking anything).
Run Time: $O(n)$ subproblems $\times$ $O(1)$ time/subproblem $= O(n)$ time

**Problem 5-2.** [20 points] **Rye Elections**

(a) [5 points] **Solution:** No.

```
Prof. Caufield’s algorithm:
advertise to (c1, c3, c6); pay p1 + p3 + p6 = 15
```

```
p1 = 5  p2 = 6  p3 = 5  p4 = 5  p5 = 6  p6 = 5
```

More efficient approach:
advertise to (c2, c5); pay p2 + p5 = 12

**Figure 2:** Counterexample; $n = 6$

(b) [10 points] **Solution:**
Summary: Cities $c_{i-1}, c_i, c_{i+1}$ can be reached by advertising to $c_i$ with a price of $p_i$. We need to compute the subset of cities to run advertisements in so that the total cost is minimized and all cities are reached.
Algorithm: dynamic programming with suffixes
- Subproblem: minimum cost to reach all cities $c_i, c_{i+1}, ..., c_{n-1}$ for $0 \leq i \leq n - 1$
- Guessing: whether to
  - advertise to $c_i$ and not advertise to $c_{i+1}$
  - advertise to $c_{i+1}$ and not advertise to $c_i$
  - # choices: 2
• Recurrence:
  \[-dp[i] = \min\{p_i + dp[i + 2], p_{i+1} + dp[i + 3]\}\]
  \[-dp[n - 2] = \min\{p_{n-2}, p_{n-1}\}\]
  \[-dp[n - 1] = p_{n-1}\]
  \[-dp[n] = 0\]
  \- memoize

• Order: for \(i = n - 1, n - 2, ..., 2, 1, 0\)

• Solve original: \(dp[0]\)

Correctness: Right to left ordering guarantees no cycle to be in the recurrence.
Run Time: \(O(n)\) subproblems \(\times O(1)\) time/subproblem \(= O(n)\) time. Finding the minimum from two choices takes constant time.

(c) [5 points] Solution:
Summary: Cities \(c_{i-k}, ..., c_i, ..., c_{i+k}\) can be reached by advertising to \(c_i\) with a price of \(p_i\). We need to compute the subset of cities to run advertisements in so that the total cost is minimized and all cities are reached.
Algorithm: dynamic programming with suffixes
• Subproblem: minimum cost to reach all cities \(c_i, c_{i+1}, ..., c_{n-1}\) for \(0 \leq i \leq n - 1\)
• Guessing: \(p = \{0, 1, 2, ..., k\}\) are options
  \- advertise to \(c_{i+p}\) and not to any of \(k\) neighboring cities on each side
  \- \# choices: \(k+1\)
• Recurrence:
  \[-dp[i] = \min\{p_i + dp[i + k + 1], p_{i+1} + dp[i + k + 2], ..., p_{i+k} + dp[i + 2k + 1]\}\]
  \- for \(1 \leq z \leq k + 1, dp[n - z] = \min\{p_{n-z}, p_{n-z+1}, ..., p_{n-1}\}\]
  \- \(dp[n] = 0\)
  \- memoize
• Order: for \(i = n - 1, n - 2, ..., 2, 1, 0\)
• Solve original: \(dp[0]\)

Correctness: Right to left ordering guarantees no cycle to be in the recurrence.
Run Time: \(O(n)\) subproblems \(\times O(k)\) time/subproblem \(= O(nk)\) time. Finding the minimum from \(k + 1\) choices takes \(O(k)\) time.
Problem Set 5 Solution

Part B

Problem 5-3. [45 points] Halloween

(a) Submitted on alg.csail.mit.edu
(b) Submitted on alg.csail.mit.edu

(c) [5 points] Solution:

Summary: Using graphs, we need to kill three given ghosts with the shortest possible move sequence.

Algorithm: We create a graph $G$ with nodes named $(i, j, k)$ where $0 \leq i \leq l_1$, $0 \leq j \leq l_2$, $0 \leq k \leq l_3$. Then we draw the specified edges for each cases below and keep track of the moves to use. With all nodes and edges in $G$, we run BFS from $(0, 0, 0)$ to find the shortest path to $(l_1, l_2, l_3)$ and use the recorded moves to find the shortest move sequence.

- For any node $(i, j, k)$ where $i \neq l_1, j \neq l_2, k \neq l_3$, consider $s_1[i], s_2[j], s_3[k]$.
  - Case 1: all three moves different
    * $((i, j, k), (i + 1, j, k)), ((i, j, k), (i, j + 1, k)), ((i, j, k), (i, j, k + 1))$
  - Case 2: two moves same and one different
    * if $s_1[i] == s_2[j]$: $((i, j, k), (i + 1, j + 1, k)), ((i, j, k), (i, j, k + 1))$
    * if $s_2[j] == s_3[k]$: $((i, j, k), (i, j + 1, k + 1)), ((i, j, k), (i + 1, j, k))$
    * if $s_3[k] == s_1[i]$: $((i, j, k), (i + 1, j, k + 1)), ((i, j, k), (i, j + 1, k))$
– Case 3: all three moves same
  * \(((i, j, k), (i + 1, j + 1, k + 1))\)

• For \((l_1, j, k)\) where \(j \neq l_2, k \neq l_3\), we draw the specified edges for each cases below. (Similar logic applies to \((i, l_2, k)\) and \((i, j, l_3)\))
  – Case 1: two moves different
    * \(((l_1, j, k), (l_1, j + 1, k)), ((l_1, j, k), (l_1, j, k + 1))\)
  – Case 2: two moves same
    * \(((l_1, j, k), (l_1, j + 1, k + 1))\)

• For \((l_1, l_2, k)\) where \(k \neq l_3\), we draw \((l_1, l_2, k + 1)\). (Similar logic applies to nodes \((l_1, j, l_3)\) and \((i, l_2, l_3)\))

Correctness: Drawing all possible edges and make sure we are open to all possible paths. Dividing into three major cases handles all edge cases.

Run Time: \(V = O(l_1 \cdot l_2 \cdot l_3)\). For each node \(u = (i, j, k)\), \(Adj[u] \leq 3\). So \(E = O(l_1 \cdot l_2 \cdot l_3)\). BFS runs in \(O(V + E)\), which is \(O(l_1 \cdot l_2 \cdot l_3)\).

(d) [5 points] Solution:
The idea is the same as the one in part (c). But we pay special attention to the bounds and cases. We create a graph \(G\) with nodes named \((m_1, m_2, ..., m_{g-1}, m_g)\) where \(0 \leq m_i \leq l_i\) for \(1 \leq i \leq g\). Then we draw the edges by observing \(s_1[m_1], s_2[m_2], ..., s_g[m_g]\).

Say \(moves = set([s_1[m_1], s_2[m_2], ..., s_g[m_g]])\). For each \(move\) in \(moves\), we draw the edge \(((m_1, m_2, ..., m_{g-1}, m_g), (n_1, n_2, ..., n_{g-1}, n_g))\) where \(n_i = m_i + 1\) if \(s_i[m_i] == move\) and \(n_i = m_i\) otherwise. Keep track of the taken move also. For each node \(u = (m_1, m_2, ..., m_{g-1}, m_g)\), \(Adj[u] \leq g\) because we have \(|moves| = O(g)\) and \(Adj[u] = g\) in the worst case when all \(g\) moves are distinct. With all nodes and edges in \(G\), we run BFS from \((0, 0, ..., 0)\) to find the shortest path to \((l_1, l_2, ..., l_g)\) and use the recorded moves to find the shortest move sequence.

\(V = O(\prod_{i=1}^g l_i), E = O(g \prod_{i=1}^g l_i) = O(\prod_{i=1}^g l_i)\).

BFS runs in \(O(V + E)\), which is \(O(\prod_{i=1}^g l_i)\).

(e) [5 points] Solution:
The idea is the same as the one in part (d). But we pay special attention to the behavior of this thunderbolt move. We create a graph \(G\) with nodes named \((m_1, m_2, ..., m_{g-1}, m_g)\) where \(0 \leq m_i \leq l_i\) for \(1 \leq i \leq g\). Then we draw the edges by observing \(s_1[m_1], s_2[m_2], ..., s_g[m_g]\).

Say \(moves = set([s_1[m_1], s_2[m_2], ..., s_g[m_g]])\). Follow the edge-drawing-instructions in part (d). But during the iteration through \(moves\) set, if \(move\) is the thunderbolt move, we ignore other rules and draw the edge \(((m_1, m_2, ..., m_{g-1}, m_g), (m_1 + 1, m_2 + 1, ..., m_{g-1} + 1, m_g + 1))\).

Like in part (d), we continue to keep track of the moves to use when we draw the edges. With the minor change in edge initialization step, rest of the algorithm (i.e., BFS) and runtime analysis follows from part (d).

Runtime: \(O(V + E)\), or \(O(\prod_{i=1}^g l_i)\).