Solution of Fifth Homework

Han Wu, Yu Chen

1 Shortest Travel Sequence (10 points)

Two friends f_1 and f_2 plan to travel together. Each one has his/her preferred travel route. In this problem, we only consider the sequence of the tourist attractions. For example, f_1 wants to have a trip to visit BCAEF following the very accurate sequence. In addition, they do not care the tourist attractions inserted in their preferred sequences. Please design an algorithm to output the length of the shortest valid travel sequence for the two friends with given two travel sequences.

For example, the sequence of f_1 is ABCB and that of f_2 is DBC. Your algorithm should output 5, as ADBCB or DABCB are two valid shortest travel sequences and all are in length of 5.

[We are expecting: pseudocode, time complexity analysis and an informal justification that it is correct.]

- **Intuition:** To find the shortest valid travel sequence for the two friends, we need to find the longest shared travel sequence between the two friends. This is a typical problem of **Longest Common Subsequence(LCS)**. Thus, we can first find the longest shared travel sequence by **LCS**, denoted by l, then the answer will be the total length of the two friends' travel sequences minus l. Details are shown in algorithm 1.
- **Time Complexity:** $O(mn)$(m and n are the length of f_1 and f_2 respectively). The time complexity is mainly related to the process of **LCS**, thus it's $O(mn)$.

Algorithm 1 Shortest Travel Sequence(f_1, f_2)

```
m ← len(f1)
n ← len(f2)
f ← A (m + 1) × (n + 1) matrix        ▷ f[i][j] records the length of the longest common subsequence between f1[0 : i) and f2[0 : j)
for i in [1 : m] do                    ▷ Fill in the table
    for j in [1 : n] do
        if f1[i − 1] == f2[j − 1] then
            f[i][j] = f[i − 1][j − 1] + 1
        else
            f[i][j] = max(f[i − 1][j], f[i][j − 1])
return m + n − f[m][n]
```

2 Big Bang (10 points)

One interesting function introduced in Smartisan OS (a flavor android system) is *Big Bang*. It can break the text in the screen into several small pieces. Let's design an algorithm to implement this function. Assume that you have a dictionary of words, the task is to break a sequence of characters into pieces of words contained in the dictionary. For example, the dictionary is {alg, algorithm, is, interest, interesting, interested}. For the input of "algorithmisinteresting", your algorithm should output the smallest set of words: "algorithm", "is",

"interesting". If the sequence cannot be fully divided (e.g., "algithmisinteresting"), just output "Incomplete Dict!".

[We are expecting: pseudocode, time complexity analysis and an informal justification that it is correct.]

- **Intuition:** By bruteforce, we can find every valid decomposition of the input string and choose the smallest one. However, the time complexity of this method could be extremely high when there are many valid substrings in the input string which will result in large number of possible decompositions. Thus, we can memoize results of the subproblems. Suppose that the length of the input string is n, currently we are at index i and we know the results of previous subproblems, i.e. when the input strings are $s[0:0], s[0:1], \ldots, s[0:i-1]$, we know the corresponding minimum decomposition sets. Now we want to get the result of $s[0:i]$. We just need to iterate through previous subproblems' results and update current result. Details are shown in algorithm 2.
- **Time Complexity:** $O(n^2)$(n is the length of the input string). The initialization step and the output step both consumes $O(n)$ time, and the process of dp consumes $O(n^2)$. Thus, the overall time complexity is $O(n^2)$.

Algorithm 2 Big Bang(s, $dict$)

```
n ← len(s)
f ← Anarrayoflengthn + 1        ▷ f[i] records the size of the minimum decomposition set of s[0 : i − 1]
pre ← Anarrayoflengthn          ▷ pre[i] records the start index of the word ends in i
for i in [0 : n) do             ▷ Initialization, Note that f[0] == 0
    f[i + 1] ← n + 1
    pre[i] ← −1
for i in [1 : n] do             ▷ DP
    for j in [1 : i] do
        if f[i] == 1 then       ▷ Early Stop, Since we cannot find a better answer
            break
        if dict.contains(s[j − 1 : i − 1]) and f[i] > f[j − 1] + 1 then
            f[i] = f[j − 1] + 1
            pre[i − 1] = j − 1
if f[n] < n + 1 then            ▷ We can find a valid decomposition
    s ← a stack to save the words
    idx ← n − 1
    while pre[idx]! = −1 do
        s.push(s[pre[idx] : idx])
        idx = pre[idx]
    while s ≠ ∅ do
        output s.pop() + ' '
else
    output "Incomplete Dict!"
```

3 Shortest Path in Grid Space (10 points)

Let us consider the shortest path in the grid space. Given a $n \times n$ cost table G, each $G[i, j]$ indicates the positive travel cost in the corresponding grid cell $C[i, j]$ in i-th line and j-th column. Please design an algorithm to output the shortest path from the left-top corner, $C[0, 0]$, to the right-bottom corner, $C[n-1, n-1]$.

[We are expecting: pseudocode, time complexity analysis and an informal justification that it is correct.]
As shown in algorithm 3.We use $O(n^2)$ to traverse each cell to construct the graph, and then use Dijkstra algorithm to get the shortest path from the start to the end in $O((n^2)^2)$ *or* $O(n^2 \lg n)$, so the time complexity

of the whole algorithm is $O(n^4)$ *or* $O(n^2 \lg n)$.In the process of building the graph, we add an edge to all the four directions that can pass, so the shortest path we get through Dijkstra algorithm is correct.

Algorithm 3 SPGP($grid[n][n]$)

```
G ← initial a empty graph
for i in 0..n do
    for j in 0..n do
        id ← i * n + j
        if i − 1 ≥ 0 then
            G.addUndirectedEdge(id, id − n)
        if i + 1 < n then
            G.addUndirectedEdge(id, id + n)
        if j − 1 ≥ 0 then
            G.addUndirectedEdge(id, id − 1)
        if j + 1 ≥ n then
            G.addUndirectedEdge(id, id + 1)
return Dijkstra(G, 0, n^2 − 1)
```

4 Fish fish eat eat fish (10 points)

Plucky the Pedantic Penguin enjoys fish, and he has discovered that on some days the fish supply is better in some lake and some days the fish supply is better in other lake.

He has access to a 2-D table S, where $S[i, j]$ is the number of fish he can catch in Lake j on day i. Assume that $S[i, j]$ is positive integer for any $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Plucky's goal is to have as many fish as possible at the end of day n. If he happens to be at Lake x on day i and wants to be at Lake y on day $i + 1$, he may pay $C(x, y)$ fish to a polar bear who can take him from Lake x to Lake y overnight; the same is true if he wants to go from Lake y back to Lake x. Here $C(x, y) > 0$ is a positive integer for any $x, y \in [1, m]$ and $x \neq y$. Assume that before day 1 begins, Plucky is at Lake x, and he has zero fish.

Plucky will save all the fish he gets until the end of day n (at which point he will feast), but the polar bear does not accept credit. So Plucky must have the fish on hand in order to pay the polar bear; he must pay *before* he travels.

In this question, you will design a dynamic programming algorithm that finds the maximum number of fish that Plucky will have on day n. Do this by answering the two parts below. [hints: You can first try to solve a problem of only two lakes.]

(a) (5 points) What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between a problem and the sub-problems? What is the base case for this recursive relationship? Justify your answer.
[We are expecting: a formal definition of your sub-problems, as well as a recursive relationship that they satisfy. We are also expecting an informal justification that your recursive relationship is correct given your definitions.]
Sub-problem: The most fish you can get on day i at lake j.
Relationship: We use $dp[i][j]$ to represent the above sub-problem, and then we can express our problem by $dp[i][j] = \max_{k \in [1,m] \text{ and } dp[i-1][k]-C[k][j] \geq 0} \{dp[i-1][k] - C[k][j] + S[i][j]\}$.
Justification: The above formula shows that the maximum number of fish that can be obtained by being in the j-th lake on day i can be accessed in two ways: one is that penguin is in the j-th lake on day $i-1$, so we don't need to pay polar bears to move; the other is that penguin is in Lake k on day $i-1$, and we need to pay $C(k, j)$ to move from Lake k to Lake j.Therefore, we can use $dp[i][j]$ to express the correct answer of any valid i and j.(Here we assume that $C(k, k) = 0$).

(b) (5 points) Write pseudocode for a dynamic programming algorithm that takes as input of fish supply table S, travel cost table C, m, and n, and returns the maximum number of fish that Plucky can eat on day n.
[We are expecting: the pseudocode, a brief explanation (one or two sentences) about why it works, and explain (also one or two sentences) that its running time.]
As shown in algorithm 4. We need to fill in the solutions of $m * n$ subproblems. For each subproblem, we need to make m times comparison to get the best, so the final complexity is $O(nm^2)$

Algorithm 4 Fish(S, C)

$dp \leftarrow$ Initialize a two-dimensional array to store answers of sub-problems
for $i \in [1, m]$ **do** ▷ Fill the answers on the first day in each lake
    $dp[0][i] = S[0][i]$
for $i \in [2, n]$ **do** ▷ Fill solution array one by one
    for $j \in [1, m]$ **do**
        $dp[i][j] = \max_{k \in [1,m] \ and \ dp[i-1][k]-C[k][j] \geq 0} \{dp[i-1][k] - C(k, j) + S[i][j]\}$
return $\max_{i \in [1,m]} \{dp[n][i]\}$ ▷ Get and return the maximum result