CSOR W4231 Analysis of Algorithms I - Spring 2023

Homework 3

Tong Wu, tw2906

Problem 1

1.1 Main Idea

The problem ask to find a maximum decrease in the distance between two fixed cities s and t in the network if a new road e' has been added into the network from the list E'. Assume the new road e' connected two cities u and v, $e' = (u, v)$, and the new shortest path will follow the road e', then the problem can be simplified to the path $s \to u \to v \to t$. In this case, the idea is to first run the Dijkstra's algorithm in the original road network to find the shortest path. Then in a for loop, run the Dijkstra's algorithm in the new network with the each new created road to find the new shortest path and compare with the original path to calculate the difference. Finally output the maximum decrease and the added road. The detailed steps is below:

1. Run the Dijkstra's algorithm in the original road network to find the minimum path `d_st`
2. For each potential new road $e' \in E'$, create a new road e' to the graph G, then run the Dijkstra's algorithm again to find the shortest path distance `d_st_new`. Calculate the decrease in distance `derease_dis`, if the `derease_dis` is larger then previous one `max_decrease`, then update the `max_decrease` and the road `added_road`.
3. Output the `max_decrease` and `added_road`

1.2 Pseudo Code

```
def dijkstra_max_decrease(G=(V,E,l), E`, s, t):
    # Run the Dijkstra algorithm in the original road network to find the minimum path
    d_st = Dijkstra(G, s)
    # Initialise the variable
    max_decrease = 0
    added_road = NULL
    for e` in E`:
        # Add road to the original graph
        G` = add_road(G, e`)
        d_st_new = Dijkstra(G`, s)
        decrease_dis = d_st[t] - d_st_new[t]
        if decrease_dis > max_decrease:
            max_decrease = decrease_dis
```

```
            added_road = e`
        end if
    end for
    return added_road
```

In the pseudo code, the input of the function is the original road network graph G and the edge list E', and two specificized cities s and t. The called function `Dijkstra(G, s)` is mentioned in lecture with returned an shortest distance array `dist[]`. In the for loop, use the terminal city t to determine the array to find the shortest distance `d_st[t]`. Finally output the road `added_road`

1.3 Correctness

Assuming that the algorithm can find m potential new roads in $e' \in E'$.

Base case: $m = 1$

If there is only 1 potential new road, then the algorithm add the new road to the road network graph G, and calculate the new shortest path distance between cities s and t. This is correct since there is only one road to choose from.

Induction: Try to prove that $m = n + 1$ is true.

Induction hypothesis: The algorithm correctly finds the best road e' for E'

When the algorithm running to e'_{n+1}, it will add the road to the graph G and compare the current maximum decrease in distance. Under this situation it will be two cases:

1. The decrease in distance after add the road e'_{n+1} is greater than the current maximum decrease in distance. Then, the algorithm will updates `max_decrease` and `added_road` .
2. The decrease in distance after add the road e'_{n+1} is lower than or equal to the current maximum decrease in distance. Then the algorithm will not update `max_decrease` and `added_road` .

In both cases, the algorithm can correctly find the best road. Therefore $n + 1$ is also correct, which prove the correctness.

By induction, the algorithm is correct for all number of potential new roads.

1.4 Running Time

Assuming that the function `Dijkstra(G, s)` use binary heap implementation, which has running time $\mathrm{O}((|V| + |E|)log(|V|))$. The algorithm will call function `Dijkstra(G, s)` once for each potential new road in E', so the total running time should be:

$$T(n) = \mathrm{O}(|E| \times (|V| + |E|)log(|V|))$$

Problem 2

2.1 Main Idea

The problem ask to find an algorithm to calculate the minimum number of truck to transfer a set of boxes with preserving order and specific weight. Each box i has integer weight $w_i \geq 0$ and each truck has maximum loading capacity $W \geq 0$. The main idea to solve this problem is using the greedy algorithm, which makes myopic decision at every step in order to gain the maximum reward. Assume that each box weight must smaller than the truck maximum loading capacity $w_i \leq W$. The input of the algorithm is the maximum loading capacity W of each truck, and a class for all boxes which containing the weight w_i, and the output of the algorithm is the number of used truck. The algorithm will load as many boxes to the current truck as possible before loading the next truck, which will minimize the number of used truck. The detailed step is below:

1. Initialize the variable `trucks` shows the used trucks, and the `remain_load` shows the remaining load capacity of the current truck.
2. Create a for loop, for each box, determine whether the current truck can load this box.
    1. If the current truck can load this box, update the remaining loading capacity of the current truck.
    2. If the current truck cannot load this box, create a new truck to load this box, and let the new truck be the current truck.
3. Output the number of used trucks.

2.2 Pseudo Code

```
def greedy_trucks(W, box):
    trucks = 1
    remain_load = W
    # Greedy algorithm
    for box.weight in box:
        # Determine if the current truck is capable of carrying a load
        if (remain_load >= box.weight):
            remain_load -= box.weight
        else:
            trucks += 1
            remain_load = W - box.weight
        end if
    end for
    return trucks
```

In this code, the number of used trucks `trucks` is initialized to `1` at the beginning since there must use one truck. The remaining load of the current truck `remain_load` is initialized to `W` which is set by the problem. Each round of loop the `remain_load` will be updated.

2.3 Correctness

Assuming that the algorithm is used to load m boxes with weight $w_1, w_2, \ldots, w_m$

Base case: $m = 1$

If there is only one box to load, the algorithm will load the box to the first truck, then the remaining loading capacity will be updated. Since there must have one truck to load the boxes, so the base case can be proved its correctness.

Induction: Try to prove that $m = n + 1$ is true.

Induction hypothesis:

When the algorithm is used to load $n + 1$ boxes, there will be two cases:

1. The remaining capacity of the current truck can hold the next box. In this case, the algorithm load this box to the current truck and update the remaining loading capacity. The number of used truck will keep unchanged.
2. The remaining capacity of the current truck cannot hold the next box. In this case, the algorithm will append a new truck to be the current truck and load this box, then update the remaining loading capacity. The number of used truck will be added by 1.

In both cases, the decision is optimal since the algorithm is using the greedy algorithm to find the maximum number of boxes can be loaded in each truck obeying the rule that preserving the order of the boxes. Hence in both cases, the algorithm minimizes the number of used truck to load $n + 1$ boxes. Therefore, the correctness of $m = n + 1$ has been proved.

By induction, the algorithm is correct for all number of boxes.

2.4 Running Time

Since there is a for loop in the algorithm, so the running time should be:

$$T(n) = \mathrm{O}(n)$$

Problem 3

3.1 Main Idea

The problem ask to find an algorithm to calculate the minimum total cost and a set of libraries where copies of books must be placed to achieve the minimum total cost. It can be solved by dynamic programming approach. Define the $OPT(j)$ as the minimum total cost to place the book in some of the libraries. According to the problem, the copy must stored in library L_n. Therefore the equation can be got:

$$OPT(j) = min_{1 \leq i < j}\{c_j + OPT(j-1) + (j-i)\}$$

Where c_j is the cost to store the cost to store the copies in some libraries, and $(j-i)$ is the user delay search copies from library L_i to L_j. The boundary condition set to $OPT(0) = 0$, and let M be an array to store the values. The detailed step is below:

1. Initialize the array `m[]` with size $n+1$ to store the minimum total cost for libraries. Set `M[0] = 0`
2. For each library, calculate the all user delay, then iterate through all possible libraries to find the minimum total cost. Finally, update `m[]` .
3. Backtrack through the array to find the libraries where the book copies should be placed, and store them in a set.
4. Output the minimum total cost and the set of libraries which copies should be placed in.

3.2 Pseudo Code

```
def min_cost(n, c):
    # Initialize m with size n+1
    m = [0,0,0,...,0] with total n+1 terms
    for i=1 in n:
        tmp = c[i] + (i-j)
        # Compute delay and find minimum
        for j in range i:
            // Recurrence OPT
            min_cost = min( (m[j] + c[i] + (i-j)), tmp)
        end for
        m[i] = min_cost
   end for
   return m[n]

def opt(j):
    # Boundary condition
    if (j==0) then return
    else:
        if (m[j]==c[j]+m[j-1]):
```

```
            print(library[j])
            opt(j-1)
        else:
            opt(j-1)
        end if
    end if
```

The input `c` is an array contains all costs to store a copy in each library. `n` is the total number of libraries. The function `min_cost()` returns the minimum cost, and the function `opt()` print out all libraries need to store copies.

3.3 Correctness

The $OPT(j)$ represents the minimum total cost for placing books in some libraries. Then define the $OPT(j)$ recursively by considering all possible libraries before the library L_j.

$$OPT(j) = min_{1 \leq i < j}\{c_j + OPT(j-1) + (j-i)\}$$

For the expression: c_j means the cost to store the copies in library L_j, $(j-i)$ representing the user delay from library L_i search to L_j. With defining the recurrence, the dynamic programming solution can be created to calculate the minimum total cost of placing books in libraries by iteratively solving several subproblems.

3.4 Running Time

In the code, there is a for loop nested in a for loop. The outside loop count from 1 to n and inner loop iterate from 1 to i with the backtrack. Therefore, the running time should be:

$$T(n) = \mathrm{O}(n^2)$$

3.5 Space Complexity

The array contains the minimum cost store all values for n libraries. Hence the array takes $\mathrm{O}(n)$ space.

The array contains libraries needs $\mathrm{O}(n)$ space to store all n libraries at the worst case.

In general, the total space complexity is $\mathrm{O}(n)$

Problem 4

4.1 Main Idea

The problem ask to find an algorithm to calculate the minimum distance of the distance between two time-series signal functions mapped using non-decreasing function, with values $\{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_m\}$. Dynamic programming approach can be used to solve this problem. Create an matrix M to obtain all the values of the minimum distance. The fill the matrix, the algorithm will start from the first row and fill the value from left to right, `M[0][j]`, until the last term `M[n][m]`. Define the $OPT(i, j)$ represents the minimum distance between two functions. Since the map is non-decreasing, so the minimum distance calculation for the next element `[i+1]` can only refer the last and the current distance, which are `[i-1][j-1]` and `[i-1][j]`, find the minimum of this two value and plus the current cost. The expression will be written:

$$OPT(i, j) = min\{OPT(i-1, j),\ OPT(i-1, j-1) + |a[i] - b[j]|\}$$

Where $|a[i] - b[j]|$ is the cost of mapping. The boundary condition set to $OPT(i = 0) = 0$, and let M be an array to store the values. The detailed step is below:

1. Create a table `m[][]` to store all the cost for each pairs. The size of the table should be $(n + 1) \times (m + 1)$
2. For each element in the table, calculate the distance of the specific pair element, then calculate the expression $min\{OPT(i-1, j),\ OPT(i-1, j-1) + |a[i] - b[j]|\}$. Store into the table `m[i][j]`.
3. Output the minimum distance which stored at the bottom-right corner of the table `m[n][m]`

4.2 Pseudo Code

```
def min_distance(a, b, n, m):
    M = [0,...,0][0,...,0] with size (n+1)*(m+1)
    for i in range(n+1):
        for j in range(m+1):
            # Calculate the cost of the current pair
            cost = abs(a[i-1]-b[j-1])
            # Calculate the minimum distance for now
            M[i][j] = cost + min(M[i-1][j], M[i-1][j-1])
        end for
    end for
    return M[n][m]

def OPT(a, b, n, m):
    # Boundary condition
```

```
    if n == 0:
        print('0')
    else:
        print(min_distance(a, b, n, m))
```

In the code, the input of two functions are: `a` and `b` are arrays that contains all values of two functions. `n` and `m` are the number of two functions values. The output `M[n][m]` is the minimum distance calculated by the algorithm.

4.3 Correctness

$$OPT(i, j) = min\{OPT(i-1, j),\ OPT(i-1, j-1) + |a[i] - b[j]|\}$$

The recurrence relation is defined by the $OPT(i, j)$ function, which calculates the minimum distance between the `a[i]` element and `b[j]` element.

$OPT(i-1, j)$ represents the minimum distance between `a[i-1]` element and the `b[j]` element.

$OPT(i-1, j-1)$ represents the minimum distance between `a[i-1]` element and the `b[j-1]` element.

The recurrence relation for $OPT(i, j)$ takes the minimum of these two options. It chooses the option with the smallest cost. This approach ensures that we find the minimum distance between `a[i]` and `b[j]` .

By recursively computing the values of $OPT(i, j)$ for all `i` and `j` , the dynamic programming approach calculates the minimum distance between the function a and b.

4.4 Running Time

Since the algorithm need to calculate all minimum distance through the table with size $(n+1) \times (m+1)$, the function `min_distance()` has a for loop with a nested for loop, one iterates from 1 to n and another iterates from 1 to m. Therefore, the running time of this algorithm should be:

$$T(n) = \mathrm{O}(n \times m)$$

4.5 Space Complexity

The table `M` which stores all minimum distance has size $(n+1) \times (m+1)$, hence it spends space complexity $\mathrm{O}(n \times m)$.

Problem 5

5.1 Main Idea

The problem ask to find an algorithm to calculate the minimum cost of breaking a string with n unit of length into $m+1$ pieces. Assume that the cuts are sorted, which the next cut must at the second part of the split. The dynamic programming approach can solve this problem. Define the function $OPT(i,j)$ representing the minimum cost of break the string from position i to position j, with the cutting point k, where $i<k<j$. Each subproblem will cut the string once and calculate the current cumulative cost, which should be the cost of copying the string with length $j-i+1$, the cost of previous part $OPT[i][k]$ and the cost of the subsequent part $OPT[k+1][j]$. Therefore, the expression of the recurrence can be expressed as:

$$OPT(i,j) = min_{i\leq k<j}\{OPT(i,k)+OPT(k+1,j)+(j-i+1)\}$$

The boundary condition of the algorithm is $OPT(i,i-1)=0$, which represent the cutting of a substring with length smaller than 0 is not acceptable hence its minimum cost is 0. Let M be an array to store the values. The detailed step is below:

1. Create the table `M` to store the minimum cost to cut the string with starting position at `i` and ending position at `j`. The size of table `M` should be $(n+1)\times(n+1)$ since the length of the original string is n.
2. For each possible substring pairs, calculate the minimum cost of breaking the substring into `m+1` pieces using dynamic programming.
3. Use nested for loop to iterate all possible substrings and then compute the value of minimum cost using recurrence relations.
4. Output the minimum cost of the original string split into $m+1$ pieces, which should stored in `M[1][n]`

5.2 Pseudo Code

```
def min_cost(n, m):
    M = [0, 0, ... , 0][0, 0, ..., 0] in size (n+1)*(n+1)
    for length=1 in range(m+1):
        for i=1 in range(n-length+1):
            j = i + length
            # Set the initial value in matrix to infinity
            M[i][j] = 'inf'
            for k in range(i, j):
                current_cost = M[i][k] + M[k+1][j] + (j-i+1)
                M[i][j] = min(M[i][j], current_cost)
            end for
        end for
    end for
    return M[1][n]
```

```
15
16 def OPT(n, m):
17     if m == 0:
18         print('0')
19     else:
20         print(min_cost(n, m))
```

5.3 Correctness

$$OPT(i,j) = min_{i \leq k < j}\{OPT(i,k) + OPT(k+1,j) + (j-i+1)\}$$

The recurrence relation states the minimum cost of cutting the substring from i to j with the cutting point k, compute recursively by breaking the substring into two smaller substrings and computing the minimum cost of breaking each of them. The minimum cost of two substring $OPT(i,k) + OPT(k+1,j)$ and the cost of copying $j-i+1$ are summed to form the final cost.

The recurrence relation split the problem into subproblems to solve in bottom-up approach. By recursively computing the values of $OPT(i,j)$ for all `i` and `j`, the dynamic programming approach calculates the minimum cost of splitting a string with length n into $m+1$ pieces.

5.4 Running Time

Generate the matrix `M` requires $\mathrm{O}(n^2)$ of running time since `M` has size $(n+1) \times (n+1)$. The nested for loop to calculate the minimum cost has three loops, the outer loop iterates all possible substring length, the middle loop iterates all possible starting point, and the inner loop iterates all possible cutting point. Each loop costs $\mathrm{O}(n)$ running time, so the nested loop costs $\mathrm{O}(n^3)$ running time. In general the algorithm has running time:

$$T(n) = \mathrm{O}(n^3)$$

5.5 Space Complexity

The table matrix `M` has size $(n+1) \times (n+1)$, so the overall space complexity should be $\mathrm{O}(n^2)$