COT 5405 Analysis of Algorithms Fall 2021 Assignment #2

UFID: 96703101
Name: *Vyom Pathak*
Instructor: Professor Alin Dobra
Due Date: November 19, 2021

Problem 1

Weighted approximate common substring [50]

Problem Definition

Given two input strings, not necessarily of the same length, based on alphanumeric characters [A-Z], determine the best common substring. A substring is a contiguous sequence of characters within a string, and the score that determines the best substring is defined as the sum of the weights w_l for each character in the sequence (i.e. w_a is the weight of matching letter A) and a penalty $-\delta$ for each mismatch (negative penalty term to drive down the score). In your experiments, consider situations in which w_l=1 and $\delta = 10$ and in which w_l is proportional to the frequency of the letter in English and δ takes values between the smallest and the largest weight (multiple experiments for 10 intermediate values). Note: you are now allowed to add gaps in the solution, i.e. both matched substring have the same length. Example: inputs 'ABCAABCAA'and 'ABBCAACCBBBBBB'. The substring 'CAABC'that starts at position 3 in the first string and position 4 in the second, has a score $2*w_c$+$2*w_a$-δ since the B is mismatched in the second string.

Solution

Algorithm

Assumption: Considering there is no gap between two characters possibles in a given string. The algorithm can be devised as a dynamic programming problem where the bellman equation can be formed in the following way: If the current character is matching in both the strings, add the weight of the current character to the previous best string. Otherwise, take the max of penalty [for selecting a mismatching character] or start a new BWACS. The algorithm to find the best weighted approximate common substring is given in Algorithm 1.

Analysis

Proof of Correctness
The algorithm always finds the best weighted approximate common substring. We prove this by claiming that the memory table is computed correctly, and then use induction.
Claim: For all $0 \leq i \leq n$ and $0 \leq j \leq m$, $memo[i, j] = WACS[A^i, B^j]$.
Proof: Here, base condition is that when $i = 0$ and $j = 0$, $memo[i, j] = 0$ which is correct. Let's assume without any loss of generality that the claim holds true for all entries $memo'[i, j]$ such that for some entry $memo[i, j]$; where $i' + j' < i + j$. Let $Sol' = WACS[A^i, B^j]$
If $A[i] = B[j]$, then the last character of X' must be the current character weight plus the last weight value for the A^{i-1} and B^{j-1}. So from induction hypothesis $memo[i, j] = memo[i - 1, j - 1] + w_l$, we compute the entry correctly.
Otherwise, let x be the last character of X'. If $x \neq A[i]$, then X' must be weighted approximate common substring with a penalty [δ] from the previous value i.e. A^{i-1}, and B^{j-1} or we do not extend the current solution with a penalty i.e. use 0 and start a new best weighted approximate common sub-string. Thus, it can be expressed as $WACS[A^i, B^j] = max\{memo[i - 1, j - 1] - \delta, 0\}$, this gives the feasible and correct solution.

Algorithm 1 Function for finding the best weighted approximate common sub-string

```
1: Input: String A, String B, penalty value δ, memo as a memory table with zero values.
2: Output: Prints the best weighted approximate common sub-string with the weight value.
3: function FindWACS(A, B, δ, memo[MAX, MAX]):
4:     maxWeight ← 0;
5:     for i in range(1, A.length()) do
6:         for j in range(1, B.length()) do
7:             if A[i − 1] = B[j − 1] then
8:                 memo[i, j] ← memo[i − 1, j − 1] + w_l;
9:             else
10:                memo[i, j] ← max{0, memo[i − 1, j − 1] − δ};
11:                maxIndex(i, j) ← i, j;                    ▷ Store the max valued indices
12:                maxWeight ← max{maxWeight, memo[i, j]};
13:            end if
14:        end for
15:    end for
16:    i, j ← maxIndex(i, j);
17:    sol ← ' ';
18:    while (i > 0 and j > 0) and (memo[i, j] > 0) do
19:        sol ← A[i − 1] + sol;
20:        i − −;
21:        j − −;
22:    end while
23:    print('The Weight of the BWACS is : ', maxWeight);
24:    print('The BWASC is : ', sol);
25: end function
26: findWACS('ABCC', 'ABCCDA', 10, memo);
```

Running Time

The algorithm finds the BWACS in $O(n \cdot m)$.

Proof: Here, we loop through each character of string A, and for each of those character we loop through all the characters of string B. Thus, n being the length of string A, and m being the length of string B, the overall time-complexity of the solution is $O(n \cdot m)$. We store a 2-D table of length n and width m, and thus the overall space-complexity is $O(n \cdot m)$.

The figure 2 shows the analysis of runtime as a function of size of the string. The evaluation is done for the same in the following way.

1. The first experiment was run on 1000 values [size of string] where the penalty was taken as -10 and the weights of each character was taken as 1.

2. The second experiment was run on 1000. values [size of string] where the weight of each character was taken as the frequency of that character occurring in the English language [2] and the penalty value is taken randomly between the maximum and minimum word count. Also, the following figure 1, 10 intermediate values shows how increasing the penalty value for the weight case; the weight of the common substring decreases as well as the size of the string decreases. The figure 1 shows the selection of $WACS$ for strings 'ABCAABCAA'and 'ABBCAACCBBBBBB'.

```
The penalty is : -5
The weight of the Common Substring is : 157
The Common Substring is : AABCAA

The penalty is : -6
The weight of the Common Substring is : 156
The Common Substring is : AABCAA

The penalty is : -10
The weight of the Common Substring is : 152
The Common Substring is : AABCAA

The penalty is : -12
The weight of the Common Substring is : 150
The Common Substring is : AABCAA

The penalty is : -16
The weight of the Common Substring is : 146
The Common Substring is : AABCAA

The penalty is : -18
The weight of the Common Substring is : 144
The Common Substring is : AABCAA

The penalty is : -23
The weight of the Common Substring is : 139
The Common Substring is : AABCAA

The penalty is : -27
The weight of the Common Substring is : 135
The Common Substring is : AABCAA

The penalty is : -33
The weight of the Common Substring is : 129
The Common Substring is : AABCAA

The penalty is : -36
The weight of the Common Substring is : 126
The Common Substring is : AABCAA

The penalty is : -56
The weight of the Common Substring is : 119
The Common Substring is : BCAA
```

Figure 1: Intermediate penalty value for the character frequency weight case and respective WACS

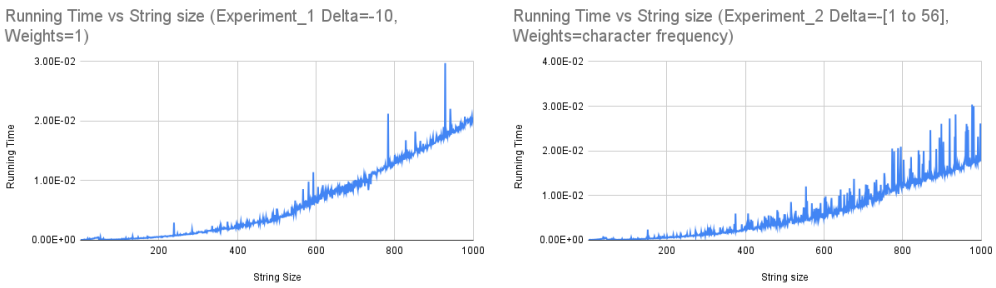

Figure 2: Experiment 1 and 2 for checking the runtime of the WACS algorithm as a function of size of string

Problem 2

Interval-based constant best approximation [50]

Problem Definition

Given a set of N points (x_i, y_i) with integer values for x_i between 1 and M and real values for y_i, find a partitioning of the interval [1,M] into contiguous intervals such that the error of approximating points in each interval element by the average value of y in the interval is minimized. You need to add a penalty factor proportional to the total number of intervals the solution has. For example, if you have $x \in [1, 100]$ and you partition the X dimension in intervals $[1 - 10], [11, 20], \ldots, [91, 100]$ the penalty is $10 * \delta$.

Solution

Algorithm

Assumption: For each point, the x-coordinate value is unique. The problem is a variant of the segmented least-squared with a special value for how to control the y values. Here, as the error function we consider the y_avg as the value to be used to subtract from the original y value for each interval. It can be shown as follows:

$$Errors[i, j] = \sum_{i=1}^{n} (y_i - y\_avg_{ij})^2 \ [for\ each\ j\ from\ i\ to\ n].$$

We calculate this effectively by using statistics i.e. sum of squares minus the square of sums which turns the above formula into the following form:

$$Errors[i, j] = \frac{\sum_{k=i}^{j} x_k^2}{n} - \left(\frac{\sum_{k=i}^{j} x_k}{n} \right)^2$$

We can effectively pre-compute the square of sums and sum of squares in the running like cumulative sums and while calculating the errors for a range, we can find that range's sum of squares and square of sums by just subtracting by the index values for the cumulative stored in those indices. Thus, this reduces the error computation from $O(n^3)$ to $O(n^2)$. For the rest of the algorithm, the bellman equation remains the same which can be expressed as follows:

$$OPT(j) = min_{1 \leq i \leq j}(Errors[i, j] + delta + OPT[i - 1]); \tag{1}$$

Here, the problem can be understood as a linear regression model but for segments and to avoid any kind of over-fitting of too many segments we introduce the penalty which is essentially a regularization parameter. The algorithm for finding interval based constant best approximation is given in Algorithm 2. The Algorithm 3 is a helper function which pre-computes the errors for all the points in a given range using sum of squares and square of sums optimized by a running cumulative sum.

Algorithm 2 Function for finding the interval-based constant best approximation

```
1: Input: Point Array points with x and y coordinates, count of total number of points as num_points,
   penalty term delta, Error matrix with pre-computed errors.
2: Output: Prints the number of segment, total cost of the solution, and prints the points between the
   segment indices.
3: function FINDICA(points, num_points, delta, Errors[MAX, MAX]):
4:     opt_seg ← [ ];
5:     solution ← [ ];
6:     for j in range(1, num_points + 1) do
7:         temp ← ∞;
8:         temp_ind ← 0;
9:         for i in range(1, j + 1) do
10:            bellman_eq ← Errors[i, j] + solution[i − 1];
11:            if bellman_eq < temp then
12:                temp ← bellman_eq;
13:                temp_ind ← i;
14:            end if
15:        end for
16:        solution[j] ← temp + delta;
17:        opt_seg[j] ← temp_ind;
18:    end for
19:    point_indices_stack ← φ;
20:    i ← num_points;
21:    j ← opt_seg[num_points];
22:    while i > 0 do
23:        point_indices_stack.add(i);
24:        point_indices_stack.add(j);
25:        i ← j − 1;
26:        j ← opt_seg[i];
27:    end while
28:    print('Cost of the Optimal Solution is : ', solution[num_points]);
29:    print('Total number of segments are : ', point_indices_stack.size()/2);
30:    while point_indices_stack ≠ φ do
31:        i_x ← point_indices_stack.pop();
32:        j_y ← point_indices_stack.pop();
33:        print('The segments between indices: ', i_x, ' ', j_y, 'with error values: ', Errors[i_x, j_y]);
34:        while i_x < j_y do
35:            print('( ', points[i_x − 1].getX(), ', ', points[i_x − 1].getY(), ' )');
36:            i_x + +;
37:        end while
38:    end while
39: end function
40: points.SortByX(); ▷ Sort by x-coordinate values
41: calculateErrors(points, num_points);
42: findICA(points, num_points, delta, Errors);
```

Algorithm 3 Helper Function to calculate the errors

```
1: Input: Point Array points with x and y coordinates, count of total number of points as num_points.
2: Output: Stores the Errors in a global error matrix Errors.
3: function CALCULATEERRORS(points, num_points):
4:     prefix_sum_y ← [ ];
5:     prefix_sum_sqry ← [ ];
6:     for j in range(1, num_points + 1) do
7:         pt ← points[j − 1].getY();
8:         prefix_sum_y[j] ← prefix_sum_y[j − 1] + pt;
9:         prefix_sum_sqry[j] ← prefix_sum_sqry[j − 1] + (pt · pt);
10:        for i in range(1, j + 1) do
11:            if i = j then
12:                Errors[i, j] ← 0;
13:            else
14:                interval ← j − i + 1;  ▷ Calculate the SSE error for the points between the range i and j
15:                y_sum_ij ← prefix_sum_y[j] − prefix_sum_y[i − 1];
16:                sqry_sum_ij ← prefix_sum_sqry[j] − prefix_sum_sqry[i − 1];
17:                Errors[i, j] ← (sqry_sum_ij/interval) − ((y_sum_ij/interval) ∗ (y_sum_ij/interval));
18:            end if
19:        end for
20:    end for
21: end function
```

Analysis

Proof of Correctness

The proposed algorithm correctly computes $OPT[j]$ for each $j = 1, 2, 3, \ldots, n$.

Proof: *Here, we assume that the FindICA is a recursive function.* Be definition, $OPT[0] = 0$. Now, taking some $j > 0$, and suppose by the way of induction that the algorithm at i correctly computes $OPT(k)$ for all $k < j$. Now by the induction hypothesis, we know that the algorithm $FindICA(p(j))$ will give the $OPT(p(j))$, and also the algorithm $FindICA(j-1)$ at point $j-1$ gives the solution $OPT(j-1)$ by checking every i from 1 *to* $j-1$ in equation (1). Hence, from equation (1) we are able to get the solution to $OPT(j)$ as follows:

$$OPT(j) = min_{1 \leq i \leq j}(Errors[i, j] + delta + FindICA[i-1]);$$
$$OPT(j) = FindICA(j);$$

Hence, the proposed algorithm correctly computes $OPT[j]$ for each $j = 1, 2, 3, \ldots, n$.

Running Time

The running time of the proposed algorithm is $O(n^2)$, where n is the number of points.

Proof: Here, first the points are sorted based on the x-coordinate values which essentially takes $O(n \cdot log(n))$ using the merge-sort algorithm. Then, the *calculateErrors*() function runs from 1 to number of points for the outer-loop, and for each point we run the loop with all the other points to find the interval-based error using the above mentioned formulae. We use the prefix sum calculation to efficiently calculate the sum and square of sums for each interval. Then, for finding the error is just $O(1)$. Thus, this function runs in $O(n^2)$ time.

For, the function $findICA()$, we iterate through all the points and check if the interval can be added or not based on the given bellman equation. This takes $O(n^2)$ time. After that, we print the interval coordinates from the segment indices which basically takes $O(n^2)$. Thus, the overall running time of this function is also,

$O(n^2)$.
Therefore, the overall running time for all of these functions combined is $O(n^2)$.

The figure 3 shows the analysis of runtime as a function of size of the string. The analysis is done in a sense that the number of points are increased and accordingly the y value and penalty ($delta$) is set. We can clearly see from the figure that the runtime of the proposed algorithm is in-fact $O(n^2)$. Also figure 3 shows that, for the recursive function; even if the time complexity of the algorithm is same, because of the recursive nature it takes more time than the iterative version to execute.

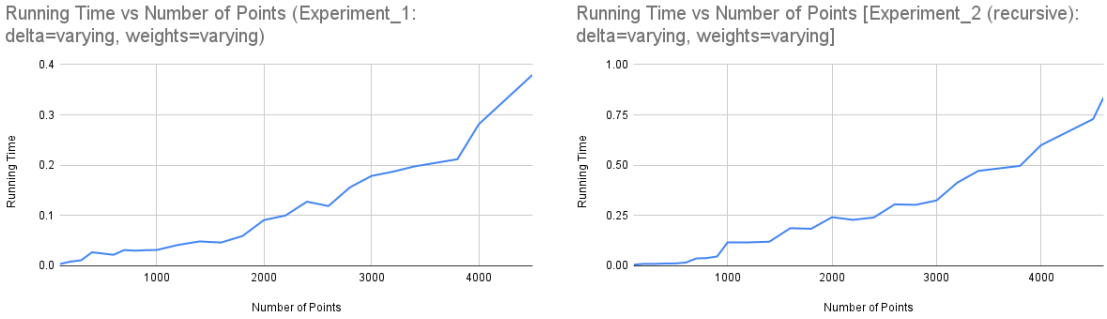

Figure 3: Experiment 1 for checking the runtime of the iterative ICA algorithm, and Experiment 2 as the recursive version. Both as a function of number of points

The overall space complexity is $O(n^2)$ as we store the error matrix and the solution segments in 1-dimension. While not using memoization, we store an 1-D array of size the number of segments which is $O(n)$. However, even after using the memoized version of the algorithm it does not budge with respect to the runtime of the algorithm i.e. the runtime of the memoized version of the algorithm is still $O(n)$ as there is no overlap in finding the segments, thus the space complexity is still more $O(n)$ in addition to the recursion stack memory. Here, the figure 4 shows the analysis of space as a function of number of points for the iterative as well as the memoized version of the function $findICA()$, which clearly states that the space used in the memoized version is more because of the recursion stack [$0.94GB > 0.88GB$]. The memory was calculated using the JProfiler Tool [1].

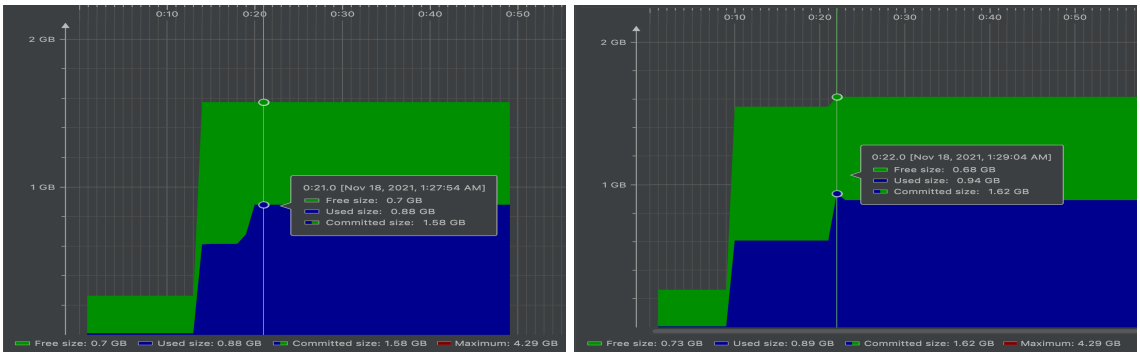

Figure 4: Comparision of memeory utilization for iterative [left] and memoized [right] version for the proposed algorithm

Problem 3

BONUS [30]

Problem Definition

Generalize problem 2 to a 2D grid for the input, i.e. the input is of the form x_i, y_i, z_i with $x_i, y_i \in [1..M]$ and z_i a real number. You must both develop the theory and implement this version of the algorithm and test it similarly to the setup above.

Solution

The proposed problem essentially cannot be generalized from the previous 2D case because no-optimal sub-structure can be found to solve the given problem using Dynamic-Programming in polynomial time.
Proof:
Proof by counter-example. Here, let's assume that somehow the previous algorithm for the 2D case works for the 3D case with a modification in the the selection of point accordingly. Now, consider the following 3D case points: (1, 2, 3.0), (2, 2, 4.0), (3, 4, 5.0), (1,4,10.0), (3,5,1.0), (3,3,2.0), and the delta value as 10.0. Now, even with the modifications to the 2D algorithms, we can sort the points first on x values and then on y values which is essentially $O(n \cdot log(n))$. After that, we calculate the Errors similar to 2D case but in a 3D manner which turns the runtime algorithm to $O(n^3)$. Further, to find the optimal cost, the problem is that it is in 3D space. Thus even the modified version of the previous problem cannot be used because the equation (1) is true for 2D case but for the 3D case it doesn't work in a simple linear way as we have measure the change in points in 2 dimension which cannot be executed in polynomial time with an optimal sub-structure.

Guide to Code

Both the codes were written in Java language.

Steps

1. Download and Install JDK 11+ on your system using the following link: JDK
2. Create new java projects one-by-one and use the $WACS.java$ and $ICA.java$ for prob. 1 and prob. 2.
3. Run the java projects and interactively give input according to the prompt.

* *Tip: Use VS code and Java extensions for easy install and execution.*

References

[1] Eun-young Cho. Jprofiler: Code coverage analysis tool for omp project. *Technical Report: CMU 17–654 & 17, Tech. Rep.*, 2006.

[2] Oxford Dictionaries. What is the frequency of the letters of the alphabet in english, 2010.