Assignment 4

Vivek Bansal (Id : 112044493)
CSE 548 : Analysis of Algorithms

Due Date : November 07, 2018

1 Longest Path in an Ordered Graph

a) The given algorithm is not correct. Let's consider an example when we have an ordered graph G with edges (v1,v2), (v2,v5), (v1,v3), (v3,v4) and (v4, v5). The algorithm will pick edge (v1,v2) as 2 is smaller than 3. Next it will pick (v2,v5). So the solution given by the algorithm is 2 whereas the length of optimum solution is 3, i.e. (v1,v3), (v3,v4), (v4,v5).

b) We will use dynamic programming to solve this problem. We will take an array dp where dp[i] denotes the maximum length of the path from v1 to v_i. We will initialise dp[i] with -∞ for all i since there is a possibility of no path from v1 to v_i. We will set dp[1] = 0 since the path length from v1 to v1 is 0.

Algorithm :
1) Take an array dp[1...n].
2) Initialise dp[1] = 0 and dp[i] = -∞ for i = 2,...,n.
3) For i = 2,...,n
    For each edge from j to i
        if dp[j] != -∞ and dp[i] $<$ dp[j] + 1
            dp[i] = dp[j] + 1
4) Return dp[n] as the length of the longest path from v_1 to v_n.

Correctness:
For each entry i in dp[ ] array, we are checking for all possible cases by considering each edge incident on i and finding the optimal solution to i, so algorithm guarantees to find the optimum solution from v_1 to v_n.

Running time:
The running time of the algorithm is O(n^2) since to compute each entry i in array dp[ ], we are looking all edges incident on i. So for each entry we can check upto n entries in worst case, so running time is O(n^2).

2 Word Segmentation Problem

Subproblems: Let total[i] be the maximum quality of any segmentation of the letters $y_1y_2...y_i$. To compute the maximum quality at (i+1) letter, we use the subproblems to find the best segmentation till this letter. We will break the word at very possible point and check the maximum value which we are getting to compute the optimal entry at index (i+1).

total[i] = $max_{1 \leq j \leq i}$(total[j-1] + Quality($y_jy_{j+1}...y_i$))

There will be 2 functions, one which will compute the maximum quality till n and other which will print the best segmentation of maximum quality.

Algorithm:

```
GetBestQuality(y1y2...yn):
    total[0] = 0
    for i = 1 to n
        total[i] = -∞
        for j = 1 to i
            if total[i] < total[j-1] + Quality(yjyj+1...yi)
                total[i] = total[j-1] + Quality(yjyj+1...yi)
                parent[i] = j
    return total[n]
end function

PrintSegmentation(parent):
    segment = [ ]
    i = n
    while i > 0
        j = parent[i]
        add yjyj+1...yi to starting of segment
        i = j - 1
    return segment
end function
```

Correctness:
Let's prove the correctness of the algorithm using induction. The base case is trivial when we have only one word with one letter. Using induction hypothesis, we have optimal solution till i-1 index and we have to show that total[i] is our optimal solution at index i. Let's assume the last word in the optimal segment starts at j<=i. But from induction hypothesis we know that prefix containing only j-1 characters must be optimal. Therefore the solution at index i which is the sum of total[j] and quality of last word must also be optimal.

Running time:
The running time of the algorithm is O(n^2) as to compute entry at every index i, we are checking all values from 1 to i.

3 Opportunity Cycle Problem

Solution :
We have to find out whether :

$$r[1,2].r[2,3].r[3,4].....r[k,1] > 1$$

Taking reciprocal of above equation and changing the inequality, we get:

$$\frac{1}{r[1,2]}.\frac{1}{r[2,3]}.\frac{1}{r[3,4]}.....\frac{1}{r[k,1]} < 1$$

Taking log on both sides, we get:

$$\log(\frac{1}{r[1,2]}) + \log(\frac{1}{r[2,3]}) + \log(\frac{1}{r[3,4]}) + ... \log(\frac{1}{r[k,1]}) < \log(1)$$

$$=> -\log(r[1,2]) - \log(r[2,3]) - \log(r[3,4]) - ..... \log(r[k,1]) < 0$$

Therefore, if we define the weight of an edge (v_i,v_j) as :

$$w(v_i, v_j) = -\log(r_{ij})$$

So, now we have find out whether there exists a negative weight cycle in the graph G with edge weights as defined by above relation.

We will use Bellman-Ford algorithm to detect negative cycle in the above graph. We will add an extra vertex which will point to every other vertex in the graph and will assign edge-weights of 0 to all these edges. We will run Bellman-Ford from this newly added vertex and check the result after running that algorithm. Bellman-Ford returns TRUE if there is no negative-weight cycle and FALSE if there is a negative-weight cycle. We will invert the result returned by Bellman-Ford for the solution of our problem.

This method will work because adding a new vertex with all edge-weights as 0 can't introduce any new negative cycle. It is just added so that every vertices are reachable from this vertex.

Running time:
The running time of the algorithm is O(n^3) since Bellman-Ford takes O(n^3) to detect the negative cycles in the graph.

4 Interleaving Problem

Approach :
Let's say string s contains n characters in total. Let's say x' is the repetition of x and y' is the repetition of y containing exactly n characters. We have done this so that we don't need to start again from either of the strings x and y. So now our problem becomes to check whether s is an interleaving of x' and y'.

Now, if s is an interleaving of x' and y', then its last character should either be x'[n-1] or y'[n-1]. So our problem now reduces to n-1 sub-problems. Thus we can apply dynamic programming to solve this problem. We will be creating a 2D table dp[n+1][n+1] to store solutions to subproblems in bottom up manner. dp[i][j] is a boolean entry denoting whether string s[0...i+j-1] is an interleaving of x'[0...i-1] and y'[0...j-1].

Algorithm :

1) Take a matrix dp[n+1][n+1]. Initialize all entries with False.
2) Initialize some entries of this matrix:
   dp[0][0] = true
   Initialize first row and first col of this matrix:
   dp[0][j] = dp[0][j-1] if s[j-1] == y'[j-1] (for j = 1 to n)
   dp[i][0] = dp[i-1][0] if s[i-1] == x'[i-1] (for i = 1 to n)
3) If current character of s matches with both current character of x' and current character of y'
   if (s[i+j-1] == x'[i-1] and s[i+j-1] == y'[j-1])
      dp[i][j] = dp[i-1][j] || dp[i][j-1]
   If current character of s matches with current character of x' and not with current character of y'
   if (s[i+j-1] == x'[i-1] and s[i+j-1] != y'[j-1])
      dp[i][j] = dp[i-1][j]
   If current character of s matches with current character of y' and not with current character of x'
   if (s[i+j-1] != x'[i-1] and s[i+j-1] == y'[j-1])
      dp[i][j] = dp[i][j-1]
4) return dp[n][n]

Running time:
The running time of the algorithm is O(n^2) since there are n^2 values to compute in our table dp[ ][ ] and taking constant time to get the result by looking at subproblems.

5 Gerrymandering problem

Solution:
The brute force solution to this problem is O(2^n) in which we will assign each precint one time to district 1 and one time to district 2 and check the results for both the cases. This approach is really inefficient.

A better approach can be if we know the solution till precint j, then solution to precint (j+1) can be easily computed by including this precint either in district 1 or in district 2 and using the solutions of the subproblems. So we will use dynamic programming to solve this problem.

Let's say we have dp[c, s, a1, a2] which is storing the results to our subproblems. dp[c, s, a1, a2] = true when we are at current precint c, s precints are already selected to be in district 1, atleast a1 votes of party A are in district 1, and a2 votes of party A are in district 2, otherwise it will be false. Let's say we have to compute dp[c+1, s, a1, a2], we will put c+1 precint either in district 1 or in district 2. Let's say precint (c + 1) has k votes of party A. If we are putting (c+1) precint in district 1, then we have to check solution of subproblem dp[c, s-1, a1 - k, a2], and if we

are putting (c+1) precint in district 2, then we have to check solution of subproblem dp[c, s, a1, a2-k].

To get the solution to the whole problem, scan the complete table to check for the true value in every positions of the type dp[n, n/2, x, y] where x $>$ mn/4 and y $>$ mn/4 since each district has mn/2 votes in total. Our algorithm runs in the order of increasing j values since to solve the problem we need the solutions to its subproblems.

Running time:
The running time of the algorithm is O(n^2m^2) since there are n^2m^2 subproblems in the table.