Assignment 4: CSE548: Analysis of Algorithms

Shivasagar Boraiah — #112077826

November 8, 2018

Problem 1. Textbook [Kleinberg & Tardos] Chapter 6, page 312, problem #3.

Proof. ...
Input: Ordered Graph, G with nodes $v_1 v_2 .... v_{n-1} v_n$
Approach for (a): One such example where the algorithm fails is that consider graph G with edges, (v_1, v_2), (v_1, v_3), (v_2, v_5), (v_3, v_4), (v_4, v_5), (v_5, v_6). The algorithm will output 3, where as the desired result is 4. Algorithm starts with node, v_1 and considers v_2 as it is shortest to v_1 ignoring v_3 which reaches v_6 with as depth as possible. While finding the shortest path, we should consider all possible sub problems and proceed with the maximum path length to the next step. The best solution is dynamic approach.
Approach for (b): The problem can broke down and solved incrementally, bottom-up approach as we did in class for the problem, Longest Increasing Subsequence. let L[i] be the longest path from v_1 to v_i and let $L[i] = -\infty$, if there is no path from v_1 to v_i. while constructing the actual longest path, we store the prdecessor of i on a longest path from v_1 to v_i while computing, $L[i]$.
LongestPath(G):

for i from 2 to n;
  $L[i] = -\infty$;
   for (v_i, v_j) belongs to EdgeSet, E;
    if 1+ L[j] > L[i];
     if L[i] = 1+ L[j];

Output: To find the maximum distance from v_1 to v_i, output, $L[i]$.
Correctness: As we take bottom up approach by calculating and solving smaller sub problems for every vertex, this algorithm correcteness can be achieved by induction hypothesis.
Time Complexity: As there are two nested loops in the worst case the program runs in $O(n^2)$ running time. □

Problem 2. Textbook [Kleinberg & Tardos] Chapter 6, page 312, problem #5.

Proof. ...
Input: Long string of English letters, Y = $y_1 y_2 ..... y_{n-1} y_n$ and a Blackbox that compute $Quality(x)$, where x is a possible English word.
Approach: Idea is to break the big string to smaller sub problems by segmenting them into prefix string and and a character match. i.e. If segmentation of Y into $y_1 y_2 ..... y_{n-1} y_n$

yields the maximum total quality, then $y_1 y_2 .....y_{n-1}$ gives the maximum quality for string Y without y_n. Maintaining a score matrix for every sub problem staring from index 1 to n and incrementally adding the calculating scores should give us the maximum quantity at score[n][n].

Score(Array[0....n-1]):

```
Score = [[0 for i in range(length(Array)+1)]0 for i in range(length(Array)+1)];
for i in range(length(Array)+1);
  for j in range(i, length(Array)+1);
    Score[i][j] = min(Score[i-1][j-1] + quality(Array[j][n-1]);
return Score[n][n]
```

Output: As the problem counts the quality of each character with the rest of the string, the highest quality is computed at score[n][n]

Correctness: As we are calculating incrementally whether the "*ith*" character place any quality with the rest of the string keeping the computation of i-1 value on string, This incremental computation finds the cost correctly at the end of the ith index.

Time Complexity: from the above algorithm implemenatation, we can conclude it is $O(n^2)$ running time. □

Problem 3. Textbook [Kleinberg & Tardos] Chapter 6, page 312, problem #13.

Proof. ...

Input: Graph G with vertices $i_1 ....i_k$ and to find a oppurtunity cycle in it.

Approach: One way to0 find the cycle is by running Djikstra's algorithm or Bellman Ford on the given edge list. Recursively by solving for cycles with small subset of vertices, we can find the existence of cycle for a given graph G using Dynamic programming too.

Approach for (b): The problem can broke down and solved incrementally, bottom-up approach as we did in class for the problem, Longest Increasing Subsequence. let L[i] be the longest path from v_1 to v_i and let $L[i] = -\infty$, if there is no path from v_1 to v_i. while constructing the actual longest path, we store the prdecessor of i on a longest path from v_1 to v_i while computing, $L[i]$.

Algorithm

for i from 1 to k;
  $Distance[i] = -\infty$;
  $Visited[i] = FALSE$;
  $Previous[i] = NULL$;
$Current = v$: for v in V;
for v in V:;
  for every vw in E;
    Update $Distance[v]$;
    Mark $Previous[v] = \{w\}$;
    Mark $v = Unvisited()$;

Output: If the combined weight of that cycle gives you "opportunity" then it is an opportunity cycle. If it does not, then there is none. Applying them to all vertices one by one we can check for the cycle.

Correctness: Existence of cycle can be found for every vertex using the distance list, which

proves that the results at all levels are identified and captured properly.
Time Complexity: As we are running the algorithm for n nodes, this is $O(n)$ complexity. □

Problem 4. Textbook [Kleinberg & Tardos] Chapter 6, page 312, problem #19.

Proof. ...
Input: Three strings, x, y, s
Approach: The problem can be break down into simpler and smaller sub problems. Given that, x, y, s strings, if we are to find whether s is formed by interleaving x and y, we start comparing $s[n]$, the last character of s is equal to x or y and if so by extending it to comparing till $s[1]$ with x and y, we say that the original computation is true. 2D matrix, $Output[length(x)+1][length(y)+1]$ is used to save the smaller computations, $output[i][j]$ means that, $s[i+j]$ can be formed using $x[i]$ and $y[j]$ characters.
Output(x, y):

output = [[0 for i in range(length(x)+1)]0 for j in range(length(y)+1)];
for i in range(length(x)+1);
  for j in range(i, length(y)+1);
    if output[i-1][j] == $True$ and $s[i+j-1] == x[i-1]$;
      output[i][j] = True;
    else if output[i][j-1] == $True$ and $s[i+j-1] == y[j-1]$;
      output[i][j] = True;
    else;
      output[i][j] = False;
return output[n][n]

Output: If output[n][n] is $True$ then the string s can be formed by interleaving two inputs, else not.
Correctness: At any point of time, $output[i][j]$ holds value computed from , $s[i+j]$ and $x[i]$ and $y[j]$ characters. So, as this algorithm is incrementally working by dividing into sub problems and combining them, back. By induction on s we can verify the correctness
Time Complexity: from the above algorithm implemenatation, we can conclude that it runs in $O(n^2)$ running time. □

Problem 5. Textbook [Kleinberg & Tardos] Chapter 6, page 312, problem #24.

Proof. ...
Input:set of n precincts $P_1, P_2, ..., P_n$ each containing m registered voters. **Output:** Give an algorithm to determine whether a given set of precincts is susceptible to gerrymandering; the running time of your algorithm should be polynomial in n and m.
I have discussed this problem with Abhishek Reddy Y N Approach: Consider precinct n, and assume precinct n has an party-A voters. We can choose either to include precinct n in our district or not. If we do include precinct n, our system is susceptible to gerrymandering if we can come up with a set of n/2 1 precincts from among the remaining precincts 1,. . . , n 1 that has s a_k total party-A voters. if we exclude precient n, then system can be susceptible to gerrymandering if we can come up with a set of n/2 precincts

from among the remaining precincts 1, . . . , n1 that has s total party-A voters.

Gerrymander(n)

G[n][n/2][a-nm/4-1]:;
for i from 1 to n;
$G[k][0][0] = True$;
for k from 1 to n;
for l from 1 to n/2;
for w from 1 toa-nm/4 -1;
if k == 1, l == 1, w == a1;
$G[k][l][w] = True$;
else;
if G[k-1][l-1][w-ak];
$G[k][l][w] = True$;
else;
if G[k-1][l][w];
$G[k][l][w] = True$;
else;
$G[k][l][w] = False$;

Output: Return True if the given precients are succestiple else False.

Time Complexity: As the algorithms ravels through three loops, filling the column takes $O(n^3)$ time. The final travelsal which is dominated by first traversal by taking overall time of $O(mn^3)$. □