Untitled

April 26, 2019

ANLY 550 Assignment 5 Norman Hong

1. Prove that there is a unique minimum spanning tree on any connected undirected graph when the edge weights are unique.
   Suppose G has unique edge weights. Suppose X is the MST on G. Lets replace an edge in X with another edge in graph G that still allows for the creation of MST. This implies that edges that got swapped have the same edge weight because if we have a MST, then that means that the sum of edge weights is the minimum and the minimum is unique to a graph. Therefore, this implies edge weights are not unique. This leads to a contradiction, therefore, there must be a unique minimum spanning tree.

2. Suppose we have an array A containing n numbers, some of which may be negative. We wish to find indices i and j so that the sum of elements from i to j is maximized. Find an algorithm that runs in time O(n).
   Let subproblem i denote the max sum of contiguous integers from A[j, j+1, . . . , i] where j represents the start of the subarray. The solution to the subproblem is either the sum of all entries from j to i-1, which is the solution to subproblem i-1, or the ith element. Let B[i] be an array that contains the ith subproblem solution. Then, B[i] = max(B[i-1] + A[i], A[i]). The desired solution is the max value of all subproblems, denoted by max(B). The algorithm will iterate from i = 1 to n, and fill in B[i] according to the maximization equation. Then it will simply return the max value in array B.

Pseudocode:

```
Max_subarray_finder(array A of length n){
B[0] <- 0
B[1] <- A[1]
    for i in 2, . . . , n{
B[i] <- max(B[i-1] + A[i], A[i])
}
return max(B)
}
```

The for loop runs n-2+1= n-1 times doing constant amount of work. Return max(B) will not take longer than n steps, if do brute force approach. Therefore, the overall runtime is O(n).

Proof of correctness:

At each iteration, the correct subproblem solution is found. For subproblem 1, the solution is simply the first element in A. For subproblem 2, the algorithm compares A[1, 2] to A[2]. It is clear, that the solution to this subproblem is the max value of the comparison. Assume that the algorithm correctly solves each problem for input size less than n. Suppose Max_subarray_finder

is called on an array of size n. The inductive hypothesis implies that all the subproblems from 1 to n-1 are computed correctly. On the nth subproblem, the algorithm will do B[n] <- max(B[n-1] + A[n], A[n]). The inductive hypthesis implies that B[n-1] is the correct solution. It is clear that the solution to the nth subproblem is that either max subarray contains the nth element or it doesn't. Therefore, it is clear that B[n] will contain the correct solution. The algorithm then returns the max value out of array B, which is the desired output.

3. Reference problem set 5 for prompt.
   The neatest page is determined by minimizing the sum of the penalty for all lines except last line where penalty $= (m - j + i - \sum_{k=i}^{j} l_k)^3$. $i - j$ is the number of spaces in the line. Let a = 1, 2, ... denote the ath subproblem where each subproblem's goal is to create the neatest line. The neatest page is determined by minimizing the penalty for the page and making sure each line does not exceed M. Let A[1, 2, 3, ...] be the array that contains the solution to the ith subproblem. In other words, Array A[1] contains the words to be printed out for line 1. Let array W contain the words to be printed out. Let array P[1, 2, 3, ..., n] be the array that contains penalty for ith sub problem. The algorithm simply runs until W becomes empty. At each iteration, the algorithm starts at the beginning of W and computes the penalty for j = 1, 2, ... until the penalty becomes negative. The penalty is stored in array temp for each iteration. For the ith subproblem, the minimum value from temp is stored in P and A[i] = W[1, 2, 3, ..., length(temp)] stores the words for line i. The length of temp represents the final word to be added to the line. The words that are added to A are then deleted from W and this is repeated until W is empty.

Pseudocode:
neatest_page(M, array W of words){
while W does not equal null{
a=1
i=1
temp=[] //track penalty. for j in 1, 2, ..., n{
Pen <- $(m - j + i - \sum_{k=i}^{j} l_k)^3$
if Pen>0{ // making sure the line is not longer than M temp[j] <- Pen
}
P[a] <- min(temp)
A[a] <- W[1, 2, 3, ..., length(temp)]
a = a+1
}
Delete the 1st 1, 2, 3, ..length(temp) elements from W.
} return A, P }

The while loop occurs worst case n times and the for loop occurs worst case n times. Therefore, the runtime is O(n^2).

Proof of correctness:
At each subproblem, the algorithm correctly minimizes the penalty by fitting in as much words as possible to a line without exceeding M by using an iterative process through the search space. This implies that at each iteration of the while loop, the correct solution is found. Because each subproblem has the correct minimum penalty, the overall penalty for the page is also minimized. This implies that the desired output is achieved.

4. Another type of problem often suitable for dynamic programming is problems on tree graphs. For example, suppose we have a graph G = (V, E) that is a tree with a root r. Derive a

recursion to find the size of the maximum-sized independent set of G. (An independent set is a subset of graph vertices, such that no two have an edge between them). For full credit, show that you can find the size of the maximum-sized independent set in linear time.

Let X be the independent set of vertices. Define the ith subproblem to be subtrees of node rooted at ith node. Let solution to the subproblem be smax size of independent set rooted at i node. The solution to i subproblem is the max of the i-2 solution plus 1 or just the previous solution. In other words, the solution is should we add root to X or not. Assume trees have pointers to child nodes. Let A[1, 2, . . . , n] be array that contains the nodes of distance from high to low of tree rooted at R. A is the reverse order of the distance of each node from r, which is calculated by BFS. This is the order of the subproblems to solve. Let I[i, i+1, . . . , n] be the max size of independent set of subtree rooted at node i, where i, . . . , n are nodes. Therefore, I[i] = max(1+$\sum_{x=grandchild} I(x)$, $\sum_{x=child} I(x)$). The algorithm simply iterates from n to 1, and fills in array I. Then the desired solution is simply to return I[1].

Pseudocode:

maximum_sized_independent_set(Graph G){ call BFS on G rooted at r and return an array A of nodes in decreasing distance.
Let I[1, 2, . . . , n] = 0 be array that stores the solution to the subproblem rooted at i.

for e in A { gc = granchildren(e) // array of grandchildren of node e
c = children(e) //array of children of node e.

sum_gc = 0 for x in gc{ sum_gc = I[x] + sum_gc }

sum_c = 0 for x in c{ sum_c = I[x] + sum_c } I[e] = max(1+sum_gc, sum_c) } return I[1] }

Since there is a nested for loop, it is reasonable to think that the runtime is O(n^2) because it is possible in the worst case that there is n grandchildren or n children. However, note that the algo does a constant amount of work when it checks out a node. Notice that each vertex j is processed 3 times, 1 time when algo encounters j, when j is a child, and when j is a grandchild. This means each vertex is looked at 3 times and each time a constant amount of work is done. This implies the overall runtime is O(n).

Proof of correctness: Let X denote independent set of nodes. Suppose the max independent set size was returned incorrectly. The algo solve sthe subproblem of finding the max independent set of subtree rooted at ith node by determining if i belongs to the set X or not. If it belongs to X, then the size of set X is incremented by 1 by adding 1 to the sum of size of subtree rooted on i's grandchildren. The only way this fails is if the graph is not a tree. This is the case if there are cycles or if entire graph is not connected. The proble massumes the graph is a tree. Therefore, we have a contradiction when assumed the max independent set size was returned incorrectly. Therefore, it must be the case that the max independent set size was returned correctly.