ANLY550 Homework 5

Hongyang Zheng

Problem 1

Prove that there is a unique minimum spanning tree on a connected undirected graph when the edge weights are unique.

Proof: Assume there are two minimum spanning tree T_1 and T_2 for this connected undirected graph with unique edge weights. Since every edge has different weight, the edge e_1 with the minimum weight must in T_1 or T_2. Assume it is in T_1, then $T_2 \cup e_1$ must generate a cycle. In this cycle, we know there is an $e_2 \in T_2$ and $e_2 \notin T_1$, since edge weights are unique, and we know $weight(e_1) < weight(e_2)$. Thus $T_2 \cup e_1 \setminus \{e_2\}$ will generate a spinning tree T_3, and T_3 has a smaller weight than T_2, which means T_2 is not a minimum spanning tree. Since there is a contradiction, our original assumption is not true. As a result, there is a unique minimum spanning tree on a connected undirected graph when the edge weights are unique.

Problem 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 $\sum_{k=i}^{j} A[k]$ is maximized. Find an algorithm that runs in time $O(n)$.

Describe the Algorithm

Step 1: Initialize $i = 0$ and $j = 0$. Let current maximum cur_max and maximum max be $A[0]$.

Step 2: For k from 1 to n which n is the length of array A. Compare $A[k]$ with $A[k] + cur\_max$.

- If $A[k] > cur\_max + A[k]$, which means cur_max may be negative. Thus excluding it can help us find the maximum sum, so we let the start index $i = k$, and the current maximum is $A[k]$.
- If $A[k] <= cur\_max + A[k]$, the current maximum is $cur\_max + A[k]$.

Then we compare max with cur_max to find the end index. If $max < cur\_max$ we let the end index $j = k$ and $max = cur\_max$.

Pseudocode

```
Let i=0, j=0, cur_max=A[0] and max=A[0]

For k in range(1, n):

    If A[k]>cur_max+A[k]:

        i = k, cur_max = A[k].

    Otherwise:

        cur_max = cur_max + A[k].

    If max < cur_max, then j=k, max = cur_max.

Return i, j, max.
```

Correctness

If $A[k] > cur\_max + A[k]$, we can start from the index k to find the maximum subarray sum (cur_max may be negative). As the cur_max stores the current maximum sum, it may be smaller than the final maximum sum. We update the final maximum sum only when $max < cur\_max$, and update the end index j to index k.

Analysis of Time

Since A is an array and we check each element of A only once, the running time is $O(n)$.

Problem 3

Suppose we want to print a paragraph neatly on a page. The paragraph consists of words of length $l_1, l_2, \ldots, l_n$. The maximum line length is M . (Assume $l_i \leq M$ always.) We define a measure of neatness as follows. The extra space on a line (using one space between words) containing words l_i through l_j is $M - j + i - \sum_{k=i}^{j} l_k$. The penalty is the sum over all lines except the last of the cube of the extra space at the end of the line. This has been proven to be an effective heuristic for neatness in practice. Find a dynamic programming algorithm to determine the neatest way to print a paragraph. Of course you should provide a recursive definition of the value of the optimal solution that motivates your algorithm.

The code is shown in neatness.py

Describe the Algorithm

Let $p[n]$ be the total penalty for the neatest way to print words l_1 through l_n.

Let the subproblem be $p[j]$ which is the optimized total penalty for printing words from l_1 to l_j.

Let $extra[i,j]$ indicate the extra space between words l_i through l_j.

Let $part\_p[i,j]$ indicate the penalty from l_i to l_j, which is the cube of $extra[i,j]$ except the last of the cube of the extra space at the end of the line.

The recursion is: $p[j] = \min\{p[i] + part\_p[i+1, j], p[j]\}(1 \leq i < j)$.

Pseudocode

```
for j from 1 to n:

    for i from 1 to j:

        p[j] = min { p[i]+ part_p[i+1, j], p[j] }.
```

Correctness

The sum penalty $p[j]$ for words l_1 through l_j can either be

- $p[j]$ itself, if adding the word l_j still keep all words on one line

or

- The sum penalty for all lines except the last line, which is the words l_1 through some l_i $(1 \leq i < j)$, plus the penalty for words on the last line l_{i+1} through l_j.

Analysis of Time

We have one outer loop and one inner loop, so the time is $O(n^2)$.

Problem 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 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 and the set itself in linear time.

Describe the Algorithm

In a tree, the maximum-sized independent set is either the root + the maximum-sized independent set of its gradchildren or the maximum-sized independent set of its children. Let $I(j)$ be the size of the maximum-sized independent set in the subtree rooted at vertex j. Let $G(j)$ be the set of all the grandchildren of vertex j, and let $C(j)$ be the set of all the children of vertex j. Then we can express the recursive relation as $I(j) = \max\{1 + \sum_{u \in G(j)} I(u), \sum_{v \in C(j)} I(v)\}$.

Pseudocode

```
Let the size of the maximum independent set I=0, and let the set of the maxi
mum independent set s=[].

For each vertex j of G in the postorder of the DFS of G:

    if j is a leaf:

        then s.append[j] and I = I+1

    else if 1 + sumI(u) > sumI(v), where u is the grandchildren of j and v i
s the children of j:

        then s.append[j] and I = I+1

Return I, s
```

Correctness

Assume given a vertex j, we know the size of the maximum-sized independent set of its subtrees. As describe in the above, there are only two cases of the maximum-sized independent set: either j is in the set, or it is not. If it is not, to be independent, the maximum independent set is the union of the maximum-sized independent sets of the `children` of j. If it is, to be independent, the maximum independent set is the union of j and the maximum-sized independent sets of the `grandchildren` of j.

When we check each vertex in postorder, we can find a leaf before the leaf's parent. The number of leaves of a vertex is always no smaller than 1, so always adding leaves to the independent set can make sure the independent set is maximized and does not generate conflict.

Analysis of Time

Suppose there are total n vertex, since we check each vertex only once in the postorder, the run time is $O(n)$.