Lecture 5: Dynamic Programming II

Scribe: Weiyao Wang

September 12, 2017

1 Lecture Overview

Today's lecture continued to discuss dynamic programming techniques, and contained three parts. First, we will continue our discussions on knapsack problem, focusing on how to find the optimal solutions and the correctness proof for the algorithm. Then we will discuss two more dynamic programming problems: largest common subsequence problem and maximum independent set on trees problem.

2 Knapsack Problem (continued)

2.1 Algorithm Recap

Definition: We define states (sub-problems) as the following: the maximum value for a knapsack with capacity j and we are given the first i items.

Then, we have $\forall i, j$, the corresponding state, $a[i, j]$ is $\max(a[i-1, j-w_i]+v_i, a[i-1, j])$, where the first option is the value if item i is in the knapsack and the second option is the value if item i is not in the knapsack.

2.2 Algorithm

We first give an algorithm calculating the state a, and then we give an additional algorithm on how we can transform a to the optimal solution:

```
calculatestate (w,v) ;
Initialize a[i,0] = a[0,j] = 0 as base case ;
for i = 1 → n do
    for j = 1 → W do
        a[i,j] = a[i-1,j] ;
        if j ≥ w[i] and a[i-1,j-w[i]] + v[i] > a[i,j] then
            a[i,j] = a[i-1,j-w[i]] + v[i] ;
        end
    end
end

findoutput(n, W, a);
if n = 0 or W = 0 then
    return
end
if a[n,W] = a[n-1,W] then
    findoutput(n-1, W, a) ;
end
else
    findoutput(n-1, W-w[n], a) ;
    print(n) ;
end
```

2.3 Correctness Proof for Knapsack

We will do proof by induction here:

We say pair $(i,j) < (i',j')$ if $i < i'$ or $(i = i'$ and $j < j')$. For example

subproblem def item x weight

$$(0,0) < (0,1) < (0,2) < ... < (1,0) < (1,1) < ...$$

Induction Hypothesis: algorithm is correct for all values of $a[i,j]$ where $(i,j) < (i',j')$. Or in other words, all previous elements in table are correct.

Base Case: $a[i,0] = a[0,j] = 0$ for all i,j out of room or out of items

Induction Step: When computing $a[i',j']$, by induction hypothesis, we have $a[i'-1,j']$, $a[i'-1,j'-w_{i'}]$ are already computed **correctly**. Then algorithm considers the optimal value for item i' in knapsack as $a[i'-1,j'-w_{i'}]+v_{i'}$ and for item i' not in knapsack as $a[i'-1,j']$. Therefore,

the value at $a[i', j']$ is correct.

3 Largest Common Subsequence (LCS)

3.1 Problem Description

We are given two sequences, a, b, and we want to find an algorithm that outputs the length of the longest common subsequence.

Definition: A subsequence of a sequence is defined as a subset of elements of the sequence that has the same order (not necessarily continue).

For example, if a[]='ababcde' and b[]='abbecd'. We have abac is a subsequence of a[], but not b[] abed is a subsequence of b[] but not a[]. In this example, the LCS='abbcd', and thus our algorithm should output its length, 5.

3.2 Building States (Sub-problems)

Like previous questions in DP, we break down the problem into sub-problems. For this question, let's consider the last decisions to made at each point:

Last Decision: whether $a[n]$ should be in the LCS and whether $b[m]$ should be in the LCS.

Here, we have three possible cases for the decision. Let $c[i, j]$ be the length of LCS of $a[1...i]$, $b[1...j]$, then we have the following decision table:

	$b[m]$ in LCS	$b[m]$ not in LCS
$a[n]$ in LCS	$c[n-1, m-1]+1$	$c[n, m-1]$
$a[n]$ not in LCS	$c[n-1, m]$	

We hope to have the longest subsequence, and thus we want to maximize c. Therefore, $c[n, m]$ can be written as the maximum of the cases below:

1. $c[n-1, m]$ case 1: $a[n]$ not in LCS
2. $c[n, m-1]$ case 2: $b[m]$ not in LCS
3. $c[n-1, m-1]+1$ case 3: $a[n] = b[m]$, both in LCS

Base Case: The base case should be if we are considering no element for either a or b, and LCS should have length zero. Thus, we have if $i = 0$ and $j = 0$, $c[i, j] = 0$

Ordering: $i = 1$ to n, $j = 1$ to n

3.3 Algorithm

case 1: a[i] != b[j]
the LCS is either
c[i-1, j] or c[i, j-1]
matching the current letter
w/ a letter further along

case 2: a[i] == b[j]
we have a match &
can compute the LCS
for the rest of each
substring
c[i-1, j-1] + 1

calculatestate (w,v) ;
Initialize $c[i, 0] = c[0, j] = 0$ as base case ;
for $i = 1 \rightarrow n$ **do**
  for $j = 1 \rightarrow m$ **do**
    $c[i, j] = max(c[i-1, j], c[i, j-1])$ (case 1 and 2) ;
    if $a[i] = b[j]$ *and* $c[i-1, j-1] + 1 > c[i, j]$ **then**
      $c[i, j] = c[i-1, j-1] + 1$;
    end
  end
end

The proof of correctness should be similar to the knapsack problem through induction.

4 Maximum Independent Set on Trees

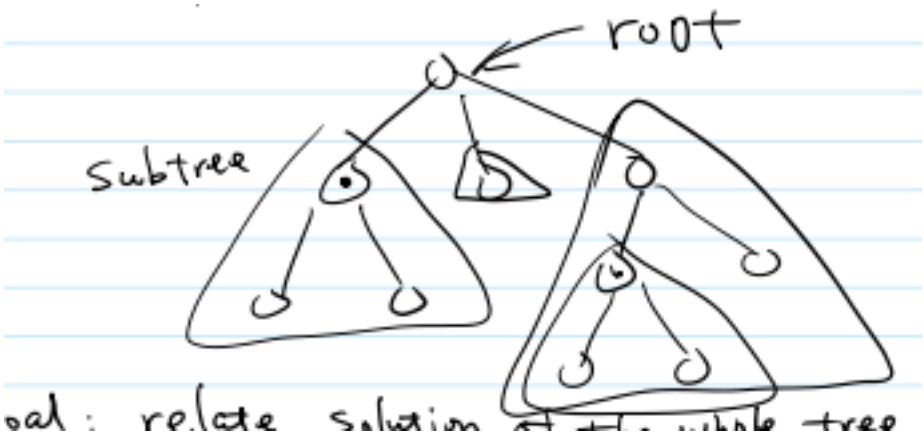
4.1 Problem Description

We are given a tree (not necessarily binary), and we are hoping to find an independent set such that the size (number of nodes) of the set is maximum.

Independent Set: Set of nodes that are not connected by any edges.

4.2 Building States (Sub-problems)

We consider each subtree as a sub-problem, and thus our goal is to relate solution of the whole tree to solutions of the subtrees.



For the root, there are two cases:

1. if the root is not in the independent set, take maximum independent tree for all children's subtrees

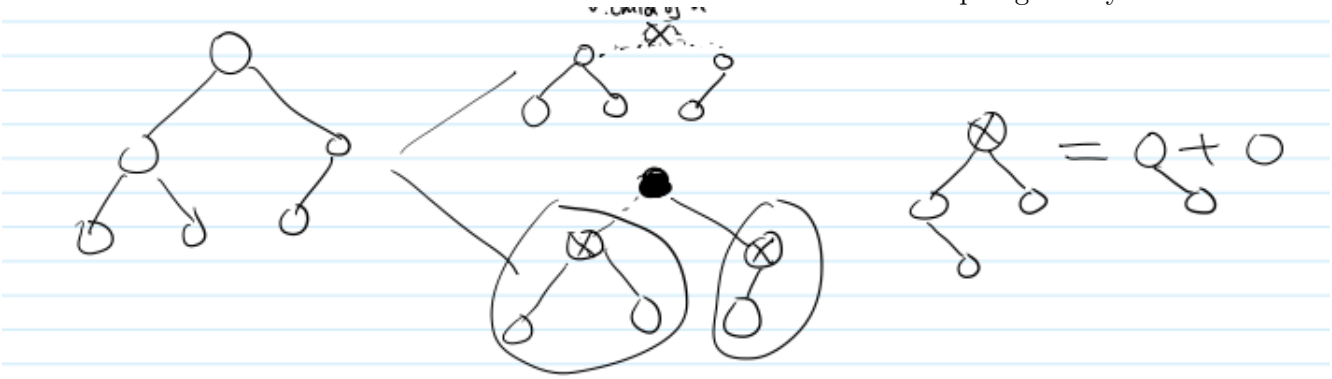
2. if the root is in the independent set, maximum independent set on a subtree if root of the subtree cannot be chosen.

Therefore, we define two functions respectively

1. $F(u)$ = maximum independent set of subtree rooted at u

2. $G(u)$ = maximum independent set of subtree but u cannot be in the set

In the first case, if the root u is not in the set, we may consider the sub-problems for all subtrees where the root of the subtrees can be in the set since the current root is not. If the root is in set, then the roots of its subtrees cannot be in the set since they share edges with the root. And thus we have $F(u) = max(\sum_{v:child\ of\ u} F(v), \sum_{v:child\ of\ u} G(v) + 1)$, whereas the first case is for u not in the set and the second case is for u in the set. And we also have $G(u) = \sum_{v:child\ of\ u} F(v)$, the same as case 1 for F since u is not in the set. An illustration of the relationship is given by



4.3 Algorithm

Base Case: If u is a leaf (or only one node), then we have $F(u) = 1$ and $G(u) = 0$.

Ordering: We want to compute it in a fashion that we start from the deepest of the tree and continue to move to the shallower part. With the base case, we want to compute one layer above the leaf, and then one layer above. So we hope to have something where the children of a node are calculated before the node itself. To do this, we can run BFS, and we will have a traversal of the tree by distance from the node. If we reverse the list, we will have a list reversely ordered by the

distance, and thus the children come before a node.

findmaxset (Tree) ;

Run BFS and reverse order the traversal by the distance from the root so that the children of a node always come before the node in the list; suppose the list is a;

for $i = 1 \rightarrow len(a)$ **do**

  if $a[i]$ *is a leaf* **then**

    $F[i] = 1$;

    $G[i] = 0$;

  end

  else

    $F[i] = max(\sum_{a[j]:child\ of\ a[i]} F(j), \sum_{a[j]:child\ of\ u} G(j) + 1)$;

    $G[i] = \sum_{a[j]:child\ of\ a[i]} F(a[j])$;

  end

end

return $F[len(a)]$