CSE341T/CSE549T **09/29/2014**

Dynamic Programming II

Lecture 10

In this lecture, we will see more problems that can be solved using dynamic programming.

The Rod-Cutting Problem

Say, you have a friend who is in the business of selling steel rob — what he does is he buys long steel rods and cuts them into shorter rods, which he then sells. Assuming that he knows the price p_i for selling a rod with length i, and he is wondering how he can maximize his profit. For instance, say that the price for steel rods are as follows:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	7	9	10	17	17	20	24	30

If he has a rod of length 4, the best strategy is to cut it into two 2-inch pieces, which he can then sell for 10 dollars. He has figured this out via manual calculation, but once the length gets longer, he wonders if there is a systematic way of calculating the maximum price. Fortunately, he's got a friend (you), who is taking CSE 341 and knows exactly the technique to use to solve his problem — dynamic programming. So let's think about how we can solve his problem.

We can solve this problem inductively by considering all possible ways of cutting a rod (brute force):

```
CUTROD(p, n)
1  if n ≤ 1
2     then return p[n]
3  q ← −∞
4  for i ← 0 to n
5        do q ← Max(q, p[i] + CUTROD(p, n − i))
6  return q
```

At each level of recursion, we basically fix how long to the first piece to be, followed by some decomposition of the remaining rod.

This program has running time $T(n) = \sum_{i=0}^{n-1} T(i) + O(1)$, which solves to $\theta(2^n)$, exponential in n. This running time is terrible, but it gives a good hint as to how we can solve it using dynamic programming. We can see that the problem has a lot of overlapping in subproblems, and the solution to the problem is obtained by combining the solution to subproblems.

In the last lecture, we learned that one can implement dynamic programming two ways: top down and bottom up. We will use the bottom up method, i.e., building a table, solving the smaller subproblems first and combining them to get solutions to the larger subproblems. One can solve this problem using DP with a 1-dimensional table, where each table cell $r[i]$ stores the best price one can get for selling rod of length i (either by itself or by cutting it up somehow). Then we simply fill out the table for $i \leftarrow 1..n$:

BOTTOMUPCUTROD(p, n)

```
1  let r[0..n] be a new array
2  r[0] ← 0
3  for j ← 1 to n
4       do q ← −∞
5          for i ← 1 to j
6               do q ← Max(q, p[i] + r[j − i])
7          r[j] ← q
8  return r[n]
```

To solve a subproblem of size n, the inner loop iterates for $\theta(n)$ times, so the total work is $\theta(n^2)$.

Can we parallelize this program? Yes, in particular, we can compare all the different cuts in parallel. That is, we can parallelize the inner loop. We have to careful not to race on q, however, so one way to do it is to store the result of computing each $p[i] + r[j - i]$ into a temporary array, and then do a MAX-REDUCE on the resulting array. Doing that will give us a span of $\theta(n \lg n)$.

Reconstructing a solution

The program we wrote gives us the optimal solution, but doesn't tell us how to get there. While it's great to know the maximum profit one can sell a rod, it's no good if your friend still can't figure out how to cut it to get the price, so we need to extend our program to not only compute the optimal solution but also remember the choice that leads to that solution:

```
BOTTOMUPCUTRODEXTENDED(p, n)
 1  let r[0..n] be a new array
 2  r[0] ← 0
 3  for j ← 1 to n
 4       do q ← −∞
 5          for i ← 1 to j
 6               do if q < p[i] + r[j − i]
 7                    then q ← p[i] + r[j − i]
 8                         s[j] ← i
 9          r[j] ← q
10  return r and s
```

This procedure is very similar to BOTTOMUPCUTROD, except that it also creates a table s of size n that remembers the optimal size $s[j]$ for the first piece that we cut off when we solve a subproblem of size j.

```
PRINTCUTS(p, n, s)
1  while n > 0
2       do print s[n]
3          n ← n − s[n]
```

Exercise 1 *Use induction to show that* CUTROD *has work* $\theta(2^n)$.

Subset Sum

The subset sum problem (SS) is defined as follows:

Definition 1 *The* subset sum *(SS) problem is given a multiset of integers* S *and an integer value* k *to determine if there is any* $X \subset S$ *such that* $\sum_{x \in X} x = k$.

In the general case when k is unconstrained this problem is a classic NP hard problem. However, our goals here are more modest. We are going to consider the case where we include k is the work bounds. We show that as long as k is polynomial in n then the work is also polynomial in n. Solutions of this form are often called *pseudo-polynomial* work (time) solutions.

This problem can be solved by brute force simply considering all possible subsets. This clearly takes exponential time. For a more efficient solution, let's again consider an inductive solution to the problem. You might consider some form of greedy method greedily takes elements from S. Unfortunately this does not work.

We therefore consider a divide-and-conquer solution. Naively this will also lead to exponential work, but by reusing subproblems we can show that it results in an algorithm that requires only

$O(|S|k)$ work. The idea is to take one element a out of S and consider the two possibilities: either a is included in X or not. For each these two possibilities we make a recursive call—in one case we subtract a from k ($a \in X$) and in the other case we leave k as is ($a \notin X$). Here is an algorithm based on this idea. It assumes the input is given as a list (the order of the elements of S in the list does not matter):

SS$(S, begin, end, k)$

```
1  Base Case
2  if S[begin] > k
3     then return SS(S, begin + 1, end, k)
4  if (SS(S, begin + 1, end, k − a) or SS(S, begin + 1, end, k))
5     then return true
6     else  return false
```

The $begin$ and end variables are indices specifying which subset of integers in S we are using to compute subset sum.

So what is the work of this algorithm? Well it leads to a binary recursion tree that might be n deep. This would imply something like $\theta(2^n)$ work, which is not good. Nevertheless, like the Rot-Cutting problem, the The key observation, however, is that there is a overlap in the subproblems.

To see that, let's think about how many distinct instances of SS can there be. Well for an initial instance SS(S_s, k_s) there are are only $|S_s|$ distinct lists that are ever used (each suffix of S_s). Furthermore k only decreases and never goes below 0, so it can take on at most $k_s + 1$ values. Therefore the total number of possible unique instances of SS is $|S_s|(k_s + 1) = O(k|S_s|)$.

We will again use the bottom-up method to take advantage of this sharing. In particular, we can construct a 2-dimensional table, where each cell $ss(i, j) =$ solution to SS$(S[1..i], j)$ for some j such that $0 \leq j \leq k$:

BOTTOMUPSS(S, i, j, k)

```
 1  let n ← |S|
 2  for i ← 0 to n
 3       do ss[i, 0] ← true
 4  for j ← 0 to k
 5       do ss[0, j] ← false
 6  for i ← 1 to n
 7       do for j ← 1 to k
 8            do if j > S[i]
 9                 then ss[i, j] ← ss[i − 1, j] or ss[i − 1, j − S[i]]
10                 else  ss[i, j] ← ss[i − 1, j]
11  return ss[n, k]
```

Notice that the way we fill this table is very similar to what we did with Minimum Edit Distance; both have the same dependency structure — each cell depends on cells that's above and to the left of it. Thus, so you can parallelize SS exactly the same way.

What if you want to be able to reconstruct which integers from the set to use in order to get the subset sum? Similar to what we have done with the Rod-Cutting problem, we will keep another table that remembers which choice we made to get to the existing solution — $t[i, j]$ is $true$ if we had used $S[i]$ and $false$ otherwise. Assuming $ss[n, k]$ is $true$ (i.e., a solution exists), then we can reconstruct the solution by following t backward — if $t[i, j] = true$, then include $S[i]$ in the solution set and follows $t[i-1, j-S[i]]$ next; otherwise, do not include $S[i]$ and follows $t[i-1, j]$ next.

Exercise 2 *Write the pseudo code for* BOTTOMUPSSEXTENDED *that also constructs table* t *that one can use later to reconstruct the subset that sums to* k*.*

The Viterbi Algorithm

The Viterbi algorithm uses dynamic programming to solve the "Hidden Markov Models (HMM)" and has a lot of applications in AI and natural language processing. The HMM problem is defined as follows. You are given a finite state machine with hidden states — you can think of it as a graph $G = (V, E)$ with directed edges, where a node $v \in V$ corresponds to a possible state, and each edge $e \in E$ is labeled with a symbol (from a finite set of possible symbols) and a probability. There can be multiple edges from node i to node j labeled with different symbols.

The states are hidden in the sense that, when a state transition is made, you can't tell the source and destination states for the transition; rather the only thing you observe is the symbol along the edge corresponding to the transition. We also know the probability of taking an edge given the current state (so the probabilities on all outgoing edges for a given node need to sum up to 1). The question is, given a sequence of observed symbols $H[1..n]$ and the probability matrix for taking a certain transition, can we figure out the most likely path taken that generates the observed symbols?

To make your life easier (and mine), we can assume that we are given a 3-dimensional probability matrix P, where the first dimension is indexed by the symbol observed, and the second is indexed by the source state, and the last is indexed by the destination state. For example, $P[A][i][j]$ specifies the probability of observed symbol A while transitioning from state i to state j. We also know a set of possible starting states, each has probability $S[i]$ for $i = 1..|V|$ of being the actual starting state.

Let's first think about how we can solve this problem inductively. Say we want to compute the probability of an event e — the maximum likelihood path that ending at state j after observing symbol $H[1..k]$. What information do we need to compute the probability for e? Well, the only way to for e to occur is if the maximum likelihood path ending at state i after observing $H[1..k-1]$, and the maximum likelihood path subsequently takes edge (i, j), with a symbol $H[k]$ along the edge.

Then, the $probability(e) = P[H[k]][i][j] \times probability($maximum likelihood path ending in state i after observing $H[1..k-1])$. So inductively, we can compute this if we know the probability of the maximum likelihood path ending in state i after observing $H[1..k-1]$. Note that there can be multiple possible candidates for such a state i, and our goal is to maximize the probability of e.

Knowing how to solve this inductively helps us to come up with a DP solution.

```
VITERBI(P, H)
 1  Initialize a table T of size (|H| + 1)|V|
 2  T[0][i] ← S[i]
 3  for k ← 1 to |H|
 4       do for j ← 1 to |V|
 5               do q ← −∞
 6                  for i ← 1 to |V|
 7                       do p ← P[H[k]][i][j]
 8                          if P[H[k]][i][j] ≠ 0
 9                            then q ← Max(q, p ∗ T[k − 1][j])
10                  T[k][j] ← q
11  return MAX-REDUCE(T[k])
```

Notice that the dependencies here differ from what we have seen so far — to compute $T[k][j]$, we may potentially need all cells in $T[k-1]$, not just $T[k-1][i]$ for $i = 1..j$. Given the dependencies, one way to parallelize it is to parallelize the computation for each row in parallel (i.e., change the second loop to a **parallel_for**.) We can also parallelize the inner-most loop if we use a temporary array and apply the MAX-REDUCE on the resulting array. What's the work and span if we do that? The work is $\theta(n^3)$ and the span is $\theta(n \lg n)$, so the parallelism is $\theta(n^2/\lg n)$.

Exercise 3 *The* VITERBI *algorithm that we have shown you only computes the maximum likelihood of seeing a sequence of symbols, but it does not tell you the sequence of states we visit in order to get that sequence. Think about how we can construct a table that would allow us to backtrack and reconstruct the path.*