Section 5: Advanced Dynamic Programming

CS 124

March 3, 2021
Outline

1 Dynamic Programming Review

2 Dynamic Programming on Subsets
   - Example: Team Formation
   - Bitmasks

3 Dynamic Programming on Trees
   - Introduction
   - Example: Maximum Weighted Matching

4 Problems
   - Problem 1: Optimal Taxation
   - Problem 2: Stack the Blocks
   - Problem 3: Longest Paths in a Tree
# Table of Contents

1. **Dynamic Programming Review**

2. **Dynamic Programming on Subsets**
   - Example: Team Formation
   - Bitmasks

3. **Dynamic Programming on Trees**
   - Introduction
   - Example: Maximum Weighted Matching

4. **Problems**
   - Problem 1: Optimal Taxation
   - Problem 2: Stack the Blocks
   - Problem 3: Longest Paths in a Tree
The central idea of dynamic programming is to break a large problem into many sub-problems, and then solve those subproblems in order to build up to a solution.
dp[0] dp[n]
\[ dp[0] \rightarrow dp[1] \]

\[ dp[n - 1] \rightarrow dp[n] \]
# Table of Contents

1. Dynamic Programming Review

2. Dynamic Programming on Subsets
   - Example: Team Formation
   - Bitmasks

3. Dynamic Programming on Trees
   - Introduction
   - Example: Maximum Weighted Matching

4. Problems
   - Problem 1: Optimal Taxation
   - Problem 2: Stack the Blocks
   - Problem 3: Longest Paths in a Tree
Team Formation

A corporate restructuring is underway, and we’re reorganizing the company to optimize our profits. \( \mathcal{X} \) is the set of employees.

We have a real-valued function \( p : 2^{\mathcal{X}} \setminus \{\emptyset\} \to \mathbb{R} \), which tells us the profits that can be generated by a team whose members are a subset of the employees \( s \subseteq \mathcal{X} \).

The profits from the partition \( \mathcal{X} = S_1 \cup S_2 \ldots \cup S_k \) are

\[
p(S_1) + p(S_2) + \ldots + p(S_k).
\]

How do we maximize our profits?
**Brute-Force**

- In order for our algorithm’s input size to be $n$, we let $|\mathcal{X}| = \log_2 n$.

- One approach is to use brute force, and iterate over all partitions of $\mathcal{X}$.

- **Challenge:** Prove there are $n^{O(\log \log n)}$ distinct partitions of $\mathcal{X}$. 
For a faster algorithm, we will use dynamic programming.

Let \( dp[S] \) be the most profit we can get from our teammates in the subset \( S \subseteq X \).

Then, we have

\[
\begin{align*}
    dp[\emptyset] &= 0, \\
    dp[S] &= \max_{S' \subseteq S} p(S') + dp[S \setminus S'].
\end{align*}
\]

What’s the run-time of this algorithm?
Analysis

- There are $n$ states, and each transition takes at most $n$ time, so the run time is $O(n^2)$. However, the analysis actually gets us a little more than that. Namely, note that there are $\sum_{i=0}^{\log_2{n}} 2^i$ sets of size $i$, and for each of those states they take $2^i$ iterations, so

$$\sum_{i=0}^{\log_2{n}} 2^i = (1 + 2)\log_2{n} = n\log_2(3) \approx n^{1.585},$$

which is a smaller polynomial.
Analysis

- There are \( n \) states, and each transition takes at most \( n \) time, so the run time is \( O(n^2) \).

- However, the analysis actually gets us a little more than that. Namely, note that there are \( \binom{\log_2 n}{i} \) sets of size \( i \), and for each of those states they take \( 2^i \) iterations, so

\[
\sum_{i=0}^{\log_2 n} \binom{\log_2 n}{i} 2^i = (1 + 2)^{\log_2 n} = n^{\log_2(3)} \approx n^{1.585},
\]

which is a smaller polynomial.
Implementation Notes

- In practical applications, we have to decide how to represent each subset. Simply using the set to, say, key a map is quite inefficient.

- For example, in this problem we’d like iterate through each $S \subseteq X$, $S' \subseteq S$ efficiently.

- In order to obtain memory savings, we represent subsets with binary representations. This is a somewhat common trick which appears in many settings. One name for this trick is called “bitmasks”.
A story

In high school, I was part of an AI Othello tournament.

How do we represent a grid efficiently? The most standard approach is to represent the positions of the light counters with an array, say

\[
[(3, 3), (3, 4), (4, 4), (5, 5), (6, 4), (6, 5)]
\]
Using Binary

One approach is to use a *bitboard*, so the light counters define a number (with binary) and the dark counters also define a number.

![Bitboard with binary representation](image)

The light counters correspond to the squares

\[ [18, 19, 27, 36, 43, 44]. \]

The binary representation is thus

\[ 2^{18} + 2^{19} + 2^{27} + 2^{36} + 2^{43} + 2^{44}. \]
Another benefit of this integer representation is that certain set operations can be expressed as bitwise operations. For example, \( S \subseteq S' \) becomes

\[
f(s, s') := (b(s) \& \sim b(s')) = 0,
\]

where \( b \) is the binary form of the subset \( s \). One way to see this is to look at this bit-by-bit.

For example, something being zero means that each bit is set to zero.

For \( f(s, s')_i = 1 \), we would need \( b(s)_i = 1 \) and \( b(s')_i = 0 \) for some \( i \), which is exactly precluded by \( S \subseteq S' \), as then \( i \in S \) and \( i \notin S' \).
In what follows, let $x = |\mathcal{X}|$.

```c
//set all dp[i] to negative infinity
dp[0] = 0
for (i = 0; i < 2^x; i++){
    for (j = 1; j < 2^x; j++){
        if (j & ~i == 0)
            dp[i] = max(dp[i], dp[i - j] + p[j])
    }
}
return  dp[2^x - 1]
```
How do we optimize this?

Note that we are blindly checking each $j$ to see if it's a subset of $i$, and this algorithm is still $O(n^2)$. Surely there's a better way to do this?

```cpp
// set all dp[i] to negative infinity
dp[0] = 0
for (i = 0; i < 2^x; i++) {
    for (j = i; j > 0; j = (j - 1) & i)
        dp[i] = max(dp[i], dp[i - j] + p[j])
}
return dp[2^x - 1]
```

**Challenge:** Show that this iterates through all desired $j$. Can you think of a way of replacing $i - j$ with something else, to make it faster?
# Table of Contents

1. Dynamic Programming Review

2. Dynamic Programming on Subsets
   - Example: Team Formation
   - Bitmasks

3. Dynamic Programming on Trees
   - Introduction
   - Example: Maximum Weighted Matching

4. Problems
   - Problem 1: Optimal Taxation
   - Problem 2: Stack the Blocks
   - Problem 3: Longest Paths in a Tree
Many problems are particularly natural on trees. Dynamic Programming is often a very useful technique on these problems, where the subproblems are given over the subtrees.

- Each of the subproblems is solving the problem over a given subtree.
- At each step, we solve the problem over each “direct” subtree, and then aggregate those results into a solution for this subtree.
Pseudocode, first attempt

```
dfs(v){
  for (vertex j: N(v)){
    dfs(j);
    // calculate dp[v] with dp[j]
  }
}
dfs(root)
```

What’s the problem with this algorithm?
Pseudocode, fixed

dfs(v, p) {
    for (vertex j: N(v)) {
        if (j != p) dfs(j, v);
        // calculate dp[v] with dp[j]
    }
}
dfs(root, nil)
Maximum Weighted Matching

We have a tree. What’s the highest-weight matching in this tree? A *matching* is a set of edges with no vertex overlap.

*Figure: A tree.*
Algorithm

To solve this problem, we write $dp[v][0]$ to be the largest matching without using the root, and $dp[v][1]$ to be the largest matching using the root.

\[
dp[v][0] = \sum_{j \in N(v), j \neq p} \max(\ dp[i][0], dp[j][1])
\]

\[
dp[v][1] = \max_{j \in N(v)} \left( w_{v,j} + dp[i][0] \sum_{k \in N(v), k \neq p,j} \max(dp[k][0], dp[k][1]) \right)
\]
Algorithm

To solve this problem, we write $dp[v][0]$ to be the largest matching without using the root, and $dp[v][1]$ to be the largest matching using the root.

Then, we have that

$$dp[v][1] = \max_{u \in N(v), u \neq p} \left( w_{u,v} + dp[u][0] + \sum_{j \in N(v), j \neq u, p} \max(dp[j][0], dp[j][1]) \right)$$

and

$$dp[v][0] = \sum_{j \in N(v), j \neq p} \max(dp[j][0], dp[j][1]).$$

and the answer is $\max (dp[r][0], dp[r][1]).$
Runtime Analysis

What’s the runtime of this algorithm?

\[
dfs(v, p)\{ \\
\text{for} \ (\text{vertex } j: N(v))\{ \\
\quad \text{if } (j != p) \ dfs(j, v); \\
\quad \text{//calculate } dp[v] \text{ with } dp[j] \\
\}\} \\
\}
\]
\[dfs(root, nil)\]

\[O(n^2)\]
\[O(n) \text{ operations}\]
Runtime Analysis

What’s the runtime of this algorithm?

```
dfs(v, p){
    for (vertex j: N(v)){
        if (j != p) dfs(j, v);
        //calculate dp[v] with dp[j]
    }
}
dfs(root, nil)
```

Since each vertex \(j\) which is not the root has \(dfs\) called on it exactly once, the total amount of calls across all internal loops is \(O(n)\). However, we need to make sure that the internal processing is also \(O(d)\) where \(d\) is the number of neighbors, or we might be in trouble!
Worked Example

Dynamic Programming on Trees
Example: Maximum Weighted Matching

\[ C \]

\[
\begin{align*}
[0,6] & \quad 10 & \quad [15,19] \\
4 & \quad [0,0] \\
\times 100 & \quad [0,0] \\
\end{align*}
\]

\[
\begin{align*}
18 & \quad 12 & \quad [0,9] \\
100 & \quad [0,0] \\
\times 6 & \quad 9 \\
\end{align*}
\]

\[ [a, b] \text{ with root } \\
\text{without root} \]
Table of Contents

1 Dynamic Programming Review

2 Dynamic Programming on Subsets
   - Example: Team Formation
   - Bitmasks

3 Dynamic Programming on Trees
   - Introduction
   - Example: Maximum Weighted Matching

4 Problems
   - Problem 1: Optimal Taxation
   - Problem 2: Stack the Blocks
   - Problem 3: Longest Paths in a Tree
You are the king of a kingdom that’s set up like a tree, and you want to collect some taxes. You know that each city has a “tax potential” \( t_i \) that you can collect. However, you don’t want to tax adjacent cities, because they might talk to each other and begin a revolt. Find the maximum tax you can collect in linear time.
Problem 2: Stack the Blocks

We have $n$ blocks, each with weight $w_i$ and strength $b_i$. We stack the blocks on top of each other in some order.

The power of a block is its strength minus the sum of the weights which lie on it.

The strength factor of a stack is the minimum power of any particular block.

Give a $O(n \cdot 2^n)$ algorithm to find the maximum possible strength factor across all permutations containing all of the blocks (note: answer might be negative).
Problem 3: Longest Path in a Tree

Find the length of the longest path in a tree, where the edges are weighted.