CS 161 Section 6

CA: [name of CA]
Agenda

1. Dynamic Programming
2. Graphs
   a. Bellman-Ford
   b. Floyd-Warshall
3. Section Problems
Dynamic Programming
What is *dynamic programming*?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually it is for solving **optimization problems**
  - eg, *shortest* path, or *longest* common subsequence
  - (Fibonacci numbers aren’t an optimization problem, but they are a good example…)


Bottom up approach
what we just saw.

● For Fibonacci:
  ● Solve the small problems first
    ○ fill in F[0], F[1]
  ● Then bigger problems
    ○ fill in F[2]
  ● …
  ● Then bigger problems
    ○ fill in F[n-1]
  ● Then finally solve the real problem.
    ○ fill in F[n]
Top down approach

- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..

- The difference from divide and conquer:
  - **Memo-ization**
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
What have we learned?

● Dynamic programming:
  ○ Paradigm in algorithm design.
  ○ Uses optimal substructure
  ○ Uses overlapping subproblems
  ○ Can be implemented bottom-up or top-down.
  ○ It’s a fancy name for a pretty common-sense idea:

Don’t duplicate work if you don’t have to!
Longest Common Subsequence

● Subsequence:
  ○ BDFH is a subsequence of ABCDEFGH

● If X and Y are sequences, a common subsequence is a sequence which is a subsequence of both.
  ○ BDFH is a common subsequence of ABCDEFGH and of ABDFGH

● A longest common subsequence…
  ○ …is a common subsequence that is longest.
  ○ The longest common subsequence of ABCDEFGH and ABDFGH is ABDFGH.
Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
Recipe for applying Dynamic Programming

- **Step 1:** Identify *optimal substructure.*
Step 1: Optimal substructure

Prefixes:

<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>C</td>
<td>T</td>
</tr>
</tbody>
</table>

Notation: denote this prefix ACGC by $Y_4$

- Our sub-problems will be finding LCS’s of prefixes to X and Y.
- Let $C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$
Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
Goal

- Write $C[i,j]$ in terms of the solutions to smaller sub-problems

$$C[i,j] = \text{length_of_LCS}(X_i, Y_j)$$
Two cases

Case 1: $X[i] = Y[j]$

- Our sub-problems will be finding LCS’s of prefixes to $X$ and $Y$.
- Let $C[i,j] = \text{length_of_LCS}(X_i, Y_j)$

Then $C[i,j] = 1 + C[i-1,j-1]$.

- because $\text{LCS}(X_i, Y_j) = \text{LCS}(X_{i-1}, Y_{j-1})$ followed by $A$
Two cases

Case 2: X[i] \(!=\) Y[j]

- Our sub-problems will be finding LCS’s of prefixes to X and Y.
- Let \( C[i,j] = \text{length\_of\_LCS}(X_i, Y_j) \)

![Diagram showing two sequences and their comparison](image)

- Then \( C[i,j] = \max\{ C[i-1,j], C[i,j-1] \} \).
  - either \( \text{LCS}(X_i, Y_j) = \text{LCS}(X_{i-1}, Y_j) \) and \( T \) is not involved,
  - or \( \text{LCS}(X_i, Y_j) = \text{LCS}(X_i, Y_{j-1}) \) and \( A \) is not involved,
- (maybe both are not involved, that’s covered by the “or”).

These are not the same
Recursive formulation
of the optimal solution

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j - 1], C[i - 1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases}
\]
Recipe for applying Dynamic Programming

• **Step 1**: Identify optimal substructure.
• **Step 2**: Find a recursive formulation for the length of the longest common subsequence.
• **Step 3**: Use dynamic programming to find the length of the longest common subsequence.
LCS DP  OMG BBQ

- **LCS**\( (X, Y) \):
  - \( C[i,0] = C[0,j] = 0 \) for all \( i = 1,\ldots,m \), \( j=1,\ldots,n \).
  - For \( i = 1,\ldots,m \) and \( j = 1,\ldots,n \):  
    - If \( X[i] = Y[j] \):
      - \( C[i,j] = C[i-1,j-1] + 1 \)
    - Else:
      - \( C[i,j] = \max\{ C[i,j-1], C[i-1,j] \} \)

Running time: \( O(nm) \)
Example

C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{ C[i-1, j], C[i, j-1], C[i-1, j-1] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases}
Example

So the LCS of X and Y has length 3.

\[ c[i,j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j - 1], C[i - 1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 
\end{cases} \]
Recipe for applying Dynamic Programming

- **Step 1**: Identify optimal substructure.
- **Step 2**: Find a recursive formulation for the length of the longest common subsequence.
- **Step 3**: Use dynamic programming to find the length of the longest common subsequence.
- **Step 4**: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
Finding an LCS

- See lecture notes for pseudocode
- Takes time $O(mn)$ to fill the table
- Takes time $O(n + m)$ on top of that to recover the LCS
  - We walk up and left in an $n$-by-$m$ array
  - We can only do that for $n + m$ steps.
- Altogether, we can find LCS($X,Y$) in time $O(mn)$. 
Time and Space complexity

• If we are only interested in the length of the LCS:
  • Since we go across the table one-row-at-a-time, we can only keep two rows if we want.
  • If we want to recover the LCS, we need to keep the whole table.

• Can we do better than \( O(mn) \) time?
  • A bit better.
    • By a log factor or so.
  • But doing much better (e.g. \( O(mn^{0.9}) \)) is an open problem!
    • If you can do it let me know 😊
What have we learned?

● We can find LCS(X,Y) in time $O(nm)$
  ○ if $|Y|=n$, $|X|=m$

● We went through the steps of coming up with a dynamic programming algorithm.
  ○ We kept a 2-dimensional table, breaking down the problem by decrementing the length of X and Y.
Graphs: Bellman-Ford and Floyd-Warshall
Shortest path DP by recipe

- **Step 1:**
  
  **Optimal substructure:** shortest path using \( \leq i \) edges

- **Step 2:**
  
  Suppose we already know \( d^i(s,u) \) for fixed \( s \) and all \( u \)

  **Recursive formulation:** \( d^{i+1}(s,v) = \min_u \{d^i(s,u)+w(u,v)\} \)

- **Step 3+4:** Later…
Step 3: write the algorithm

Bellman-Ford(G,s):

- \(d^{(0)}[v] = \infty\) for all \(v\) in \(V\)  // initialize:
- \(d^{(0)}[s] = 0\)

- **For** \(i=0,...,n-2\):
  - \(d^{(i+1)}[v] = d^{(i)}[v]\) for all \(v\) in \(V\)  // baseline distance:
    - \(v\) doesn’t need \((i+1)\)\(^{th}\) edge
  
- **For** \(v\) in \(V\):
  - **For** \(u\) in \(v\).neighbors:
    - \(d^{(i+1)}[v] \leftarrow \min(d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))\)  // found a better path through \(u\)

- **Return** \(d^{(n-1)}\)
Bellman-Ford take-aways

● Running time is $O(mn)$
  ○ For each of $n$ rounds, update $m$ edges.

- For $i=0,...,n-1$:
  - For $u$ in $V$:
    - For $v$ in $u$.neighbors:

$$m = \# \text{ of edges} = \frac{1}{2} \sum_{v \in V} \text{degree}(v)$$

● Works fine with negative edges.
● Does not work with negative cycles.
  ○ But it can detect negative cycles!
Note on implementation

- Don’t actually keep all n arrays around.
- Just keep two at a time: “last round” and “this round”

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>Stanford</th>
<th>Point Reyes</th>
<th>S.F.</th>
<th>e</th>
<th>Yosemite</th>
<th>Yellowstone</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d^{(0)}$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>$d^{(1)}$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>$\infty$</td>
<td>-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d^{(2)}$</td>
<td>0</td>
<td>-5</td>
<td>2</td>
<td>7</td>
<td>-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d^{(3)}$</td>
<td>-4</td>
<td>-5</td>
<td>-4</td>
<td>6</td>
<td>-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d^{(4)}$</td>
<td>-4</td>
<td>-5</td>
<td>-4</td>
<td>6</td>
<td>-7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only need these two in order to compute $d^{(4)}$.
Floyd-Warshall Algorithm
Another example of DP

● This is an algorithm for **All-Pairs Shortest Paths (APSP)**
  ○ That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  ○ Not just from a special single source s.

```
<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>v</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>u</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>v</td>
<td>0</td>
<td>0</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

![Graph diagram](image)
Floyd-Warshall Algorithm
Another example of DP

• This is an algorithm for **All-Pairs Shortest Paths** (APSP)
  • That is, I want to know the shortest path from $u$ to $v$ for **ALL pairs** $u,v$ of vertices in the graph.
  • Not just from a special single source $s$.

• Naïve solution:
  • For all $s$ in $G$:
    • Run Bellman-Ford on $G$ starting at $s$.

• Time $O(n \cdot nm) = O(n^2 m)$,
  • may be as bad as $n^4$ if $m = n^2$

Can we do better?
**Floyd-Warshall algorithm**

- Initialize $n$-by-$n$ arrays $D^{(k)}$ for $k = 0,...,n$
  - $D^{(k)}[u,u] = 0$ for all $u$, for all $k$
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all $k$
  - $D^{(0)}[u,v] = \text{weight}(u,v)$ for all $(u,v)$ in $E$.
- For $k = 1, ..., n$:
  - For pairs $u,v$ in $V^2$:
    - $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$
- Return $D^{(n)}$

The base case checks out: the only path through zero other vertices are edges directly from $u$ to $v$. 
We’ve basically just shown

● Theorem:
If there are no negative cycles in a weighted directed graph G, then the Floyd-Warshall algorithm, running on G, returns a matrix \( D^{(n)} \) so that:

\[
D^{(n)}[u,v] = \text{distance between } u \text{ and } v \text{ in } G.
\]

● Running time: \( O(n^3) \)
  ○ Better than running Bellman-Ford n times!

● Storage:
  ○ Need to store two \( n \)-by-\( n \) arrays, and the original graph.

As with Bellman-Ford, we don’t really need to store all \( n \) of the \( D^{(k)} \).
Recap of today’s lecture

- **Shortest Path in weighted graph w/ dynamic programming**
  - **Bellman-Ford**: Single Source Shortest Path (SSSP)
    - **Optimal substructure**: shortest path with \( \leq i \) edges
    - Run time: \( O(nm) \)
  - **Floyd-Warshall**: All Pairs Shortest Path (APSP)
    - **Optimal substructure**: shortest path using vertices \( \{1,...,k-1\} \)
    - Run time \( O(n^3) \)
Thank you!