Lecture 20: Dynamic Programming II: Longest Common Subsequence, Parent Pointers

Lecture Overview

- Review of big ideas & examples so far
- Bottom-up implementation
- Longest common subsequence
- Parent pointers for guesses

Readings

CLRS 15

Summary

* DP $\approx$ “controlled brute force”
* DP $\approx$ guessing + recursion + memoization
* DP $\approx$ dividing into reasonable $\sharp$ subproblems whose solutions relate - acyclicly - usually via guessing parts of solution.
* $\text{time} = \# \text{subproblems} \times \text{time/subproblem}$
  treating recursive calls as $O(1)$
  (usually mainly guessing)

- essentially an amortization
- count each subproblem only once; after first time, costs $O(1)$ via memoization
Examples:

<table>
<thead>
<tr>
<th></th>
<th>Fibonacci</th>
<th>Shortest Paths</th>
<th>Crazy Eights</th>
</tr>
</thead>
<tbody>
<tr>
<td>subprobs:</td>
<td>( \text{fib}(k) )</td>
<td>( \delta_k(s,t) \forall s,t, k &lt; n )</td>
<td>( \text{trick}(i) = \text{longest} )</td>
</tr>
<tr>
<td></td>
<td>( 0 \leq k \leq n )</td>
<td>= \text{min path } s \rightarrow t )</td>
<td>( \text{trick from card}(i) )</td>
</tr>
<tr>
<td>using ( k ) edges</td>
<td>using ( k ) edges</td>
<td>\text{using } k \text{ edges}</td>
<td>\text{using } k \text{ edges}</td>
</tr>
<tr>
<td># subprobs:</td>
<td>( \Theta(n) )</td>
<td>( \Theta(V^2) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>guessing:</td>
<td>none</td>
<td>edge from ( s ), if any</td>
<td>next card ( j )</td>
</tr>
<tr>
<td># choices:</td>
<td>1</td>
<td>( \deg(s) )</td>
<td>( n - i )</td>
</tr>
<tr>
<td>relation:</td>
<td>= ( \text{fib}(k - 1) ) + ( \text{fib}(k - 2) )</td>
<td>= ( \text{min}{\delta_{k-1}(s,t)} )</td>
<td>= ( 1 + \text{max}(\text{trick}(j)) )</td>
</tr>
<tr>
<td></td>
<td>( \leq \text{edges} )</td>
<td>( { w(s,v) + \delta_{k-1}(v,t) } )</td>
<td>for ( i &lt; j &lt; n ) if</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { v \in \text{Adj}[s] } )</td>
<td>( \text{match}(c[i], c[j]) )</td>
</tr>
<tr>
<td>time/subpr:</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1 + \deg(s)) )</td>
<td>( \Theta(n - i) )</td>
</tr>
<tr>
<td>DP time:</td>
<td>( \Theta(n) )</td>
<td>( \Theta(VE) )</td>
<td>( \Theta(n^2) )</td>
</tr>
</tbody>
</table>
| orig. prob:         | \( \text{fib}(n) \) | \( \delta_{n-1}(s,t) \) | \( \text{max}\{\text{trick}(i)\}, 0 \leq i < n \} \)
| extra time:         | \( \Theta(1) \) | \( \Theta(1) \) | \( \Theta(n) \) |

**Bottom-up implementation of DP:**

- subproblem dependencies form DAG (see Figure 1)
- imagine topological sort
- iterate through subproblems in that order
  \( \Rightarrow \) when solving a subproblem, have already solved all dependencies
- often just: “solve smaller subproblems first”

\[
\text{Figure 1: DAG}
\]

Example.

Fibonacci:

\[
\text{for } k \text{ in range}(n + 1): \text{fib}[k] = \cdots
\]

Shortest Paths:

\[
\text{for } k \text{ in range}(n): \text{ for } v \text{ in } V : d[k,v,t] = \cdots
\]

Crazy Eights:

\[
\text{for } i \text{ in reversed(range}(n)): \text{trick}[i] = \cdots
\]
no recursion for memoized tests
\[ \implies \text{faster in practice} \]

- building DP table of solutions to all subprobs. can often optimize space:
  - Fibonacci: PS6
  - Shortest Paths: re-use same table \( \forall k \)

### Longest common subsequence: (LCS)

A.K.A. edit distance, diff, CVS/SVN, spellchecking, DNA comparison, plagiarism, detection, etc.

Given two strings/sequences \( x \) & \( y \), find longest common subsequence LCS\((x,y)\) sequential but not necessarily contiguous

- e.g., H I E R O G L Y P H O L O G Y vs. M I C H A E L A N G E L O
  common subsequence is Hello

- equivalent to “edit distance” (unit costs): \( \# \) character insertions/deletions to transform \( x \rightarrow y \) everything except the matches

- brute force: try all \( 2^{|x|} \) subsequences of \( x \) \( \implies \Theta(2^{|x|} \cdot |y|) \) time

- instead: DP on two sequences simultaneously

* Useful subproblems for strings/sequences \( x \):
  - suffixes \( x[i:] \)
  - prefixes \( x[:i] \)
    The suffixes and prefixes are \( \Theta(|x|) \) \( \implies \) use if possible
  - substrings \( x[i:j] \) \( \Theta(|x|^2) \)

Idea: Combine such subproblems for \( x \) & \( y \) (suffixes and prefixes work)

### LCS DP

- subproblem \( c(i,j) = | \text{LCS}(x[i:],y[j:])| \) for \( 0 \leq i,j < n \)
  \[ \implies \Theta(n^2) \text{ subproblems} \]
  - original problem \( \approx c[0,0] \) (length \( \sim \) find seq. later)

- idea: either \( x[i] = y[j] \) part of LCS or not \( \implies \) either \( x[i] \) or \( y[j] \) (or both) not in LCS (with anyone)

- guess: drop \( x[i] \) or \( y[j] \)? (2 choices)
• relation among subproblems:

\[
\begin{align*}
\text{if } x[i] = y[j] : c(i, j) &= 1 + c(i + 1, j + 1) \\
\text{(otherwise } x[i] \text{ or } y[j] \text{ unused \sim can't help)} \\
\text{else: } c(i, j) &= \max\{c(i + 1, j), c(i, j + 1)\}
\end{align*}
\]

base cases: \(c(|x|, j) = c(i, |y|) = \phi\)

\(\Rightarrow\) \(\Theta(1)\) time per subproblem

\(\Rightarrow\) \(\Theta(n^2)\) total time for DP

• DP table: see Figure 2

![Figure 2: DP Table](image)

- linear space via antidiagonal order

• recursive DP:

```python
def LCS(x, y):
    seen = {}
    def c(i, j):
        if i >= len(x) or j >= len(y):
            return \phi
        if (i, j) not in seen:
            if x[i] == y[j]:
                seen[i, j] = 1 + c(i + 1, j + 1)
            else:
                seen[i, j] = max(c(i + 1, j), c(i, j + 1))
        return seen[i, j]
    return c(0, 0)
```
Lecture 20  Dynamic Programming II of IV  6.006 Spring 2008

• bottom-up DP:

```python
def LCS(x, y):
    c = {}
    for i in range(len(x)):
        c[i, len(y)] = φ
    for j in range(len(y)):
        c[len(x), j] = φ
    for i in reversed(range(len(x))):
        for j in reversed(range(len(y))):
            if x[i] == y[j]:
                c[i, j] = 1 + c[i + 1, j + 1]
            else:
                c[i, j] = max(c[i + 1, j], c[i, j + 1])
    return c[0, 0]
```

Recovering LCS: [material covered in recitation]

• to get LCS, not just its length, store parent pointers (like shortest paths) to remember correct choices for guesses:

```python
if x[i] = y[j]:
c[i, j] = 1 + c[i + 1, j + 1]
parent[i, j] = (i + 1, j + 1)
else:
    if c[i + 1, j] > c[i, j + 1]:
        c[i, j] = c[i + 1, j]
        parent[i, j] = (i + 1, j)
    else:
        c[i, j] = c[i, j + 1]
        parent[i, j] = (i, j + 1)

...and follow them at the end:
```

```python
lcs = []
here = (0,0)
while c[here]:
    if x[here] == y[here]:
        lcs.append(x[here])
        here = parent[here]
```