Lecture 19: Dynamic Programming II: Shortest Paths, Longest Common Subsequence, Parent Pointers

Lecture Overview

- Review of big ideas & Examples
- Shortest Paths
- Bottom-up implementation
- Longest common subsequence
- Parent pointers for guesses

Quiz 2: Wednesday Nov 18, 2009 in room 34-101 from 7:30 pm - 9:30 pm.

Readings

CLRS 15

DP Review

* DP $\approx$ “controlled brute force”
* DP $\approx$ recursion + memoization
* DP $\approx$ dividing into reasonable $\#$ subproblems whose solutions relate - acyclicly - usually via guessing parts of solution.

* time $\approx \# \text{subproblems} \times \text{time/subproblem}$
  $\approx \# \text{subproblems} \times \# \text{guesses per subproblem} \times \text{overhead}$.

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

The table below shows the examples from last lecture.
### Examples: Fibonacci, Crazy Eights

<table>
<thead>
<tr>
<th>subprobs:</th>
<th>fib(k)</th>
<th>trick(i) = longest trick starting at card(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td># subprobs:</td>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td>guessing:</td>
<td>none</td>
<td>next card (j)</td>
</tr>
<tr>
<td># choices:</td>
<td>1</td>
<td>(n - i)</td>
</tr>
<tr>
<td>relation:</td>
<td>(= \text{fib}(k - 1)) + \text{fib}(k - 2))</td>
<td>(= 1 + \max{\text{trick}(j)}) for (i &lt; j &lt; n) if (\text{match}(c[i], c[j]))</td>
</tr>
<tr>
<td>time/subpr:</td>
<td>(\Theta(n))</td>
<td>(\Theta(n - i))</td>
</tr>
<tr>
<td>DP time:</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
</tr>
<tr>
<td>orig. prob:</td>
<td>fib(n)</td>
<td>(\max{\text{trick}(i), 0 \leq i &lt; n})</td>
</tr>
<tr>
<td>extra time:</td>
<td>(\Theta(1))</td>
<td>(\Theta(n))</td>
</tr>
</tbody>
</table>

### Shortest Paths to a given destination \(t\)

#### Recursive formulation:
- for all nodes \(v\):
  \[
  \delta(v, t) = \min\{w(v, u) + \delta(u, t) \mid (v, u) \in E\}
  \]  

- does this work with memoization?
  no: cycles \(\implies\) infinite loops. In Figure 4

  \[
  \delta(v_1, t) = 1 + \delta(v_2, t) = 2 + \delta(v_3, t) = 3 + \delta(v_1, t) = 4 + \delta(v_2, t) = \ldots
  \]

![Shortest Paths](image1.png)

\[\text{Figure 1: Shortest Paths}\]

#### Remedy?
A better definition:

\[
\delta_k(v, t) = \text{length of shortest path from } v \text{ to } t \text{ using } \leq k \text{ edges}
\]

New Recursion:
• $\delta_k(t, t) = 0$;
• $\delta_0(v, t) = +\infty$, for $v \neq t$;
• for all other pairs of values $v, k$

$$\delta_k(v, t) = \min \left\{ \{\delta_{k-1}(v, t)\} \cup \{w(v, u) + \delta_{k-1}(u, t) \mid (v, u) \in E\} \right\} \tag{2}$$

**Shortest path?** Assuming no negative cycles: $\delta(v, t) = \delta_{n-1}(v, t)$ for all $v$

**Runtime**

- *Naive analysis:* there are $O(V)$ values for $k$, $O(V)$ values for $v$, and every application of (2) takes time $O(V)$ in the worst case since there are $O(V)$ guesses for $u$; hence the overall time is $O(V^3)$.

- *Clever analysis:* For each value of $k$, each edge is “explored” once. Since there are $O(V)$ possible values of $k$, overall time is $O(VE)$.

<table>
<thead>
<tr>
<th>Examples:</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>$\delta_k(v, t) \forall v, k &lt; n$</td>
<td>trick($i$) = longest</td>
</tr>
<tr>
<td></td>
<td>$0 \leq k \leq n$</td>
<td>$= \text{min path } v \to t$</td>
<td>trick from card($i$)</td>
</tr>
<tr>
<td></td>
<td>using $\leq k$ edges</td>
<td></td>
<td></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 $v$, if any</td>
<td>next card $j$</td>
</tr>
<tr>
<td>$#$ choices:</td>
<td>1</td>
<td>deg($v$)</td>
<td>$n - i$</td>
</tr>
<tr>
<td>relation:</td>
<td>$= \text{fib}(k - 1)$</td>
<td>$= \text{min}{\delta_{k-1}(v, t)}$</td>
<td>$= 1 + \text{max(trick}(j))$</td>
</tr>
<tr>
<td></td>
<td>$+ \text{fib}(k - 2)$</td>
<td>$\cup{w(v, u) + \delta_{k-1}(u, t) \mid u \in \text{Adj}[v]}$</td>
<td>for $i &lt; j &lt; n$ if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>match($c[i], c[j]$)</td>
</tr>
<tr>
<td>time/subpr:</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1 + \frac{E}{V})$—on average</td>
<td>$\Theta(n - i)$</td>
</tr>
<tr>
<td>DP time:</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(V^2 + VE)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>orig. prob:</td>
<td>$\text{fib}(n)$</td>
<td>$\delta_{n-1}(v, t), \forall v$</td>
<td>$\max{\text{trick}(i), 0 \leq i &lt; n}$</td>
</tr>
<tr>
<td>extra time:</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Bottom-up implementation of DP:

So far: Recursion + Memoization

Alternative to recursion

- subproblem dependencies form DAG (see Figure 2)
- imagine topological sorting the dependency graph
- iterate through subproblems in that order
  \[\Rightarrow\] when solving a subproblem, have already solved all dependencies
- often: “solve smaller subproblems first”

```
Figure 2: DAG.
```

```
Figure 3: Subproblem Dependency Graph for Fibonacci Numbers.
```

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 subproblems
  \[\Rightarrow\] faster in practice
- building DP table of solutions to all subprobs. can often optimize space:
  - 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.)

**Input:** two strings/sequences $x$ & $y$

**Question:** the longest common subsequence of $x$ and $y$, denoted 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): minimum number of character insertions/deletions to transform $x$ to $y$ removes everything except the matches
- brute force: try all $2^{|x|}$ subsequences of $x$; for each of them scan $y$ to see if that subsequence exists in $y$ reduces to $\Theta(2^{|x|} \cdot |y|)$ time, where $|x|$ and $|y|$ represent the lengths of $x$ and $y$ respectively.
- instead: DP on two sequences simultaneously

**LCS DP**

- **Subproblem Definition:**
  
  $c[i,j] = LCS(x[i:], y[j:])$, for $0 \leq i,j < n$,

  where $x[i:]$ (resp. $y[j:]$) is the suffix of $x$ (resp. $y$) starting at position $i$ (resp. $j$).

  - $\Theta(n^2)$ subproblems
  - original problem $\approx c[0,0]$ (this gives the length; to find the sequence itself a little more book-keeping is needed)
  - **Recursion:** Forget about the original problem and focus on finding the LCS of $x[i:]$ and $y[j:]$. Look at the first positions of these sequences and distinguish the following cases:
    - if $x[i] = y[j]$, then “match” $x[i]$ and $y[j]$ and combine this with the longest common subsequence of $x[i+1:]$ and $y[j+1:]$;
    - if $x[i] \neq y[j]$, then it must be that $x[i]$ or $y[j]$ or both are NOT used in the longest common subsequence of $x[i:]$ and $y[j:]$—GUESS WHICH ONE TO DROP
- Hence, the **recursive formula** is the following:
if \( x[i] = y[j] \): 
\[
c[i,j] = 1 + c[i+1, j+1]
\]
else: 
\[
c[i,j] = \max\{c[i+1, j], c[i, j+1]\}
\]
base cases: \( c[|x|, j] = c[i, |y|] = \emptyset \)

- \( \Theta(1) \) time per subproblem \( \implies \Theta(n^2) \) total time for DP.
- DP table: See Figure 4 for subproblem dependency structure:

![Figure 4: DP Table.](image)

- recursive DP: implement the recursive formula for \( c[\cdot, \cdot] \) given above, memoizing all intermediated results

```python
def LCS(x, y):
    seen = { }
    def c[i, j]:
        if i ≥ len(x) or j ≥ len(y) : return \emptyset
        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(∅, ∅)
```

- bottom-up DP: fill in the table \( c[\cdot, \cdot] \) in a “bottom-up” fashion, that is paying attention to the dependency structure shown in Figure 4.
def LCS(x, y):
    c = {}
    for i in range(len(x)):
        c[i, len(y)] = ∅
    for j in range(len(y)):
        c[len(x), j] = ∅
    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[∅, ∅]

Recovering LCS: [material covered in recitation and discussed also in the next lecture]

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

  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:

  lcs = [ ]
  here = (∅, ∅)
  while c[here]:
      if x[i] == y[j]:
          lcs.append(x[i])
          here = parent[here]