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 ≈ "controlled brute force"

* DP ≈ recursion + memoization

* DP ≈ dividing into reasonable ♯ subproblems whose solutions relate - acyclicly - usually via guessing parts of solution.

* time ≈ ♯ subproblems × $\underbrace{\text{time/subproblem}}$
≈ ♯ subproblems × ♯ guesses per subproblem × 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
subprobs:	fib(k) $0 \leq k \leq n$	trick(i) = longest trick starting at card(i)
$\sharp$ subprobs:	$\Theta(n)$	$\Theta(n)$
guessing:	none	next card j
$\sharp$ choices:	1	$n - i$
relation:	$=$ fib$(k-1)$ $+$ fib$(k-2)$	$= 1 + \max(\text{trick}(j))$ for $i < j < n$ if match$(c[i], c[j])$
time/subpr:	$\Theta(n)$	$\Theta(n - i)$
DP time:	$\Theta(n^2)$	$\Theta(n^2)$
orig. prob:	fib(n)	$\max\{\text{trick}(i),\, 0 \leq i < n\}$
extra time:	$\Theta(1)$	$\Theta(n)$

Shortest Paths to a given destination t

Recursive formulation:

- for all nodes v:

$$\delta(v,t) = \min\{w(v,u) + \delta(u,t) \Big| (v,u) \,\epsilon\, E\} \tag{1}$$

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

$$\delta(v_1,t) = 1 + \delta(v_2,t) = 2 + \delta(v_3,t) = 3 + \delta(v_1,t) = 4 + \delta(v_2,t) = \dots$$

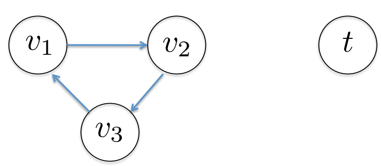

Figure 1: Shortest Paths

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:

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

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 $O(VE)$.

Examples:	Fibonacci	Shortest Paths	Crazy Eights
subprobs:	fib(k) $0 \leq k \leq n$	$\delta_k(v, t) \forall v, k < n$ $=$ min path $v \to t$ using $\leq k$ edges	trick(i) = longest trick from card(i)
$\sharp$ subprobs:	$\Theta(n)$	$\Theta(V^2)$	$\Theta(n)$
guessing:	none	edge from v, if any	next card j
$\sharp$ choices:	1	deg(v)	$n - i$
relation:	$=$ fib($k-1$) $+$ fib($k-2$)	$= \min\{\delta_{k-1}(v, t)\}$ $\cup \{w(v, u) + \delta_{k-1}(u, t)$ $\mid u \in \text{Adj}[v]\}$	$= 1 + \max(\text{trick}(j))$ for $i < j < n$ if match($c[i], c[j]$)
time/subpr:	$\Theta(n)$	$\Theta(1 + \frac{E}{V})$—on average	$\Theta(n - i)$
DP time:	$\Theta(n^2)$	$\Theta(V^2 + VE)$	$\Theta(n^2)$
orig. prob:	fib(n)	$\delta_{n-1}(v, t), \forall v$	$\max\{\text{trick}(i),\ 0 \leq i < n\}$
extra time:	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

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
  $\Longrightarrow$ when solving a subproblem, have already solved all dependencies
- often: "solve smaller subproblems first"

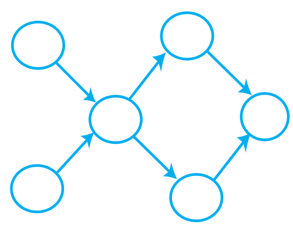

Figure 2: DAG.

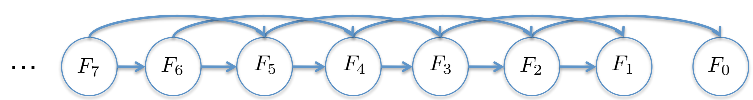

Figure 3: Subproblem Dependency Graph for Fibonacci Numbers.

Example.

Fibonacci:
  for k in range$(n+1)$: fib$[k] = \cdots$
Shortest Paths:
  for k in range(n): for v in V : $d[k, v, t] = \cdots$
Crazy Eights:
  for i in reversed(range(n)): trick$[i] = \cdots$

- no recursion for memoized subproblems
  $\Longrightarrow$ 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 ♯ character insertions/deletions to transform $x \rightarrow y$ $\longrightarrow$ 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 $y \implies \Theta(2^{|x|} \cdot |y|)$ time, where $|x|$ and $|y|$ represent the lengths of x and y respectively.
- instead: DP on two sequences simultaneously

LCS DP

- Subproblem Definition:

$$c[i,j] = \text{LCS}(x[i:], y[j:]),\ \text{for } 0 \leq i,j < n,$$

  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:

$$\text{if } x[i] = y[j] : c[i,j] = 1 + c[i+1, j+1]$$
$$\text{else: } c[i,j] = \max\{\underbrace{c[i+1,j]}_{x[i]\text{ out}}, \underbrace{c[i,j+1]}_{y[j]\text{ out}}\}$$
$$\text{base cases: } c[|x|, j] = c[i, |y|] = \emptyset$$

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

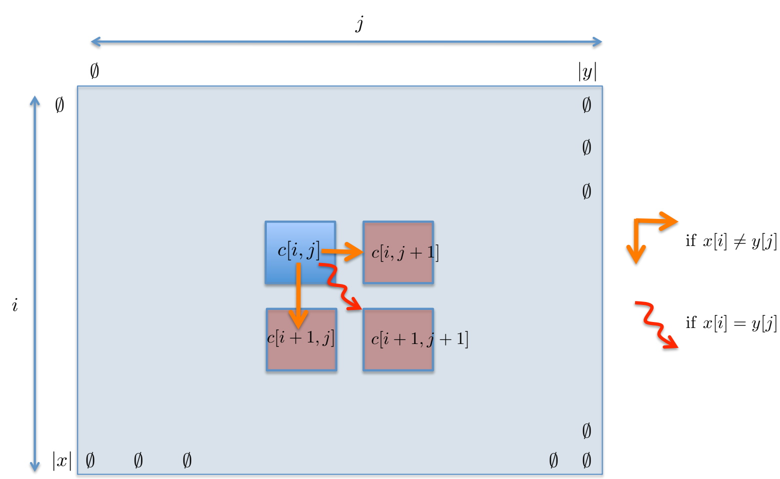

Figure 4: DP Table.

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

```
def LCS(x, y):
    seen = { }
    def c[i, j]:
        if i ≥ len(x) or j ≥ len(y) :  return ∅
        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)] = Ø
    for j in range(len(y)):
        c[len(x), j] = Ø
    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[Ø, Ø]
```

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