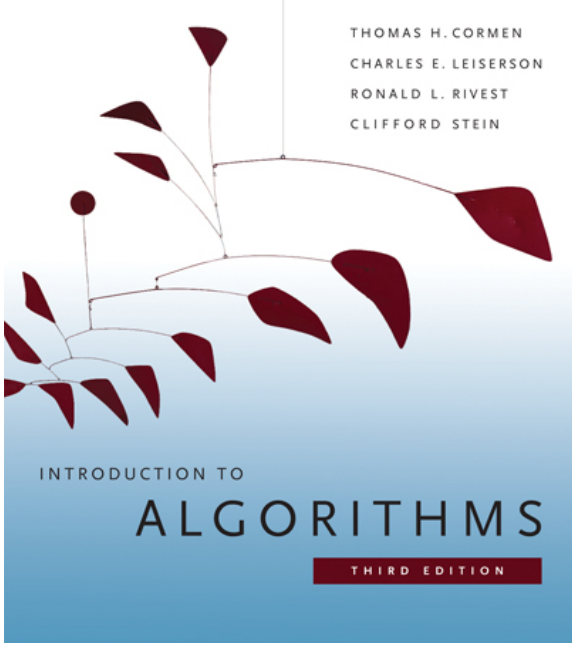
6.006- *Introduction to Algorithms*



Lecture 18

Prof. Constantinos Daskalakis

CLRS 15

Menu

- New technique: Dynamic Programming
  - Computing Fibonacci numbers – Warmup
  - “Definition” of DP
  - Crazy Eights Puzzle

Fibonacci Numbers

- Fibonacci sequence:
  - $F_0=0$, $F_1=1$
  - $F_n=F_{n-1}+F_{n-2}$
- How fast does Fn grow ?
  - $F_n=F_{n-1}+ F_{n-2} \geq 2\ F_{n-2}$ $\Rightarrow F_n= 2^{\Omega(n)}$
- How quickly can we compute F_n?
  (time measured in arithmetic operations)

$F_n = F_{n-1} + F_{n-2}$

- Algorithm I: recursion

```
naive_fibo(n):
    if n=0: return 0
    else if n=1: return 1
    else:
      return naive_fibo(n-1) + naive_fibo(n-2)
```

- Time ? $O(F_n)$
- Better algorithm ?

$F_n = F_{n-1} + F_{n-2}$

- Algorithm II: memoization

memo = { }
fibo(i):
    if i in memo: return memo[i]
    else if i=0: return 0
    else if i=1: return 1
    else:
        f = fibo(i-1) + fibo(i-2)
    memo[i]=f
    return f
return fibo(n)

- Time? O(n)

- in the whole recursive execution, I will only go beyond this point, n times
(since every time I do this, I fill in another slot in memo[])

- hence, all other calls to fibo( ) act as reading an entry of an array

Dynamic Programming

- DP ≈ Recursion + Memoization
- DP works when:
  - the solution can be produced by combining solutions of subproblems; $F_n = F_{n-1} + F_{n-2}$
  - the solution of each subproblem can be produced by combining solutions of sub-subproblems, etc;

moreover…. $F_{n-1} = F_{n-2} + F_{n-3}$ $F_{n-2} = F_{n-3} + F_{n-4}$

  - the total number of subproblems arising recursively is polynomial.

$F_1, F_2, \ldots, F_n$

Dynamic Programming

- DP ≈ Recursion + Memoization
- DP works when:

Optimal substructure

The solution to a problem can be obtained by solutions to subproblems.

$F_n = F_{n-1} + F_{n-2}$

moreover….

Overlapping Subproblems

A recursive solution contains a "small" number of distinct subproblems (repeated many times)

$F_1, F_2, \ldots, F_n$

Crazy 8s

- **Input:** a sequence of cards c[0]…c[n-1].
- E.g., 7♣ 7♥ K♣ K♠ 8♥ …
- **Goal:** find the longest "trick subsequence" $c[i_1]\ldots c[i_k]$, where $i_1 < i_2 < \ldots < i_k$.
- For it to be a trick subsequence, it must be that: $\forall\, j$, $c[i_j]$ and $c[i_{j+1}]$ "match" i.e.
  - they either have the same rank,
  - or the same suit
  - or one of them is an 8
  - in this case, we write: $c[i_j] \sim c[i_{j+1}]$
- E.g., 7♣ K♣ K♠ 8♥ is the longest such subsequence in the above example

Algorithm

- Let trick(i) be the length of the longest trick subsequence that starts at card c[i]
- **Question:** How can I relate value of trick(i) with the values of trick(i+1),…,trick(n)?
- Recursive formula:

$$\text{trick}(i) = 1 + \max_{j>i,\ c[i] \sim c[j]} \text{trick}(j)$$

- Maximum trick length:

$$\max_i \text{trick}(i)$$

Implementations

Recursive

- memo = { }
- trick(i):
  - if i in memo: return memo[i]
  - else if i=n-1: return 1
  - else
    - f := 1+max$_{j>i,\ c[i] \sim c[j]}$ trick(j)
    - memo[i] := f
    - return f
- call trick(0)
- return maximum value in memo

Implementations (cont.)

Iterative

memo = { }

for i=n-1 downto 0

  memo[i]= 1+max$_{j>i,\ c[i] \sim c[j]}$ memo[j]

return maximum value in memo

Runtime: $O(n^2)$

Dynamic Programming

- DP ≈ Recursion + Memoization
- DP works when:

Optimal substructure

An solution to a problem can be obtained by solutions to subproblems.

trick(i) = 1+ $\max_{j>i,\ c[i] \sim c[j]}$ trick(j)

moreover….

Overlapping Subproblems

A recursive solution contains a "small" number of distinct subproblems (repeated many times)

trick(0), trick(1),…, trick(n-1)

Menu

- New technique: Dynamic Programming
  - Computing Fibonacci numbers – Warmup
  - “Definition” of DP
  - Crazy Eights Puzzle
  - Next Time: **all-pairs shortest paths**

All-pairs shortest paths

- **Input:** Digraph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \to \mathbb{R}$.
- **Output**: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

Assumption: No negative-weight cycles

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where $a_{ij}=w(i,j)$ *if* $(i,j) \in E$, and define
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that uses <u>at most</u> m edges

Claim: We have

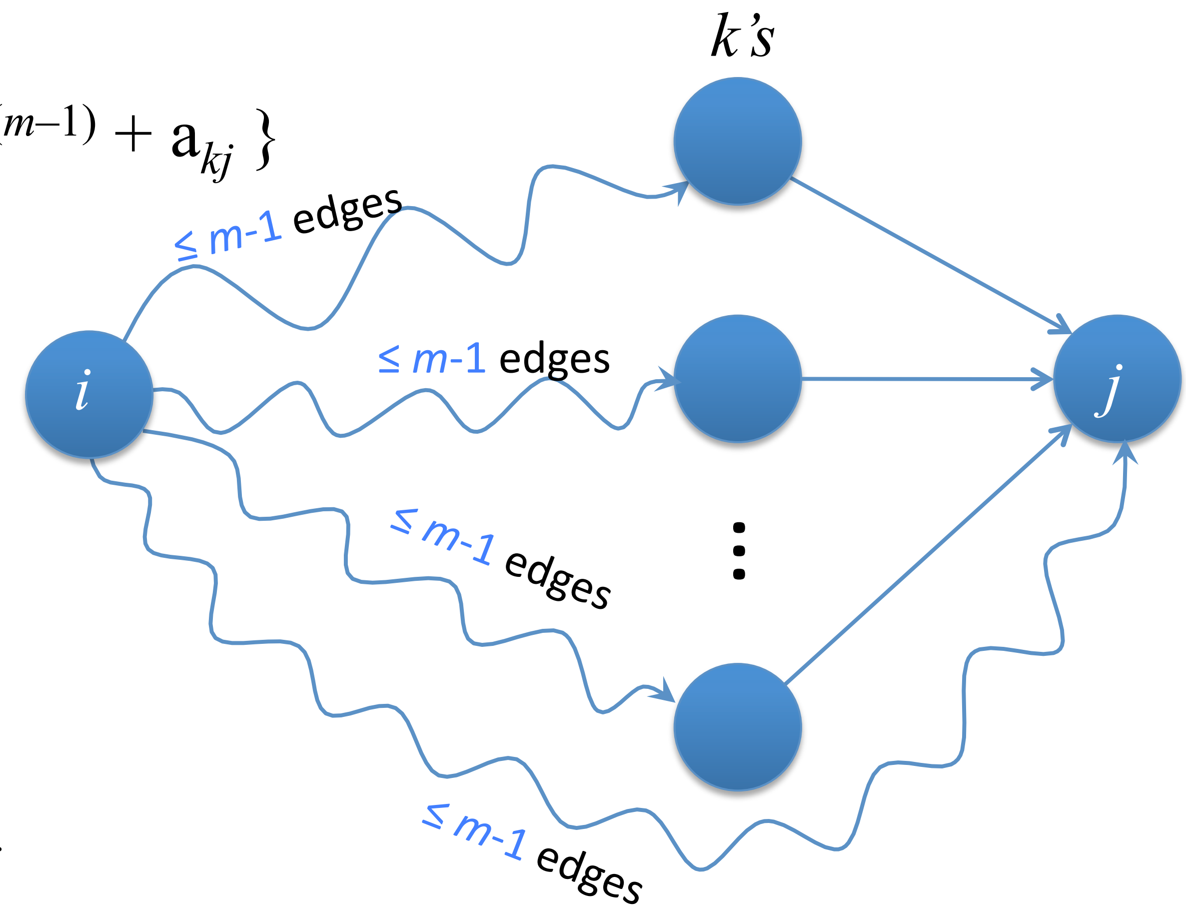
$$d_{ij}^{(0)} = 0, \text{ if } i = j, \text{ and } \infty, \text{ if } i \neq j;$$

and for $m = 1, 2, \ldots, n-1$,

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.$$

Proof of Claim

$$d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}$$



for $k \leftarrow 1$ to n
    if $d_{ij} > d_{ik} + a_{kj}$
        $d_{ij} \leftarrow d_{ik} + a_{kj}$

Relaxation

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where $a_{ij}=w(i,j)$ $if\ (i,j) \in E$, and define
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that uses at most m edges

Claim: We have

$$d_{ij}^{(0)} = 0, \text{ if } i = j, \text{ and } \infty, \text{ if } i \neq j;$$

and for $m = 1, 2, \ldots, n-1$,

$$d_{ij}^{(m)} = \min_k\{d_{ik}^{(m-1)} + a_{kj}\}.$$

Time? $O(n^4)$ - similar to n runs of Bellman-Ford