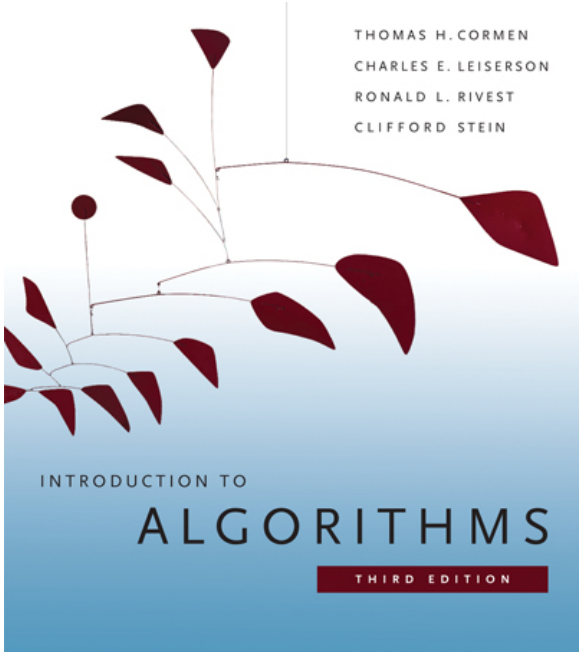
6.006- *Introduction to Algorithms*



Lecture 19

Prof. Patrick Jaillet

Lecture overview

Dynamic Programming II

- review
  - key aspects of Dynamic Programming (DP)
  - all-pairs shortest paths as a DP
- another DP for all-pairs shortest paths
- longest common subsequence

CLRS 15.3, 15.4, 25.1, 25.2

Dynamic Programming

- DP ≈ recursion + memoization
- Typically (not always) applied to optimization problems – ex Fibonaci, Crazy eight, SPPs

Optimal substructure

optimal solution to a problem can be obtained from optimal solutions to subproblems.

Overlapping subproblems

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

Dynamic Programming

- DP ≈ recursion + memoization
- DP works when:
  - the solution can be produced by combining solutions of subproblems;
  - the solution of each subproblem can be produced by combining solutions of sub-subproblems, etc;

  moreover …. is efficient if
  - the total number of subproblems arising recursively is polynomial.

All-pairs shortest paths

- **Input:** Directed graph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \to \mathrm{R}$

- **Output**: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$

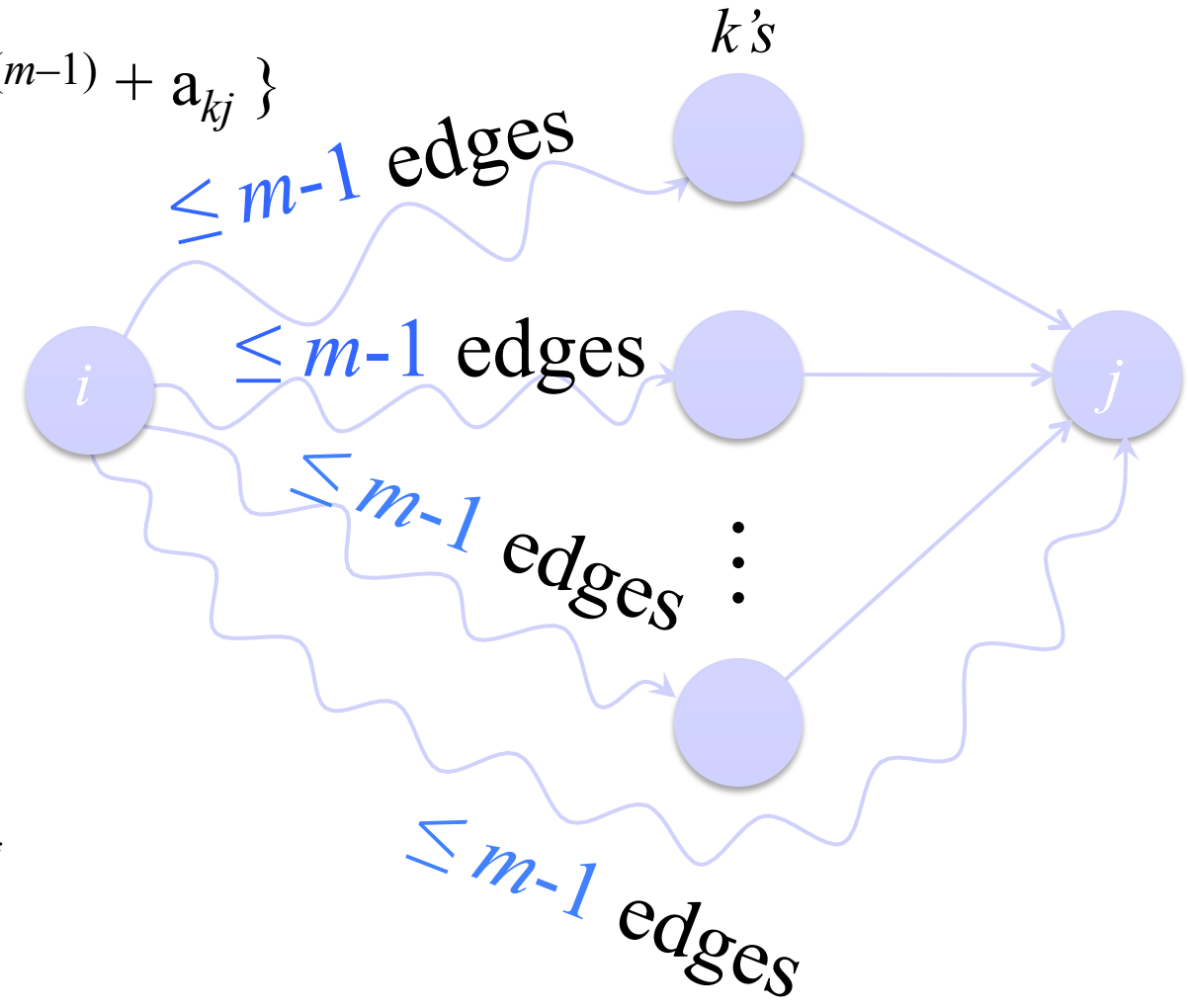
  where $a_{ij} = w(i,j)$ *if* $(i,j) \in E$.
- Define $d_{ij}^{(m)}$ = weight of a shortest path from i to j that uses at most m edges
- We have
  - $d_{ij}^{(0)} = 0$, if $i = j$,
  - and $d_{ij}^{(0)} = \infty$ otherwise;

CLAIM:

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

DP Approach, running time

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

We have

– $d_{ij}^{(0)} = 0$, if $i = j$,

– and $d_{ij}^{(0)} = \infty$ otherwise;

<u>CLAIM</u>:

for $m = 1, 2, \ldots, n-1$, $d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}$

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

Dynamic Programming

- DP ≈ recursion + memoization
- Typically (not always) applied to optimization problems

Optimal substructure ?

optimal solution to a problem can be obtained from optimal solutions to subproblems.

Overlapping subproblems ?

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

Another DP Approach

- Consider again the $n \times n$ matrix $A = (a_{ij})$
  where $a_{ij} = w(i,j)$ *if* $(i,j) \in E$.
- Define $d_{ij}^{(k)}$ = weight of a shortest path from i to j for which all intermediate vertices are chosen among $\{1,2, \ldots, k\}$

- CLAIM:
  - $\mathbf{d_{ij}^{(k)} = a_{ij}}$ **if** $\boldsymbol{k=0}$,
  - $\mathbf{d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}}$ **if** $\boldsymbol{k \geq 1}$;

$O(n^3)$ running time (Floyd-Warshall)

Dynamic Programming

- DP ≈ recursion + memoization
- Typically (not always) applied to optimization problems

Optimal substructure ?

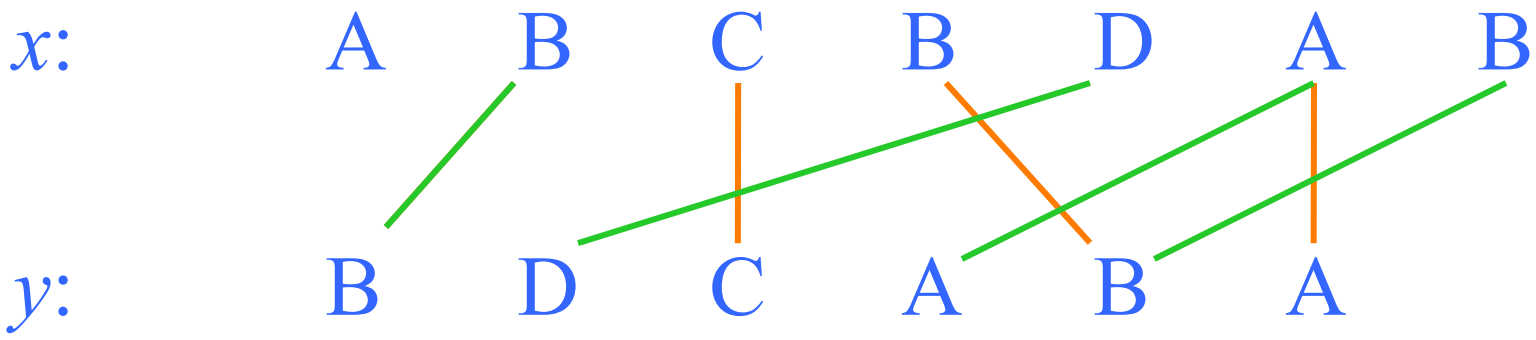
optimal solution to a problem can be obtained from optimal solutions to subproblems.

Overlapping subproblems ?

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

Longest Common Subsequence

- given two sequences x[1..m] and y[1..n], find a longest subsequence LCS(x,y) common to both:



- denote the length of a sequence *s* by |*s*|
- let us first try to get |LCS(x,y)|

Brute force solution

- For every subsequence of x[1..m] , check if it is a subsequence of y[1..n]
- Analysis
  - 2^m subsequences of x
  - each check takes $O(n)$ time ...
  - worst case running time is $O(n2^m)$
- **Pretty bad (brute!)**

Using prefixes

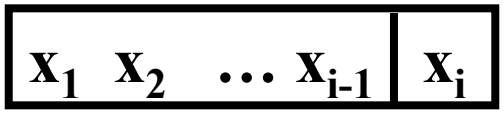
- consider prefixes of x and y
  - x[1..i] ith prefix of x[1..m]
  - y[1..j] jth prefix of y[1..n]

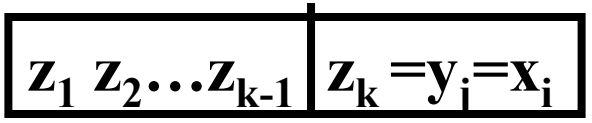
- define c[*i*,*j*] = |LCS(x[1..*i*],y[1..*j*])|

- so c[*m*,*n*] = |LCS(x,y)|

- recurrence?

1) x[1..i] and y[1..j] end with $x_i = y_j$





Z_k is Z_{k-1} followed by $z_k = y_j = x_i$ where
Z_{k-1} is an LCS of x[1..i-1] and y[1..j-1]

$$c(i, j) = c(i\text{-}1, j\text{-}1)+1$$

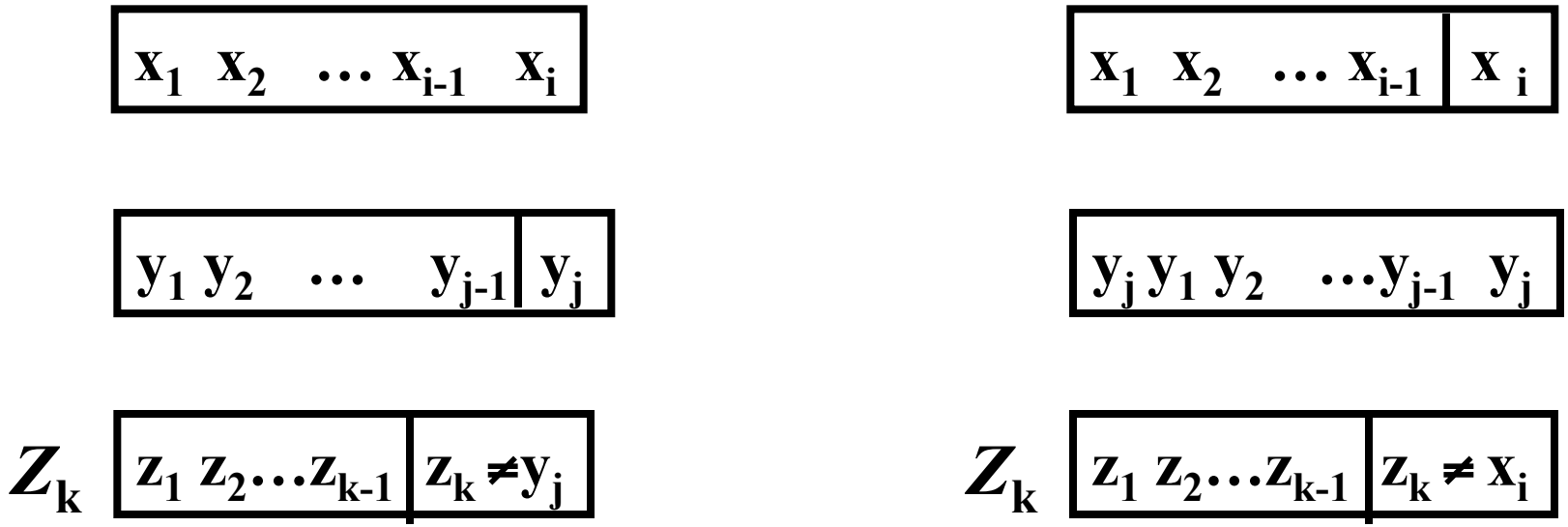
Example - use of property 1

x: B A N A N A

y: A T A N A

by inspection LCS of B A N and A T is A
so LCS(x,y) is A A N A

2) x[1..i] and y[1..j] end with $x_i \neq y_j$



LCS of x[1..i] and y[1..j-1]

LCS of x[1..i-1] and y[1..j]

$$c(i, j)=\max\{c(i, j\text{-}1), c(i\text{-}1, j)\}$$

Example: use of property 2

x: A B C D E F G

y: B C D G K

The last character of the LCS(x,y) either:

- ends with a G => can't end with a K
  - => can remove K from *y*
- doesn't end with a G
  - => can remove G from *x*

A recurrence, summary

- consider prefixes of x and y
  - x[1..i] ith prefix of x[1..m]
  - y[1..j] jth prefix of y[1..n]
- define c[*i*,*j*] = |LCS(x[1..*i*],y[1..*j*])|
  - so c[*m*,*n*] = |LCS(x,y)|
- recurrence:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise} \end{cases}$$

Solving LCS with DP

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise} \end{cases}$$

- running time is ....
  - $O(n \times m)$

Dynamic Programming

- DP ≈ recursion + memoization
- Typically (not always) applied to optimization problems

Optimal substructure ?

optimal solution to a problem can be obtained from optimal solutions to subproblems.

Overlapping subproblems ?

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