CS483 Analysis of Algorithms
Lecture 08 – Dynamic Programming 02 *

Jyh-Ming Lien

April 02, 2008

*this lecture note is based on *Algorithms* by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani and *Introduction to the Design and Analysis of Algorithms* by Anany Levitin.

Edit Distance

- Given two strings "lqorihten" and "algorithm", can you tell how similar these strings are?
- Edit distance is the number of operations (deletions, insertions, substitutions) that you can convert from one string to the other.

```
S – N O W Y          – S N O W – Y
S U N N – Y          S U N – – N Y
   Cost: 3               Cost: 5
```

- How do you compute the smallest edit distance between two strings?
  - Brute force method? What's the time complexity?
  - Greedy algorithm?
  - Dynamic programming

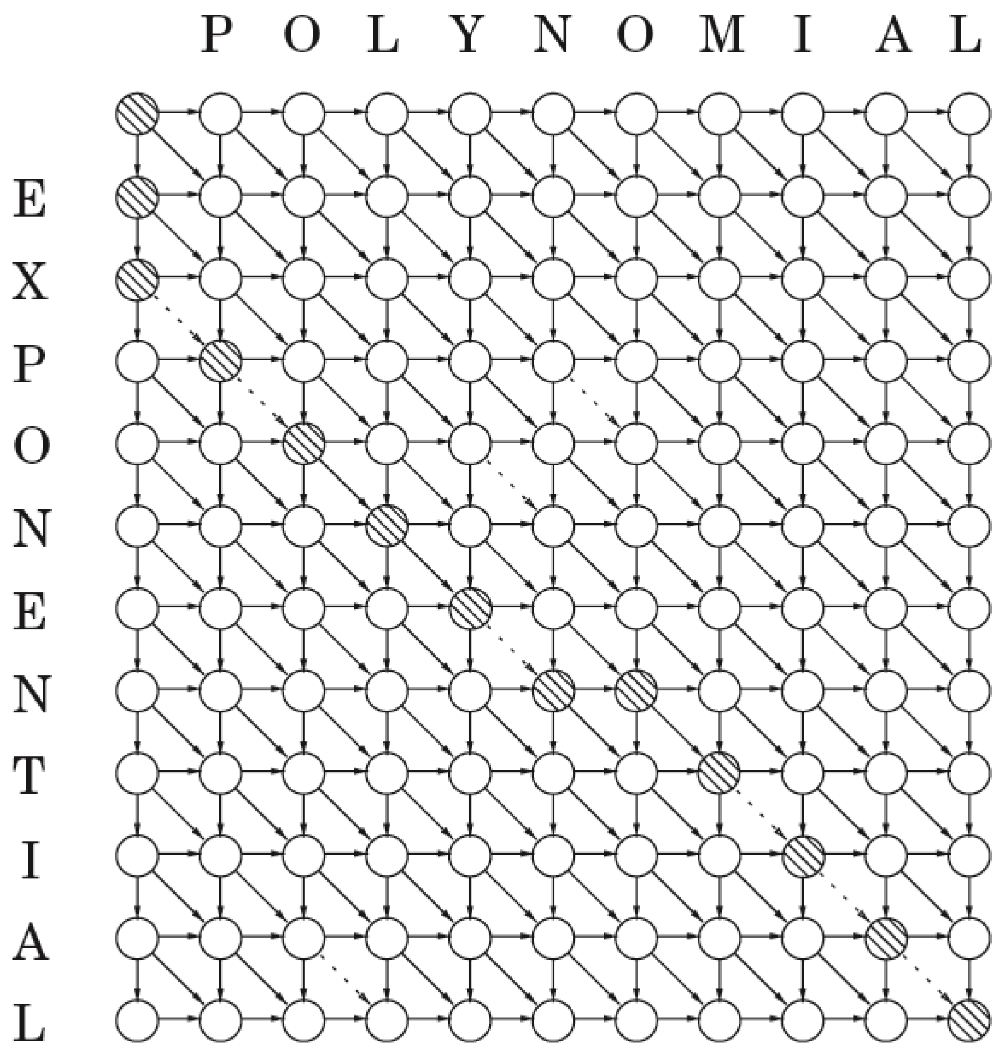
- **Observation**: Given two strings $x[1 \cdots n]$ and $y[1 \cdots m]$. No matter how we match x to y, at the end of the match, we can only have:
  - $x[n]$ matches to $y[m]$
  - $x[n]$ matches to empty
  - $y[m]$ matches to empty
  - **Question**: Is it possible $x[n]$ matches to $y[i < m]$? or $y[m]$ matches to $x[j < n]$?
- **Example**: EXPONENTIAL vs. POLYNOMIAL
  - What are possible endings?
  - What are the subproblems we should consider?
  - How do we get an optimal answer?

- Let $E(i, j)$ be the edit distance for the subproblem of strings with lengths i and j
- $E(i, j) = \min\{E(i-1, j-1) + \text{diff}(x[i], y[j]), E(i-1, j)+1, E(i, j-1)+1\}$

	□	P	O	L	Y	N	O	M	I	A	L
□											
E											
X											
P											
O											
N											
E											
N											
T											
I											
A											
L											

Edit Distance and DAG

- Representing the problem as a DAG



Chain Matrix Multiplication

- Given four matrices, $A[50 \times 20]$, $B[20 \times 1]$, $C[1 \times 10]$, $D[10 \times 100]$, we wish to compute $A \times B \times C \times D$.
- If we compute $(((A \times B) \times C) \times D)$, we will perform x multiplications?
- What about $((A \times B) \times (C \times D))$?
- How do we find the best way to group matrices so that the number of multiplications is minimized?
  - Brute force
  - Greedy algorithm
  - Dynamic programming

Chain Matrix Multiplication

- Dynamic programming and DAG
  1. A pair of parentheses groups two matrices
  2. The final matrix represents the root
  3. Example: $(((A \times B) \times C) \times D)$ and $((A \times B) \times (C \times D))$

- So, our goal is to build an optimal binary tree
- Given four matrices, $A[50 \times 20]$, $B[20 \times 1]$, $C[1 \times 10]$, $D[10 \times 100]$, we wish to compute $A \times B \times C \times D$.
  1. Subproblems with two matrices:
  2. Subproblems with three matrices:
  3. Subproblems with four matrices:

Chain Matrix Multiplication

- General cases: Give a list of matrices $\{A_i\}$, $C(i,j)$ be the minimum cost of $A_i \times \cdots A_j$, then

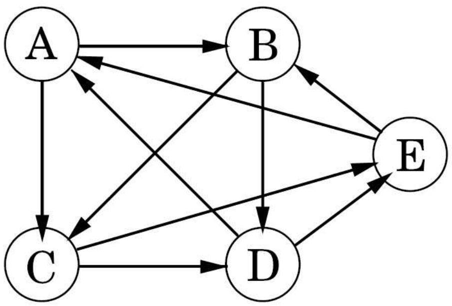
$$C(i,j) = \min_{i \leq k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

- Example: $A[50 \times 20]$, $B[20 \times 1]$, $C[1 \times 10]$, $D[10 \times 100]$

0	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$					
$i=2$					
$i=3$					
$i=4$					
$i=5$					

Transitive closure

- Transitive closure of a graph is a set of vertex pairs of a graph, which can be connected by one or multiple paths
- We can represent the transitive closure using a matrix A. The element $A_{i,j}$ is "1" if there are one or multiple paths between vertices i and j.
- Example:



- Can you design a brute force algorithm? What is its time complexity?

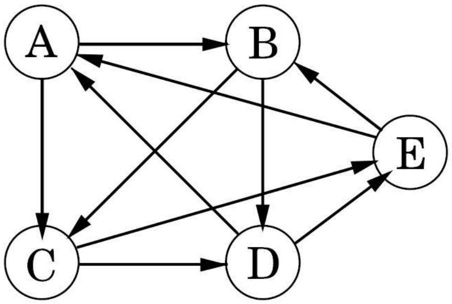
Warshall's Algorithm

- Important observation: If there is a path from a to z via s then there must be a path from a to s and from s to z
- Let A^k be the optimal answer when we only allow the first k nodes to be intermediate nodes in paths. We can compute the optimal solution for $k+1$ nodes A^{k+1} efficiently
- What is A^0?
- For $k > 0$,

$$A^{k+1}[i,j] = \begin{cases} 1 & (A^k[i,j] = 1) \\ A^k[i,k] \text{ and } A^k[k,j] & (\text{otherwise}) \end{cases}$$

Warshall's Algorithm

- Example:



via $\emptyset$	A	B	C	D	E
A					
B					
C					
D					
E					

via A	A	B	C	D	E
A					
B					
C					
D					
E					

via B	A	B	C	D	E
A					
B					
C					
D					
E					

via C	A	B	C	D	E
A					
B					
C					
D					
E					

via D	A	B	C	D	E
A					
B					
C					
D					
E					

via E	A	B	C	D	E
A					
B					
C					
D					
E					

- Algorithm

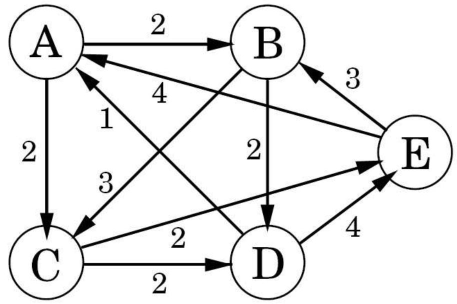
Algorithm 0.1: WARSHALL($A[1 \cdots n]$)

$$
\textbf{for } i \leftarrow \{1 \cdots n\} \\
\textbf{do} \begin{cases} \textbf{for } j \leftarrow \{1 \cdots n\} \\ \textbf{do} \begin{cases} \textbf{for } k \leftarrow \{1 \cdots n\} \\ \textbf{do } A^k[i,j] \leftarrow (A^{k-1}[i,k] \textbf{ and } A^{k-1}[k,j]) \textbf{ or } A^{k-1}[i,j] \end{cases} \end{cases}
$$

- Time complexity?

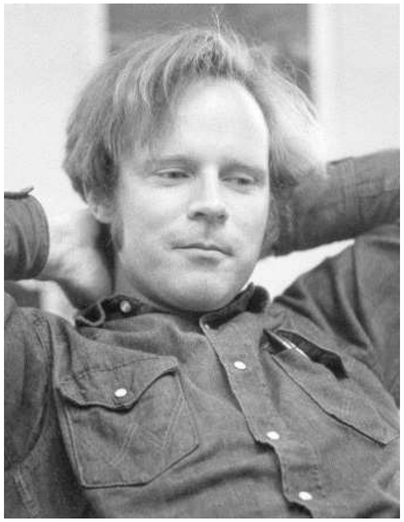
All-pairs Shortest path problem

- In this problem we want to find the shortest paths connecting all possible pairs of vertices of a graph
- Example:



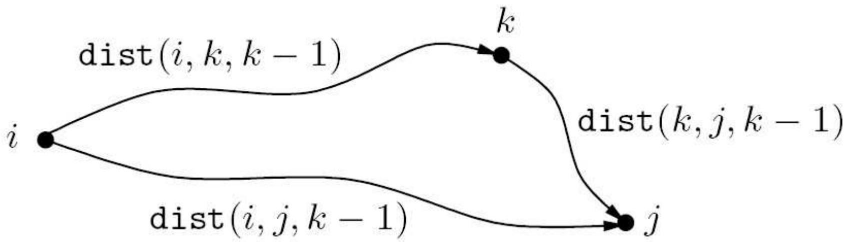
- What is the brute force algorithm and its time complexity?

- A.k.a. Floyd-Warshall algorithm (or the Roy-Floyd algorithm)
- Robert Floyd (1936-2001)



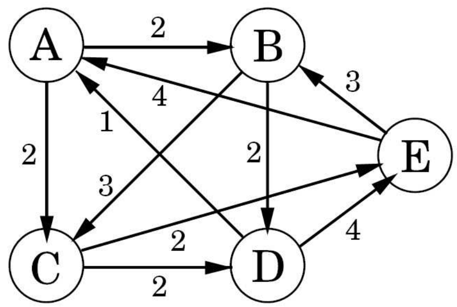
(Robert Floyd, 1972, from http://sigact.acm.org/floyd/)

- The algorithm is very similar to Warshall's algorithm
- **Basic idea**: Let A^{k-1} be the optimal answer when we only allow the first $k-1$ nodes to be intermediate nodes in paths. We can compute the optimal solution for k nodes A^k efficiently



Floyd’s Algorithm

- Example:



via $\emptyset$	A	B	C	D	E
A					
B					
C					
D					
E					

via A	A	B	C	D	E
A					
B					
C					
D					
E					

via B	A	B	C	D	E
A					
B					
C					
D					
E					

via C	A	B	C	D	E
A					
B					
C					
D					
E					

via D	A	B	C	D	E
A					
B					
C					
D					
E					

via E	A	B	C	D	E
A					
B					
C					
D					
E					

- Algorithm

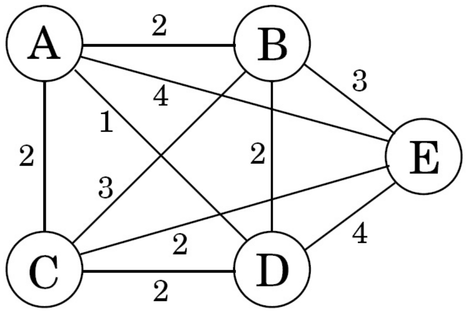
Algorithm 0.2: FLOYD$(A[1\cdots n])$

$$
\textbf{for } i \leftarrow \{1 \cdots n\}
$$
$$
\textbf{do} \begin{cases} \textbf{for } j \leftarrow \{1 \cdots n\} \\ \textbf{do} \begin{cases} \textbf{for } k \leftarrow \{1 \cdots n\} \\ \textbf{do } A^k[i,j] \leftarrow \min\{(A^{k-1}[i,k] + A^{k-1}[k,j]), A^{k-1}[i,j]\} \end{cases} \end{cases}
$$

- Time complexity?

Travelling Salesman Problem (TSP)

- **Problem**: Find the shortest path from A to A by visiting each vertex exactly once



- Brute force:
- Greedy:
- Dynamic programming:

Travelling Salesman Problem (TSP)

- Given a graph with n nodes and starting vertex is 1.
- Algorithm

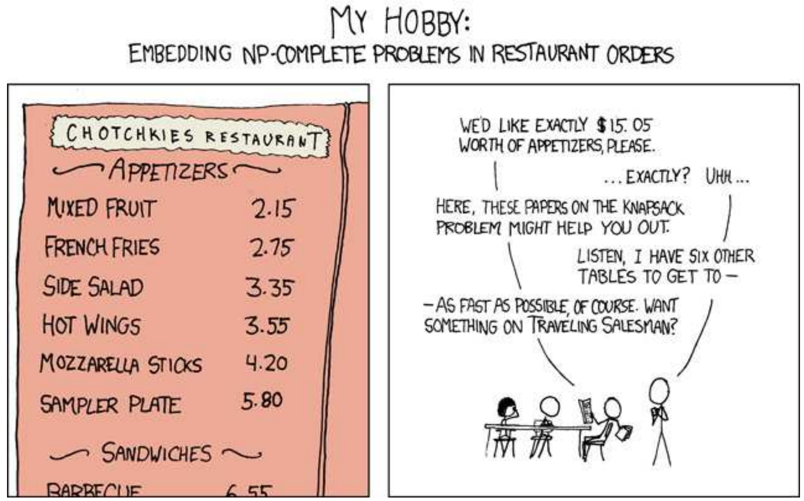
Algorithm 0.3: FLOYD($A[1 \cdots n]$)

for $s \leftarrow \{2 \cdots n\}$
  do { **for** all subsets $S \subset \{1, 2, \cdots, n\}$ of size s and containing 1
    do { **for** $j \in S$ **and** $j \neq 1$
      do $C(S, j) = \min_{i \in S, i \neq j} \{C(S - \{j\}, i) + d_{ij}\}$

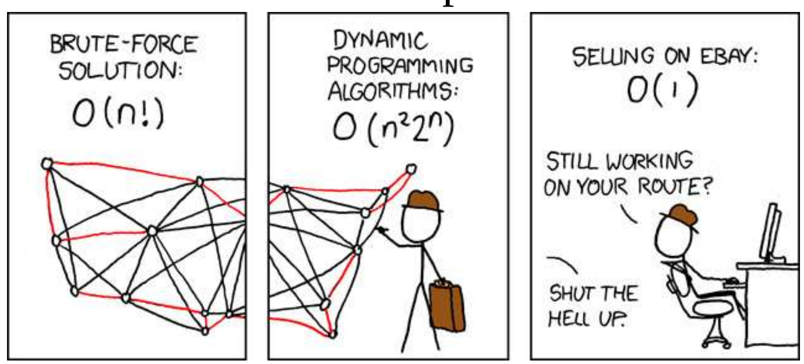
- Time complexity?

Conclusion

- Now you should understand these (better):



NP-Complete



Travelling Salesman Problem
(from Randall Munroe, creator of xkcd)

Conclusion

- We have solved the following problems using dynamic programming
  - Longest increasing sequence
  - Binomial coefficients of $(a+b)^n$ (Pascal's triangle)
  - Knapsack problem
  - Edit distance
  - Matrix multiplication chain (optimal binary tree)
  - Transitive closure (Warshall's algorithm)
  - All pairs shortest paths (Floyd's algorithm)
  - TSP
- It is usually more difficult to represent a problem as a set of sub-problems
- Next couple of weeks: Linear programming.