CS483 Design and Analysis of Algorithms

Lectures 15-16 Dynamic Programming

Instructor: Fei Li

lifei@cs.gmu.edu with subject: CS483

Office hours: STII, Room 443, Friday 4:00pm - 6:00pm or by appointments

Course web-site:

http://www.cs.gmu.edu/∼lifei/teaching/cs483_fall08/

Figures unclaimed are from books "Algorithms" and "Introduction to Algorithms"

Dynamic Programming

1. Shortest Path
2. Longest Increasing Subsequences
3. Edit Distance
4. Knapsack
5. Chain Matrix Multiplication
6. Independent Sets in Trees

Algorithmic Paradigms

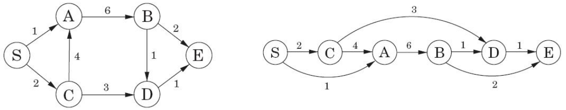
1. **Greed**
   Build up a solution incrementally, optimizing some local criterion in each step
2. **Divide-and-conquer**
   Break up a problem into two sub-problems, solve each sub-problem *independently*, and combine solution to sub-problems to form solution to original problem
3. **Dynamic programming**
   Identify a collection of subproblems and tackling them one by one, smallest first, using the answers to smaller problems to help figure out larger ones, until the whole lot of them is solved

Shortest Paths in Directed Acyclic Graphs (DAG)

Remark

The special distinguishing feature of a DAG is that its node can be linearized.

Figure 6.1 A dag and its linearization (topological ordering).



initialize all $\texttt{dist}(\cdot)$ values to ∞

$\texttt{dist}(s) = 0$

for each $v \in V \backslash \{s\}$, in linearized order:

$\quad \texttt{dist}(v) = \min_{(u,v) \in E} \{\texttt{dist}(u) + l(u,v)\}$

$$dist(D) = \min\{dist(B) + 1,\ dist(C) + 3\}.$$

This algorithm is solving a collection of *subproblems*, $\{dist(u) : u \in V\}$. We start with the smallest of them, $dist(s) = 0$.

Some Thoughts on Dynamic Programming

Remark

1. *In dynamic programming, we are not given a DAG; the DAG is implicit.*
2. *Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: If to solve subproblem B we need to answer the subproblem A, then there is a (conceptual) edge from A to B. In this case, A is thought of as a smaller subproblems than B — and it will always be smaller, in an obvious sense.*

Problem

How to solve/calculate above subproblems' values in Dynamic Programming?

Solution

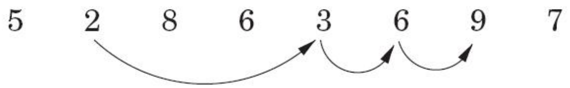
?

Longest Increasing Subsequences

Definition

The input is a sequence of numbers $a_1, \ldots, a_n$. A subsequence is any subset of these numbers taken in order, of the form $a_{i1}, a_{i2}, \ldots, a_{ik}$ where $1 < i_1 < i_2 < \ldots < i_k \leq n$, and an *increasing subsequence* is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length.

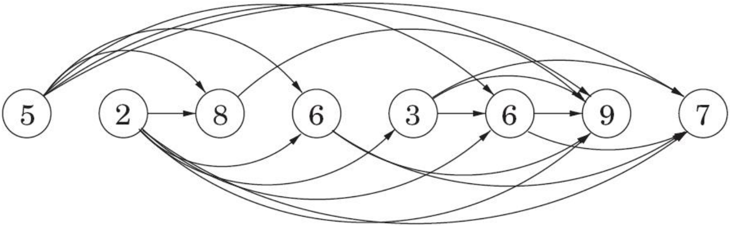
5, 2, 8, 6, 3, 6, 9, 7 is 2, 3, 6, 9:



Longest Increasing Subsequences

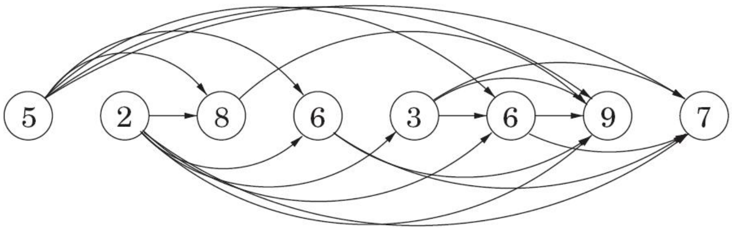
Longest Increasing Subsequences

Figure 6.2 The dag of increasing subsequences.



Longest Increasing Subsequences

1 **Figure 6.2** The dag of increasing subsequences.



2
```
function inc-subsequence(G = (V, E))

    for j = 1, 2, ... n
        L(j) = 1 + maxL(i): (i, j) ∈ E;

    return max_j L(j);
```

Longest Increasing Subsequences

Remark

There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

Theorem

The algorithm runs in polynomial time $O(n^2)$.

Proof.

$$L(j) = 1 + \max\{L(i) : (i, j) \in E\}.$$

□

Problem

Why not using recursion? For example,

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$

Solution

Bottom-up versus (top-down + divide-and-conquer).

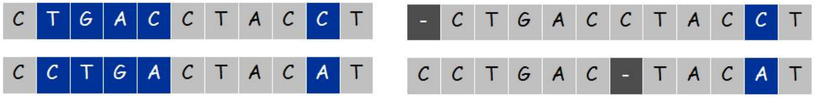
Announcements

1. Assignment 7 is released today. The due date is Nov. 19th.

Edit Distance

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

$2\delta + \alpha_{CA}$

from Wayne's slides on "Algorithm Design"

Problem

If we use the brute-force method, how many alignments do we have?

Solution

?

Edit Distance

1. **Goal**. Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$, find alignment of minimum cost. Call this problem $E(m, n)$.
2. **Subproblem** $E(i, j)$.
   Define $\mathit{diff}(i, j) = 0$ if $x[i] = y[j]$ and $\mathit{diff}(i, j) = 1$ otherwise

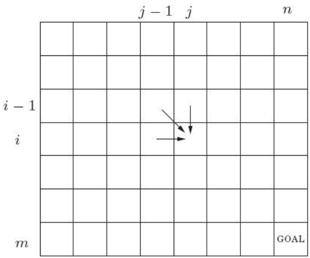
$$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \mathit{diff}(i, j) + E(i-1, j-1)\}$$

```
function edit-distance(X, Y)

    for i = 0, 1, 2, ... m
        E(i, 0) = i;

    for j = 1, 2, ... n
        E(0, j) = j;

    for i = 1, 2, ... m
        for j = 1, 2, ... n
            E(i, j) = min { E(i - 1, j) + 1, E(i, j - 1) + 1,
                    E(i - 1, j - 1) + diff(i, j) };

    return E(m, n);
```

Edit Distance

Figure 6.4 (a) The table of subproblems. Entries $E(i-1, j-1)$, $E(i-1, j)$, and $E(i, j-1)$ are needed to fill in $E(i, j)$. (b) The final table of values found by dynamic programming.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Theorem

`edit-distance` *runs in time* $O(m \cdot n)$

The Underlying DAG

Remark

Every dynamic program has an underlying DAG structure: Think of each node as representing a subproblem, and each edge as a precedence constraint on the order in which the subproblems can be tackled.
Having nodes $u_1, \ldots, u_k$ point to v means "subproblem v can only be solved once the answers to $u_1, u_2, \ldots, u_k$ are known".

Remark

Finding the right subproblems takes creativity and experimentation.

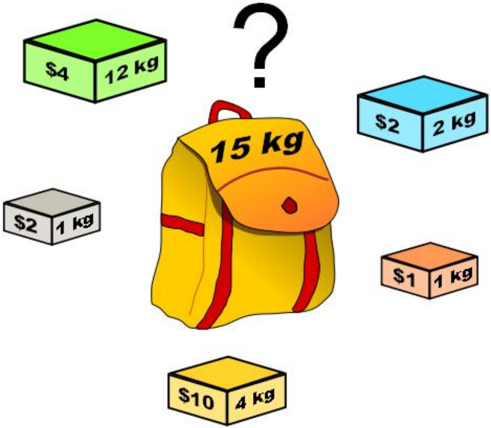
Solving Problems Using Dynamic Programming Approach

1. What is a subproblem?
   Can you define it clearly?
2. What is the relation between a smaller-size subproblem and a larger-size subproblem?
   Can we get the solution of the larger one from the smaller one?
   What is the dependency between them?
   What is the "DAG"?
   Is there a relationship between the optimality of a smaller subproblem and a larger subproblem?
3. How to solve this problem?
   What is the running-time complexity?

Knapsack

Knapsack Problem

Given n objects, each object i has weight w_i and value v_i, and a knapsack of capacity W, find most valuable items that fit into the knapsack



http://en.wikipedia.org/wiki/Knapsack_problem

Knapsack Problem

1. What will you do if all items are splittable?
2. What will you do if some items are splittable?
3. What will you do if all items are non-splittable?
4. What is the subproblem — the "node" in a "DAG"?
5. Can we always have a polynomial-time algorithm when we use Dynamic Programming approach?

Knapsack Problem

Subproblem:

$K(w, j) =$ maximum value achievable using a knapsack of capacity w and items $1, 2, \ldots, j$

Goal: $K(W, n)$

```
function knap-sack(W, S)

    Initialize all K(0, j) = 0 and all K(w, 0) = 0;

    for j = 1 to n
        for w = 1 to W
            if (w_j > w)
                K(w, j) = K(w, j - 1);
            else
                K(w, j) = max{K(w, j - 1), K(w - w_j, j - 1) + v_j};

    return K(W, n);
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

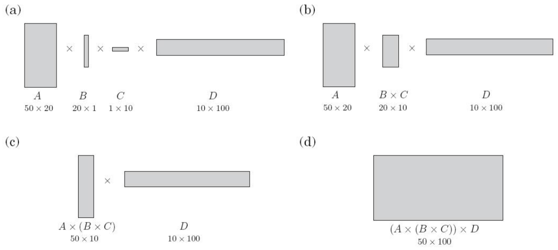
from Wayne's slides on "Algorithm Design"

Chain Matrix Multiplication

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications, to a good enough approximation. Using this formula, let's compare several different ways of evaluating $A \times B \times C \times D$:

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

Figure 6.6 $A \times B \times C \times D = (A \times (B \times C)) \times D$.



Chain Matrix Multiplication

$$
\begin{aligned}
C(i,j) &= \text{minimum cost of multiplying } A_i \times A_{i+1} \times \cdots \times A_j \\
C(i,j) &= \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}
\end{aligned}
$$

We are ready to code! In the following, the variable s denotes subproblem size.

```
for i = 1 to n:  C(i, i) = 0
for s = 1 to n-1:
  for i = 1 to n-s:
    j = i + s
    C(i, j) = min{C(i, k) + C(k+1, j) + m_{i-1} · m_k · m_j : i ≤ k < j}
return C(1, n)
```

The subproblems constitute a two-dimensional table, each of whose entries takes $O(n)$ time to compute. The overall running time is thus $O(n^3)$.

Traveling Salesman Problems

Definition

(TSP). Start from his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returned home. Given the pairwise distance between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Figure:

http://watching-movies-online.blogspot.com/2008/09/watch-mr-beans-holiday-hindi-dubbed.html

Traveling Salesman Problems

Definition

(TSP). Start from his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returned home. Given the pairwise distance between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Subproblem.
Let $C(S,j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.
Relation.

$$C(S,j) = \min_{i \in S, i \neq j} C(S - \{j\}, i) + d_{ij}.$$

```
C({1}, 1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1, 2, ..., n} of size s and containing 1:
        C(S, 1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S, j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1, ..., n}, j) + d_j1
```

There are at most $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 2^n)$.

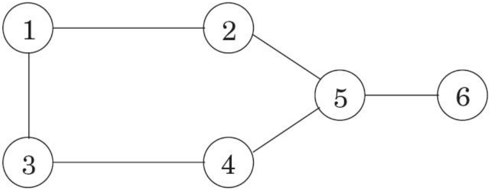
Independent Sets

Definition

A subset of nodes $S \subset V$ is an **independent set** of graph $G = (V, E)$ if there are no edges between them

Goal: Find the largest independent set
Known: This problem is believed to be intractable

The largest independent set in this graph has size 3

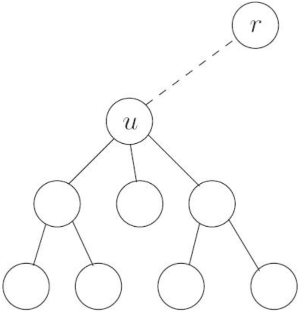


Independent Sets in Trees

$$I(u) = \text{ size of largest independent set of subtree hanging from } u$$

$$I(u) = \max\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w)\}$$

Figure 6.11 $I(u)$ is the size of the largest independent set of the subtree rooted at u. Two cases: either u is in this independent set, or it isn't.



Independent Sets in Trees

Theorem

The running time of using Dynamic Programming to find Independent Sets in Trees is $O(|V| + |E|)$

Proof.

The number of subproblems is exactly the number of vertices ? $\square$