

Chapter 9

More Dynamic Programming

CS 473: Fundamental Algorithms, Fall 2011

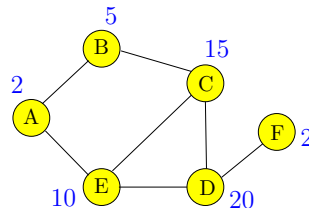
February 17, 2011

9.1 Maximum Weighted Independent Set in Trees

9.1.0.1 Maximum Weight Independent Set Problem

Input Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

Goal Find maximum weight independent set in G



Maximum weight independent set in above graph: $\{B, D\}$

9.1.0.2 Maximum Weight Independent Set in a Tree

Input Tree $T = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$

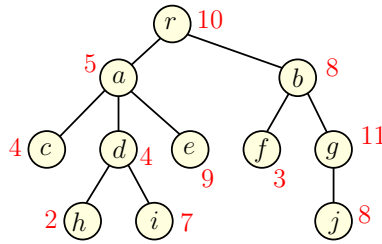
Goal Find maximum weight independent set in T

Maximum weight independent set in above tree: ??

9.1.0.3 Towards a Recursive Solution

For an arbitrary graph G :

- Number vertices as v_1, v_2, \dots, v_n
- Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).



(C) Saw that if graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for v_n is root r of T ?

9.1.0.4 Towards a Recursive Solution

Natural candidate for v_n is root r of T ? Let \mathcal{O} be an optimum solution to the whole problem.

Case $r \notin \mathcal{O}$: Then \mathcal{O} contains an optimum solution for each subtree of T hanging at a child of r .

Case $r \in \mathcal{O}$: None of the children of r can be in \mathcal{O} . $\mathcal{O} - \{r\}$ contains an optimum solution for each subtree of T hanging at a grandchild of r .

Subproblems? Subtrees of T hanging at nodes in T .

9.1.0.5 A Recursive Solution

$T(u)$: subtree of T hanging at node u

$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$$

9.1.0.6 Iterative Algorithm

- (A) Compute $OPT(u)$ bottom up. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u
- (B) What is an ordering of nodes of a tree T to achieve above? Post-order traversal of a tree.

9.1.0.7 Iterative Algorithm

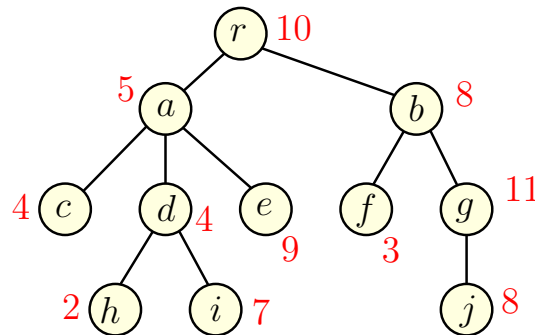
MIS-Tree(T):
Let v_1, v_2, \dots, v_n be a post-order traversal of nodes of T
for $i = 1$ **to** n **do**
 $M[v_i] = \max\left(\sum_{v_j \text{ child of } v_i} M[v_j], w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j]\right)$
return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

- (A) Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
- (B) Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

9.1.0.8 Example



9.2 DAGs and Dynamic Programming

9.2.0.9 Recursion and DAGs

Observation 9.2.1 Let A be a recursive algorithm for problem Π . For each instance I of Π there is an associated DAG $G(I)$.

- (A) Create directed graph $G(I)$ as follows...
- (B) For each sub-problem in the execution of A on I create a node.
- (C) If sub-problem v depends on or recursively calls sub-problem u add directed edge (u, v) to graph.
- (D) $G(I)$ is a DAG. Why? If $G(I)$ has a cycle then A will not terminate on I .

9.2.1 Iterative Algorithm for...

9.2.1.1 Dynamic Programming and DAGs

Observation 9.2.2 An iterative algorithm B obtained from a recursive algorithm A for a problem Π does the following:

For each instance I of Π , it computes a topological sort of $G(I)$ and evaluates sub-problems according to the topological ordering.

- (A) Sometimes the **DAG** $G(I)$ can be obtained directly without thinking about the recursive algorithm A
- (B) In some cases (*not all*) the computation of an optimal solution reduces to a shortest/longest path in **DAG** $G(I)$
- (C) Topological sort based shortest/longest path computation is dynamic programming!

9.2.2 A quick reminder...

9.2.2.1 A Recursive Algorithm for weighted interval scheduling

Let O_i be value of an optimal schedule for the first i jobs.

```

Schedule( $n$ ):
  if  $n = 0$  then return 0
  if  $n = 1$  then return  $w(v_1)$ 
   $O_{p(n)} \leftarrow$  Schedule( $p(n)$ )
   $O_{n-1} \leftarrow$  Schedule( $n - 1$ )
  if ( $O_{p(n)} + w(v_n) < O_{n-1}$ ) then
     $O_n = O_{n-1}$ 
  else
     $O_n = O_{p(n)} + w(v_n)$ 
  return  $O_n$ 

```

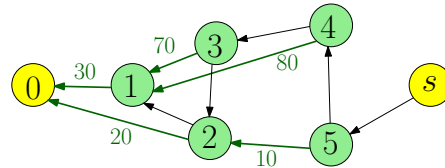
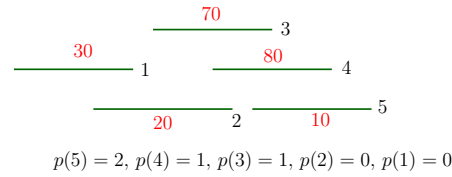
9.2.3 Weighted Interval Scheduling via...

9.2.3.1 Longest Path in a DAG

Given intervals, create a **DAG** as follows:

- (A) Create one node for each interval, plus a dummy sink node 0 for interval 0, plus a dummy source node s .
- (B) For each interval i add edge $(i, p(i))$ of the length/weight of v_i .
- (C) Add an edge from s to n of length 0.
- (D) For each interval i add edge $(i, i - 1)$ of length 0.

9.2.3.2 Example



9.2.3.3 Relating Optimum Solution

Given interval problem instance I let $G(I)$ denote the **DAG** constructed as described.

Claim 9.2.3 *Optimum solution to weighted interval scheduling instance I is given by longest path from s to 0 in $G(I)$.*

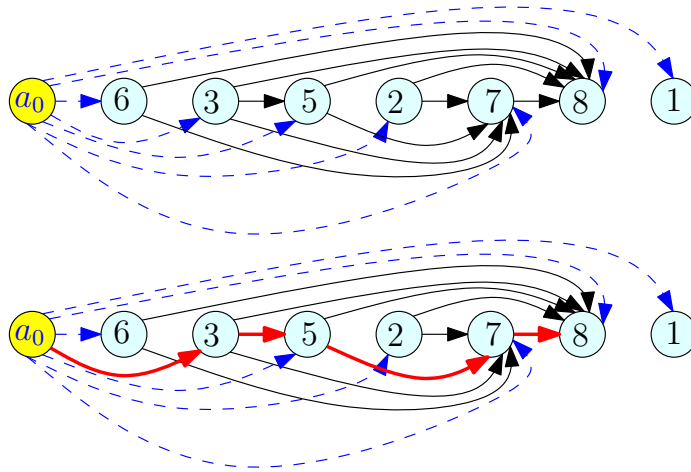
Assuming claim is true,

- (A) If I has n intervals, **DAG** $G(I)$ has $n + 2$ nodes and $O(n)$ edges. Creating $G(I)$ takes $O(n \log n)$ time: to find $p(i)$ for each i . How?
- (B) Longest path can be computed in $O(n)$ time — recall $O(m + n)$ algorithm for shortest/longest paths in **DAGs**.

9.2.3.4 DAG for Longest Increasing Sequence

Given sequence a_1, a_2, \dots, a_n create **DAG** as follows:

- (A) add sentinel a_0 to sequence where a_0 is less than smallest element in sequence
- (B) for each i there is a node v_i
- (C) if $i < j$ and $a_i < a_j$ add an edge (v_i, v_j)
- (D) find longest path from v_0



9.3 Edit Distance and Sequence Alignment

9.3.0.5 Spell Checking Problem

Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a *nearby* string?

What does nearness mean?

Question: Given two strings $x_1x_2 \dots x_n$ and $y_1y_2 \dots y_m$ what is a *distance* between them?

Edit Distance: minimum number of “edits” to transform x into y .

9.3.0.6 Edit Distance

Definition 9.3.1 *Edit distance between two words X and Y is the number of letter insertions, letter deletions and letter substitutions required to obtain Y from X .*

Example 9.3.2 *The edit distance between FOOD and MONEY is at most 4:*

$$\underline{F}OOD \rightarrow MO\underline{O}D \rightarrow MON\underline{O}D \rightarrow MONED\underline{D} \rightarrow MONEY$$

9.3.0.7 Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{cccc} F & O & O & D \\ M & O & N & E & Y \end{array}$$

Formally, an *alignment* is a set M of pairs (i, j) such that each index appears at most once, and there is no “crossing”: $i < i'$ and i is matched to j implies i' is matched to $j' > j$. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

9.3.0.8 Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

9.3.0.9 Applications

- (A) Spell-checkers and Dictionaries
- (B) Unix `diff`
- (C) DNA sequence alignment ... but, we need a new metric

9.3.0.10 Similarity Metric

Definition 9.3.3 For two strings X and Y , the cost of alignment M is

- (A) [Gap penalty] For each gap in the alignment, we incur a cost δ .
- (B) [Mismatch cost] For each pair p and q that have been matched in M , we incur cost α_{pq} ; typically $\alpha_{pp} = 0$.

Edit distance is special case when $\delta = \alpha_{pq} = 1$.

9.3.0.11 An Example

Example 9.3.4

$$\begin{array}{cccccccc} o & | & c & | & u & | & r & | & r & | & a & | & n & | & c & | & e \\ o & | & c & | & c & | & u & | & r & | & r & | & e & | & n & | & c & | & e \end{array} \quad \text{Cost} = \delta + \alpha_{ae}$$

Alternative:

$$\begin{array}{cccccccc} o & | & c & | & u & | & r & | & r & | & a & | & n & | & c & | & e \\ o & | & c & | & c & | & u & | & r & | & r & | & e & | & n & | & c & | & e \end{array} \quad \text{Cost} = 3\delta$$

Or a really stupid solution (delete string, insert other string):

$$\begin{array}{cccccccc} o & | & c & | & u & | & r & | & r & | & a & | & n & | & c & | & e & | & o & | & c & | & c & | & u & | & r & | & r & | & e & | & n & | & c & | & e \end{array}$$

Cost = 19 δ .

9.3.0.12 Sequence Alignment

Input Given two words X and Y , and gap penalty δ and mismatch costs α_{pq}

Goal Find alignment of minimum cost

9.3.1 Edit distance

9.3.1.1 Basic observation

Let $X = \alpha x$ and $Y = \beta y$

α, β : stings.

x and y single characters.

Think about optimal edit distance between X and Y as alignment, and consider last column of alignment of the two strings:

α	x	or	α	x	or	αx	
β	y		βy			β	y

Observation 9.3.5 *Prefixes must have optimal alignment!*

9.3.1.2 Problem Structure

Observation 9.3.6 *Let $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$. If (m, n) are not matched then either the m th position of X remains unmatched or the n th position of Y remains unmatched.*

(A) *Case x_m and y_n are matched.*

(A) Pay mismatch cost $\alpha_{x_my_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$

(B) *Case x_m is unmatched.*

(A) Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$

(C) *Case y_n is unmatched.*

(A) Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$

9.3.1.3 Subproblems and Recurrence

Optimal Costs

Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\text{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \text{Opt}(i-1, j-1), \\ \delta + \text{Opt}(i-1, j), \\ \delta + \text{Opt}(i, j-1) \end{cases}$$

Base Cases: $\text{Opt}(i, 0) = \delta \cdot i$ and $\text{Opt}(0, j) = \delta \cdot j$

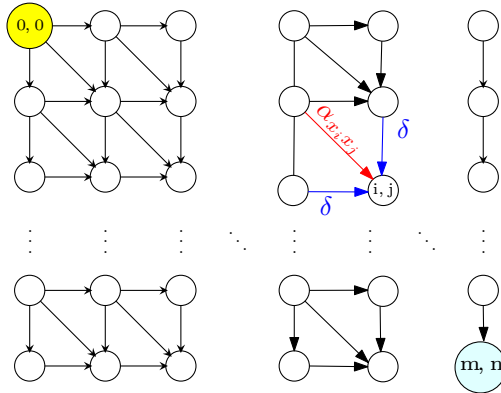


Figure 9.1: Iterative algorithm in previous slide computes values in row order. Optimal value is a shortest path from $(0, 0)$ to (m, n) in DAG.

9.3.1.4 Dynamic Programming Solution

```

for all  $i$  do  $M[i, 0] = i\delta$ 
for all  $j$  do  $M[0, j] = j\delta$ 

for  $i = 1$  to  $m$  do
  for  $j = 1$  to  $n$  do
     $M[i, j] = \min \begin{cases} \alpha_{x_i y_j} + M[i - 1, j - 1], \\ \delta + M[i - 1, j], \\ \delta + M[i, j - 1] \end{cases}$ 

```

Analysis

- (A) Running time is $O(mn)$.
- (B) Space used is $O(mn)$.

9.3.1.5 Matrix and DAG of Computation

9.3.1.6 Sequence Alignment in Practice

- (A) Typically the DNA sequences that are aligned are about 10^5 letters long!
- (B) So about 10^{10} operations and 10^{10} bytes needed
- (C) The killer is the 10GB storage
- (D) Can we reduce space requirements?

9.3.1.7 Optimizing Space

- (A) Recall

$$M(i, j) = \min \begin{cases} \alpha_{x_i y_j} + M(i - 1, j - 1), \\ \delta + M(i - 1, j), \\ \delta + M(i, j - 1) \end{cases}$$

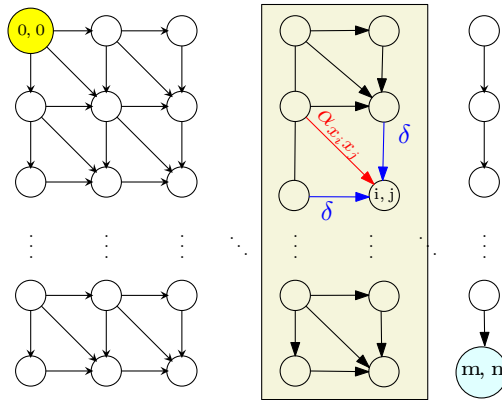


Figure 9.2: $M(i, j)$ only depends on previous column values. Keep only two columns and compute in column order.

- (B) Entries in j th column only depend on $(j - 1)$ st column and earlier entries in j th column
- (C) Only store the current column and the previous column reusing space; $N(i, 0)$ stores $M(i, j - 1)$ and $N(i, 1)$ stores $M(i, j)$

9.3.1.8 Computing in column order to save space

9.3.1.9 Space Efficient Algorithm

```

for all  $i$  do  $N[i, 0] = i\delta$ 
for  $j = 1$  to  $n$  do
   $N[0, j] = j\delta$  (* corresponds to  $M(0, j)$  *)
  for  $i = 1$  to  $m$  do
    
$$N[i, j] = \min \begin{cases} \alpha_{x_i, y_j} + N[i - 1, j] \\ \delta + N[i - 1, j] \\ \delta + N[i, j - 1] \end{cases}$$

  for  $i = 1$  to  $m$  do
    Copy  $N[i, j - 1] = N[i, j]$ 

```

Analysis

Running time is $O(mn)$ and space used is $O(2m) = O(m)$

9.3.1.10 Analyzing Space Efficiency

- (A) From the $m \times n$ matrix M we can construct the actual alignment (exercise)
- (B) Matrix N computes cost of optimal alignment but no way to construct the actual alignment
- (C) Space efficient computation of alignment? More complicated algorithm — see text book.

9.3.1.11 Takeaway Points

- (A) Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.

- (B) Given a recursive algorithm there is a natural **DAG** associated with the subproblems that are generated for given instance; this is the dependency graph. An iterative algorithm simply evaluates the subproblems in some topological sort of this **DAG**.
- (C) The space required to evaluate the answer can be reduced in some cases by a careful examination of that dependency **DAG** of the subproblems and keeping only a subset of the **DAG** at any time.