OLD CS 473: Fundamental Algorithms, Spring 2015

More Dynamic Programming

Lecture 10

February 19, 2015

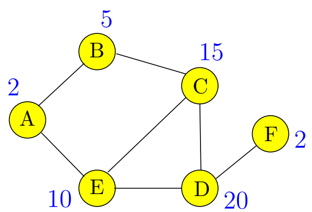
Part I

Maximum Weighted Independent Set in Trees

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

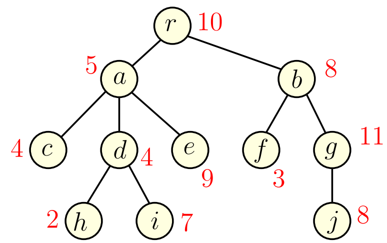


Maximum weight independent set in above graph: $\{B, D\}$

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: ??

Towards a Recursive Solution

1. For an arbitrary graph G:
    1. Number vertices as $v_1, v_2, \ldots, v_n$
    2. 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).
    3. Saw that if graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.
2. What about a tree?
3. Natural candidate for v_n is root r of T?

Towards a Recursive Solution

1. Natural candidate for v_n is root r of T?
2. 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.
3. Subproblems?
4. Subtrees of T hanging at nodes in T.

A Recursive Solution

1. $T(u)$: subtree of T hanging at node u.
2. $OPT(u)$: max weighted independent set value in $T(u)$.
3. $OPT(u) = \max \begin{cases} \sum_{v \text{ child of } u} OPT(v), \\ w(u) + \sum_{v \text{ grandchild of } u} OPT(v) \end{cases}$

Iterative Algorithm

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

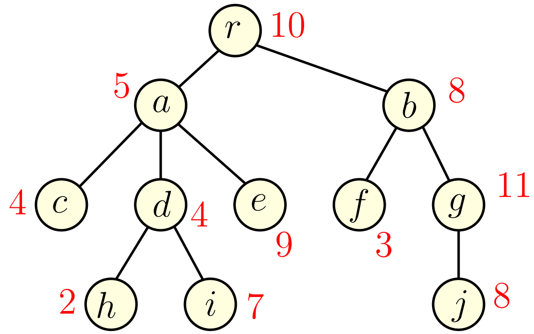
Iterative Algorithm

1. Code:

   MIS-Tree(T):

   Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
   for $i = 1$ to n **do**
   $$M[v_i] = \max\begin{pmatrix} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{pmatrix}$$
   return $M[v_n]$ (* Note: v_n is the root of T *)
2. Space: $O(n)$ to store the value at each node of T.
3. Running time:
   1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
   2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

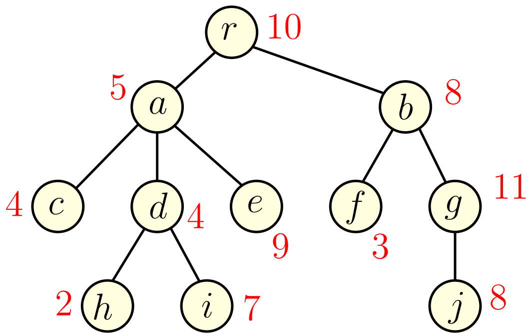
Example



Dominating set

Definition

$G = (V, E)$. The set $X \subseteq V$ is a **dominating set**, if any vertex $v \in V$ is either in X or is adjacent to a vertex in X.



Problem

Given weights on vertices, compute the **minimum** *weight dominating set in G.*

Dominating Set is **NP-Hard**!

Part II

DAGs and Dynamic Programming

Recursion and DAGs

Observation

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

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

Iterative Algorithm for...

Dynamic Programming and DAGs

Observation

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.*

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

A quick reminder...

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) <- Schedule(p(n))
    O_{n-1} <- 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
```

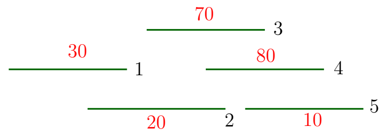

Weighted Interval Scheduling via...

Longest Path in a DAG

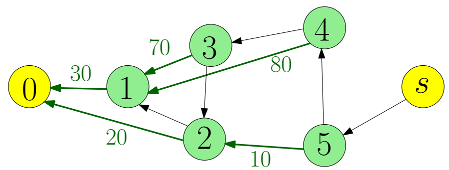
Given intervals, create a DAG as follows:

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

Example



$p(5) = 2, p(4) = 1, p(3) = 1, p(2) = 0, p(1) = 0$



Relating Optimum Solution

1. Given interval problem instance I let $G(I)$ denote the DAG constructed as described.
2. We have...

Claim

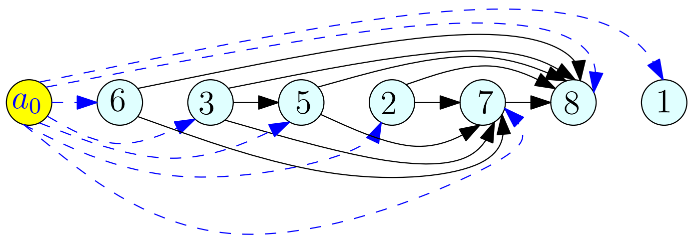
Optimum solution to weighted interval scheduling instance I is given by longest *path from s to 0 in $G(I)$.*

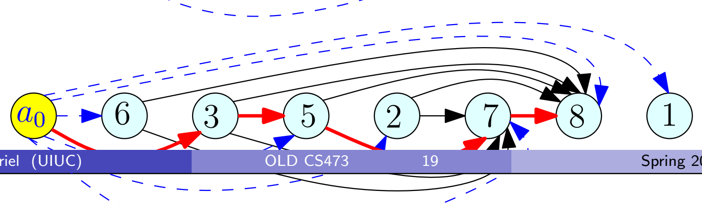
3. Assuming claim is true,
    1. If I has n intervals, DAG $G(I)$ has $n + 2$ nodes and $O(n)$ edges. Creating $G(I)$ takes $O(n \log n)$ time: to find $p(i)$ for each i. How?
    2. Longest path can be computed in $O(n)$ time — recall $O(m + n)$ algorithm for shortest/longest paths in DAGs.

DAG for Longest Increasing Sequence

Given sequence $a_1, a_2, \ldots, a_n$ create DAG as follows:

1. add sentinel a_0 to sequence where a_0 is less than smallest element in sequence
2. for each i there is a node v_i
3. if $i < j$ and $a_i < a_j$ add an edge (v_i, v_j)
4. find longest path from v_0





Part III

Edit Distance and Sequence Alignment

Spell Checking Problem

1. Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?
2. What does nearness mean?
3. Question: Given two strings $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a *distance* between them?
4. Edit Distance: minimum number of "edits" to transform x into y.

Edit Distance

Definition

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

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

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

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.

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

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.

Edit Distance Problem

Problem

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

Applications

1. Spell-checkers and Dictionaries
2. Unix `diff`
3. DNA sequence alignment ... but, we need a new metric

Similarity Metric

Definition

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

1. [Gap penalty] For each gap in the alignment, we incur a cost δ.
2. [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 $\delta = \alpha_{pq} = 1$.

An Example

Example

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

Alternative:

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

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

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

Cost $= 19\delta$.

Sequence Alignment

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

Goal Find alignment of minimum cost

Edit distance

Basic observation

1. Let $X = \alpha x$ and $Y = \beta y$.
2. α, β: strings. x and y single characters.
3. Optimal edit distance between X and Y as alignment. Consider last column of alignment of the two strings:

α	x
β	y

or

α	x
βy	

or

αx	
β	y

4. **Observation**
   Prefixes must have optimal alignment!

Problem Structure

Observation

Let $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$. If (m, n) are not matched then either the mth position of X remains unmatched or the nth position of Y remains unmatched.

1. Case x_m and y_n are matched.
   1. Pay mismatch cost $\alpha_{x_m y_n}$ plus cost of aligning strings $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_{n-1}$
2. Case x_m is unmatched.
   1. Pay gap penalty plus cost of aligning $x_1 \cdots x_{m-1}$ and $y_1 \cdots y_n$
3. Case y_n is unmatched.
   1. Pay gap penalty plus cost of aligning $x_1 \cdots x_m$ and $y_1 \cdots y_{n-1}$

Subproblems and Recurrence

Optimal Costs

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

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

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

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

1. Running time is $O(mn)$.
2. Space used is $O(mn)$.

Matrix and DAG of Computation

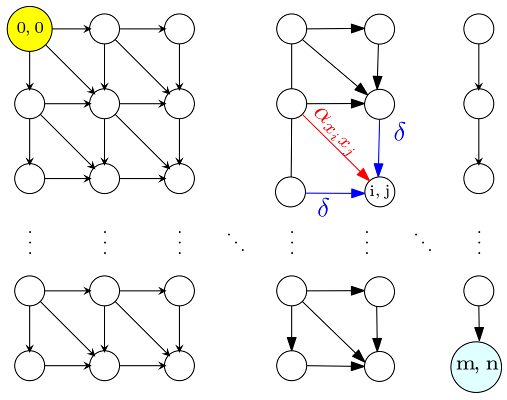


Figure: Iterative algorithm in previous slide computes values in row order. Optimal value is a shortest path from $(0, 0)$ to (m, n) in DAG.

Sequence Alignment in Practice

1. Typically the DNA sequences that are aligned are about 10^5 letters long!
2. So about 10^{10} operations and 10^{10} bytes needed
3. The killer is the 10GB storage
4. Can we reduce space requirements?

Optimizing Space

1. Recall

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

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

Computing in column order to save space

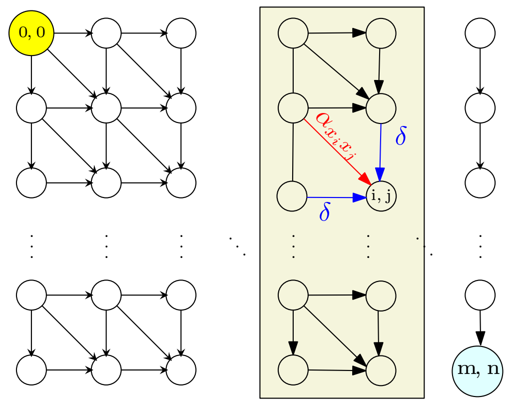


Figure: $M(i, j)$ only depends on previous column values. Keep only two columns and compute in column order.

Space Efficient Algorithm

```
for all i do N[i,0] = iδ
for j = 1 to n do
    N[0,1] = jδ (* corresponds to M(0,j) *)
    for i = 1 to m do
```

$$N[i,1] = \min \begin{cases} \alpha_{x_i y_j} + N[i-1,0] \\ \delta + N[i-1,1] \\ \delta + N[i,0] \end{cases}$$

```
    for i = 1 to m do
        Copy N[i,0] = N[i,1]
```

Analysis

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

Analyzing Space Efficiency

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

Takeaway Points

1. Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.
2. 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.
3. 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.