More DP: Edit Distance and Independent Sets in Trees

Lecture 15

March 10, 2020

Warm-up

Definition

A string is a palindrome if $w = w^R$.
Examples: I, $RACECAR$, $MALAYALAM$, $DOOFFOOD$

Warm-up

Definition

A string is a palindrome if $w = w^R$.
Examples: I, $RACECAR$, $MALAYALAM$, $DOOFFOOD$

Problem: Given a string w find the *longest subsequence* of w that is a palindrome.

Example

$MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM$ has $MHYMRORMYHM$ as a palindromic subsequence

Exercise

Assume $\boldsymbol{w}$ is stored in an array $\boldsymbol{A[1..n]}$

$\boldsymbol{LPS(i,j)}$: length of longest palindromic subsequence of $\boldsymbol{A[i..j]}$.

Recursive expression/code?

Part I

Edit Distance and Sequence Alignment

Spell Checking Problem

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

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 $x_1 x_2 \ldots x_m$ and $y_1 y_2 \ldots y_n$ what is a *distance* between them?

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 $x_1 x_2 \ldots x_m$ and $y_1 y_2 \ldots y_n$ what is a *distance* between them?

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

Edit Distance

Definition

Edit distance between 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**:

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

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.

F	**O**	**O**		**D**	A
M	**O**	**N**	**E**	**Y**	

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.

x_1	x_2	x_3		x_4
F	**O**	**O**		**D**
M	**O**	**N**	**E**	**Y**
y_1	y_2	y_3	y_4	y_5

Formally, an alignment is a set M of pairs (i, j) (x_i aligned with y_j) such that

- each index appears at most once, and
- there is no crossing: if $(i, j), (i', j') \in M$ and $i < i'$ then $j < j'$.

In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$.

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.

F	O	O		D
M	O	N	E	Y

Formally, an alignment is a set M of pairs (i, j) (x_i aligned with y_j) such that

- each index appears at most once, and
- there is no crossing: if $(i, j), (i', j') \in M$ and $i < i'$ then $j < j'$.

In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}$.

Cost of an alignment is:

$\#$ mismatched columns $+$ $\#$ unmatched indices in both strings.

More Examples

$X = \mathrm{GOT}$, $Y = \mathsf{GOAT}$

$X = \mathrm{ABCD}$, $Y = \mathsf{EFGH}$

$X = \mathrm{ABCD}$, $Y = \mathsf{EBDH}$

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 $\alpha_{pp} = 0$.

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

o		*c*	*u*	*r*	*r*	*a*	*n*	*c*	*e*
o	*c*	*c*	*u*	*r*	*r*	*e*	*n*	*c*	*e*

Cost $= \delta + \alpha_{ae}$

Alternative:

o		*c*	*u*	*r*	*r*		*a*	*n*	*c*	*e*
o	*c*	*c*	*u*	*r*	*r*	*e*		*n*	*c*	*e*

Cost $= 3\delta$

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

o	*c*	*u*	*r*	*r*	*a*	*n*	*c*	*e*										
									o	*c*	*c*	*u*	*r*	*r*	*e*	*n*	*c*	*e*

Cost $= 19\delta$.

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost **1** unit?

374

473

(A) 1
(B) 2
(C) 3
(D) 4
(E) 5

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost **1** unit?

373

473

(A) 1

(B) 2

(C) 3

(D) 4

(E) 5

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost **1** unit?

37
473

(A) 1
(B) 2
(C) 3
(D) 4
(E) 5

Sequence Alignment

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

Goal Find alignment of minimum cost

Sequence Alignment

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

Goal: Find alignment of minimum cost

Recall: An alignment is a set M of pairs (i, j) (i.e., x_i aligned with y_j) so that

- each index appears at most once, and
- there is no crossing: if $(i, j), (i', j') \in M$ and $i < i'$ then $j < j'$.

Question: $X = x_1 \ldots x_i \ldots x_m$ and
$Y = y_1 \ldots y_j \ldots y_n$. Can I have $(i, n), (m, j) \in M$? NO!

Sequence Alignment

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

Goal Find alignment of minimum cost

Recall: An alignment is a set M of pairs (i, j) (i.e., x_i aligned with y_j) so that

- each index appears at most once, and
- there is no crossing: if $(i, j), (i', j') \in M$ and $i < i'$ then $j < j'$.

Question: $X = x_1 \ldots x_i \ldots x_m$ and
$Y = y_1 \ldots y_j \ldots y_n$. Can I have $(i, n), (m, j) \in M$?

Then what are the options for x_m and y_n?

Let $X = \gamma x_m$ and $Y = \beta y_n$

γ, β: strings.

Consider last column of the optimal alignment of the two strings:

γ	x_m
β	y_n

or

γ	x_m
βy_n	

or

γx_m	
β	y_n

optimal $(m, n) \in M$

Observation

Prefixes must have optimal alignment!

$M' = M \setminus (m, n)$ $\quad M'$ is opt for (γ, β)

Problem Structure

Observation

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

$(x_1 .. x_i, y_1 .. y_j)$

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

$OPT(0, j) = j\delta \quad \forall j$

$OPT(i, 0) = i\delta \quad \forall i$

Problem Structure

Observation

Let $X = x_1x_2\cdots x_m$ and $Y = y_1y_2\cdots y_n$. If (x_m, y_n) are not matched then either the x_m remains unmatched or y_n 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 $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

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

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

OPT($x_1, \ldots x_i$, $y_1, \ldots y_j$)

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

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

Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$

$EDIST(A[1..m], B[1..n])$

```
    If (m = 0) return nδ
    If (n = 0) return mδ
```

Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$

$EDIST(A[1..m], B[1..n])$
    If $(m = 0)$ return $n\delta$
    If $(n = 0)$ return $m\delta$
    $m_1 = \alpha_{A[m],B[n]} + EDIST(A[1..(m-1)], B[1..(n-1)])$

Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$

$EDIST(A[1..m], B[1..n])$
 If $(m = 0)$ return $n\delta$
 If $(n = 0)$ return $m\delta$
 $m_1 = \alpha_{A[m],B[n]} + EDIST(A[1..(m-1)], B[1..(n-1)])$
 $m_2 = \delta + EDIST(A[1..(m-1)], B[1..n])$
 $m_3 = \delta + EDIST(A[1..m], B[1..(n-1)]))$

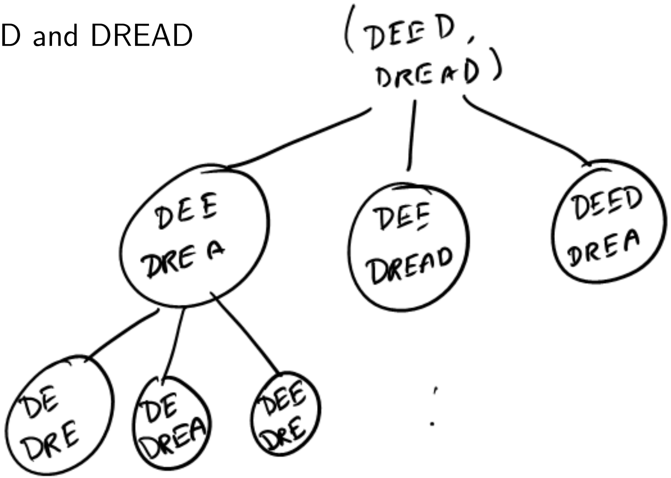
Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$

```
EDIST(A[1..m], B[1..n])
    If (m = 0) return nδ
    If (n = 0) return mδ
    m1 = α_{A[m],B[n]} + EDIST(A[1..(m − 1)], B[1..(n − 1)])
    m2 = δ + EDIST(A[1..(m − 1)], B[1..n])
    m3 = δ + EDIST(A[1..m], B[1..(n − 1)]))
    return min(m1, m2, m3)
```

Example

DEED and DREAD



Memoization

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 $\mathrm{Opt}(0,j) = \delta \cdot j$

Memoization

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 $\mathrm{Opt}(0,j) = \delta \cdot j$

Declare $M[0..m][0..n]$. $M[i,j]$ stores the value of $\mathrm{Opt}(i,j)$.

Memoization

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 $\text{Opt}(0,j) = \delta \cdot j$

Declare $M[0..m][0..n]$. $M[i,j]$ stores the value of $\text{Opt}(i,j)$.

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

Matrix and DAG of Computation

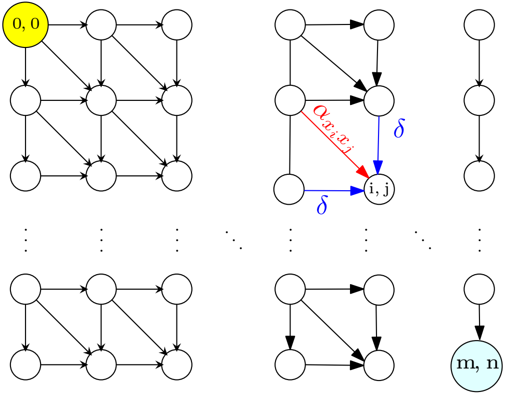


Figure: The iterative algorithm can compute values in row order.

Removing Recursion to obtain Iterative Algorithm

EDIST$(A[1..m], B[1..n])$
  int $M[0..m][0..n]$
  for $i = 1$ `to` m **do** $M[i, 0] = i\delta$
  for $j = 1$ `to` n **do** $M[0, j] = j\delta$

Removing Recursion to obtain Iterative Algorithm

```
EDIST(A[1..m], B[1..n])
    int  M[0..m][0..n]
    for i = 1 to m do M[i, 0] = iδ
    for j = 1 to n do M[0, j] = jδ

    for i = 1 to m do
        for j = 1 to n do
```

$$M[i,j] = \min \begin{cases} \alpha_{A[i],B[j]} + M[i-1,j-1], \\ \delta + M[i-1,j], \\ \delta + M[i,j-1] \end{cases}$$

Removing Recursion to obtain Iterative Algorithm

```
EDIST(A[1..m], B[1..n])
    int  M[0..m][0..n]
    for i = 1 to m do M[i, 0] = iδ
    for j = 1 to n do M[0, j] = jδ

    for i = 1 to m do
        for j = 1 to n do
```

$$M[i,j] = \min \begin{cases} \alpha_{A[i],B[j]} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

Running time is $O(mn)$.

Removing Recursion to obtain Iterative Algorithm

$EDIST(A[1..m], B[1..n])$
    $int \quad M[0..m][0..n]$
    for $i = 1$ `to` m **do** $M[i, 0] = i\delta$
    for $j = 1$ `to` n **do** $M[0, j] = j\delta$

    for $i = 1$ `to` m **do**
        for $j = 1$ `to` n **do**

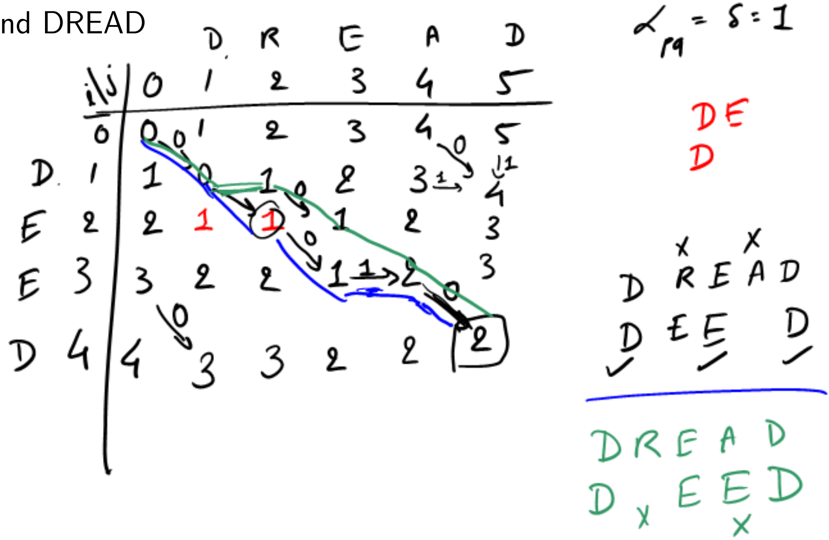
$$M[i,j] = \min \begin{cases} \alpha_{A[i],B[j]} + M[i-1,j-1], \\ \delta + M[i-1,j], \\ \delta + M[i,j-1] \end{cases}$$

Analysis

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

Example

DEED and DREAD



Sequence Alignment in Practice

1. Typically the DNA sequences that are aligned are about $\mathbf{10^5}$ letters long!
2. So about $\mathbf{10^{10}}$ operations and $\mathbf{10^{10}}$ 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

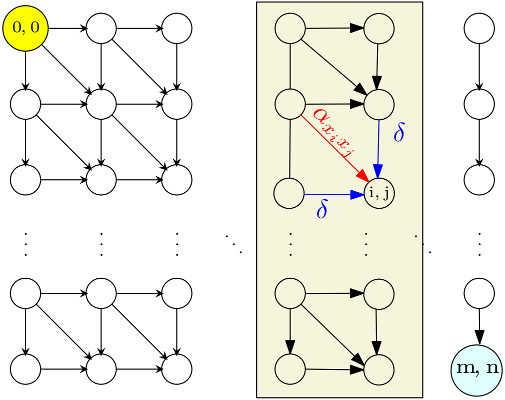


Figure: $\boldsymbol{M(i,j)}$ only depends on previous column values. Keep only two columns and compute in column order.

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 notes and Kleinberg-Tardos book.

Part II

Longest Common Subsequence Problem

LCS Problem

Definition

LCS between two strings X and Y is the length of longest common subsequence between X and Y.

Example

LCS between ABAZDC and BACBAD is

LCS Problem

Definition

LCS between two strings X and Y is the length of longest common subsequence between X and Y.

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

LCS Problem

Definition

LCS between two strings X and Y is the length of longest common subsequence between X and Y.

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.

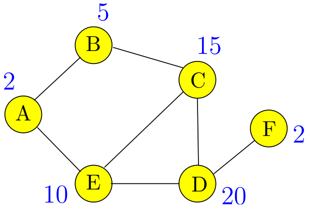
Part III

Maximum Weighted Independent Set in Trees

Maximum Weight Independent Set Problem

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

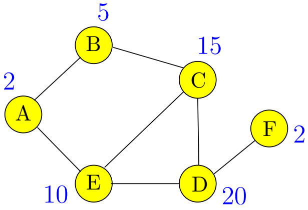
Goal Find maximum weight independent set in G



Maximum Weight Independent Set 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

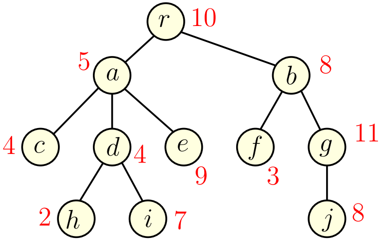


Maximum weight independent set in above graph: $\{B, D\}$

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

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. If graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.

Towards a Recursive Solution

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. If graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree?

Towards a Recursive Solution

For an arbitrary graph G:

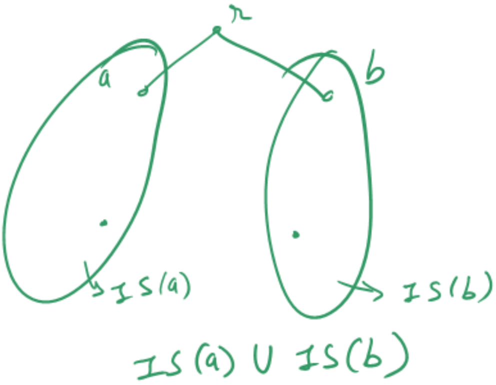
1. Number vertices as $v_1, v_2, \ldots, v_n$
2. Find recursively optimum solutions without v_n (recurse on $G - v_n$) and with v_n (recurse on $G - v_n - N(v_n)$ & include v_n).
3. If graph G is arbitrary there was no good ordering that resulted in a small number of subproblems.

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

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 $r \notin \mathcal{O}$:



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.

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.

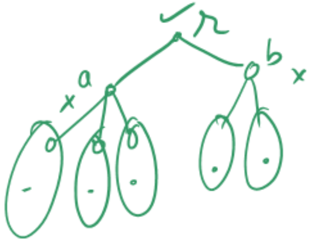
Case $r \in \mathcal{O}$: None of the children of r can be in $\mathcal{O}$.

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.



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 rooted at nodes in T.

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 rooted at nodes in T.

How many of them?

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.

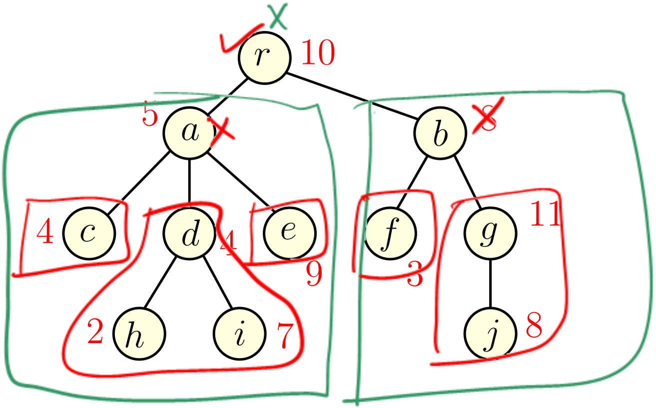
Case $r \notin \mathcal{O}$:Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

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 rooted at nodes in T.

How many of them? $O(n)$

Example



A Recursive Solution

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

$$OPT(u) =$$

A Recursive Solution

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

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

A Recursive Solution

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

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

Iterative Algorithm

1. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u.

A Recursive Solution

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

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

Iterative Algorithm

1. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u.

   Compute $OPT(u)$ bottom up.

2. What is an ordering of nodes of a tree T to achieve above?

A Recursive Solution

$T(u)$: subtree of T hanging at node u
$OPT(u)$: max weighted independent set value in $T(u)$

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

Iterative Algorithm

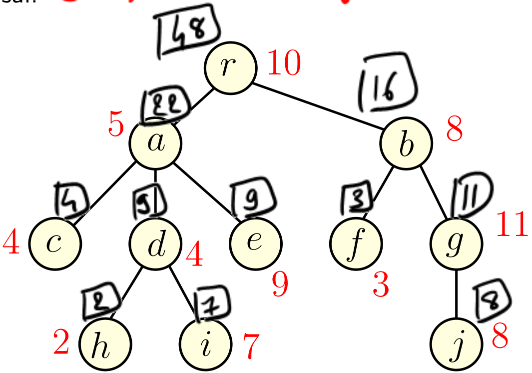
1. To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of u.

   Compute $OPT(u)$ bottom up.
2. What is an ordering of nodes of a tree T to achieve above? Post-order traversal of a tree.

Example

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

Post-order traversal: c h i d e a f j g b r



Iterative Algorithm

Declare $\boldsymbol{M[1..n]}$. $\boldsymbol{M[v]}$ stores the max weighted independent set value for tree $\boldsymbol{T(v)}$.

Iterative Algorithm

Declare $M[1..n]$. $M[v]$ stores the max weighted independent set value for tree $T(v)$.

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**

$$M[v_i] = \max \left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

  return $M[v_n]$ (* Note: v_n is the root of T *)

Iterative Algorithm

Declare $M[1..n]$. $M[v]$ stores the max weighted independent set value for tree $T(v)$.

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**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

    return $M[v_n]$ (* Note: v_n is the root of T *)

Space:

Iterative Algorithm

Declare $M[1..n]$. $M[v]$ stores the max weighted independent set value for tree $T(v)$.

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

$$M[v_i] = \max \begin{pmatrix} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{pmatrix}$$

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

Iterative Algorithm

Declare $M[1..n]$. $M[v]$ stores the max weighted independent set value for tree $T(v)$.

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**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

    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:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.

Iterative Algorithm

Declare $M[1..n]$. $M[v]$ stores the max weighted independent set value for tree $T(v)$.

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

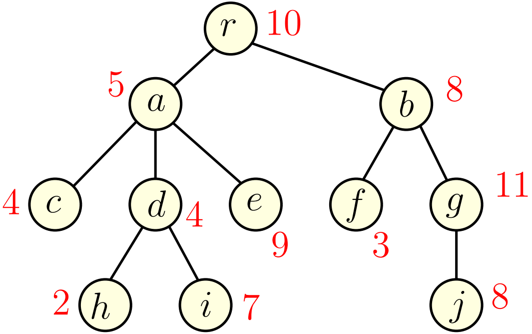
$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

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

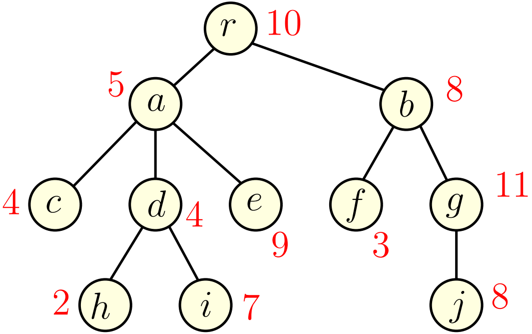
Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.

Example



Example



Better running time: A value $M[d]$ is accessed only by a (parent) and r (grand parent) $\Rightarrow O(n)$.

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.