Algorithms & Models of Computation

CS/ECE 374, Fall 2020

More Dynamic Programming

Lecture 14

Tuesday, October 13, 2020

LaTeXed: September 11, 2020 16:00

14.1
Review of dynamic programming and some new problems

What is the running time of the following?

Consider computing $f(x, y)$ by recursive function + memoization.

$$f(x, y) = \sum_{i=1}^{x+y-1} x * f(x + y - i, i - 1),$$
$$f(0, y) = y \qquad f(x, 0) = x.$$

The resulting algorithm when computing $f(n, n)$ would take:

- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- The function is ill defined - it can not be computed.

Recipe for Dynamic Programming

1. Develop a recursive backtracking style algorithm $\mathcal{A}$ for given problem.
2. Identify structure of subproblems generated by $\mathcal{A}$ on an instance I of size n
   1. Estimate number of different subproblems generated as a function of n. Is it polynomial or exponential in n?
   2. If the number of problems is "small" (polynomial) then they typically have some "clean" structure.
3. Rewrite subproblems in a compact fashion.
4. Rewrite recursive algorithm in terms of notation for subproblems.
5. Convert to iterative algorithm by bottom up evaluation in an appropriate order.
6. Optimize further with data structures and/or additional ideas.

14.1.1

Is in L^k?

A variation

Input A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function **IsStringinL**(*string* x) that decides whether x is in L, and non-negative integer k

Goal Decide if $w \in L^k$ using **IsStringinL**(*string* x) as a black box sub-routine

Example

Suppose L is ***English*** and we have a procedure to check whether a string/word is in the ***English*** dictionary.

- Is the string "isthisanenglishsentence" in $\textit{English}^5$?
- Is the string "isthisanenglishsentence" in $\textit{English}^4$?
- Is "asinineat" in $\textit{English}^2$?
- Is "asinineat" in $\textit{English}^4$?
- Is "zibzzzad" in $\textit{English}^1$?

Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L^{k-1}$ and $v \in L$

Assume w is stored in array $A[1..n]$

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO  // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L^{k-1}$ and $v \in L$

Assume w is stored in array $A[1..n]$

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO  // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

Recursive Solution

When is $w \in L^k$?

$k = 0$: $w \in L^k$ iff $w = \epsilon$

$k = 1$: $w \in L^k$ iff $w \in L$

$k > 1$: $w \in L^k$ if $w = uv$ with $u \in L^{k-1}$ and $v \in L$

Assume w is stored in array $A[1..n]$

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO   // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

Analysis

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO   // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

- How many distinct sub-problems are generated by **IsStringinLk**$(A[1..n], k)$? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2 k)$

Analysis

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO   // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

- How many distinct sub-problems are generated by IsStringinLk($A[1..n], k$)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2 k)$

Analysis

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO   // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

- How many distinct sub-problems are generated by IsStringinLk($A[1..n], k$)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2 k)$

Analysis

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO   // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

- How many distinct sub-problems are generated by IsStringinLk($A[1..n], k$)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2k)$

Analysis

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO   // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

- How many distinct sub-problems are generated by IsStringinLk($A[1..n], k$)? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2k)$

Analysis

```
IsStringinLk(A[1...i], k):
    if k = 0 and i = 0 then return YES
    if k = 0 then return NO  // i > 0
    if k = 1 then
        return IsStringinL(A[1...i])

    for ℓ = 1...i − 1 do
        if IsStringinLk(A[1...ℓ], k − 1) and IsStringinL(A[ℓ + 1...i]) then
            return YES

    return NO
```

- How many distinct sub-problems are generated by **IsStringinLk**$(A[1..n], k)$? $O(nk)$
- How much space? $O(nk)$
- Running time if we use memoization? $O(n^2 k)$

Another variant

Question: What if we want to check if $w \in L^i$ for some $0 \leq i \leq k$? That is, is $w \in \cup_{i=0}^{k} L^i$?

Exercise

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

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 w is stored in an array $A[1..n]$

$LPS(A[1..n])$: length of longest palindromic subsequence of A.

Recursive expression/code?

THE END

...

(for now)

14.2
Edit Distance and Sequence Alignment

14.2.1
Problem definition and background

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 $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a distance between them?

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

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 $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a distance between them?

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

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 $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$ what is a distance between them?

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:

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

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.

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

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: 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) 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: 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) 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 $\delta = \alpha_{pq} = 1$.

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 $\delta = \alpha_{pq} = 1$.

THE END

...

(for now)

14.2.2
Edit distance as alignment

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

1. 1
2. 2
3. 3
4. 4
5. 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

- 1
- 2
- 3
- 4
- 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

- 1
- 2
- 3
- 4
- 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 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?

THE END

…

(for now)

14.2.3
Edit distance: The algorithm

Edit distance

Basic observation

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

α, β: strings.

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
β	y

or

α	x
βy	

or

αx	
β	y

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

$x_1 \ldots x_{i-1}$	x_i
$y_1 \ldots y_{j-1}$	y_j

or

$x_1 \ldots x_{i-1}$	x
$y_1 \ldots y_{j-1} y_j$	

or

$x_1 \ldots x_{i-1} x_i$	
$y_1 \ldots y_{j-1}$	y_j

Optimal Costs

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

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

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

Subproblems and Recurrence

$x_1 \ldots x_{i-1}$	x_i
$y_1 \ldots y_{j-1}$	y_j

or

$x_1 \ldots x_{i-1}$	x
$y_1 \ldots y_{j-1} y_j$	

or

$x_1 \ldots x_{i-1} x_i$	
$y_1 \ldots y_{j-1}$	y_j

Optimal Costs

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

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

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

Recursive Algorithm

Assume X is stored in array $A[1..m]$ and Y is stored in $B[1..n]$
Array $COST$ stores cost of matching two chars. Thus $COST[a, b]$ give the cost of matching character a to character b.

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

Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε						
D						
E						
E						
D						

Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1					
E	2					
E	3					
D	3					

Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2					
E	3					
D	3					

Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3					
D	3					

Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3					

Example: DEED and DREAD

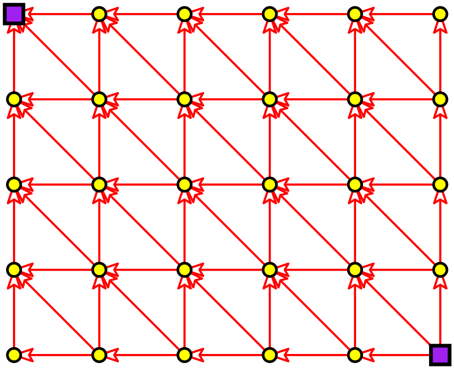
	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3	3	3	2	2	2

D	R	E	A	D
D	E	E		D

Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3	3	3	2	2	2

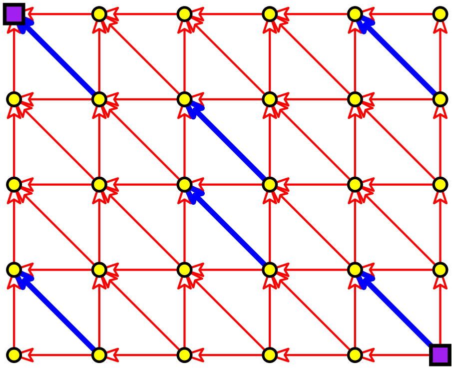
D	R	E	A	D
D	E	E		D



Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3	3	3	2	2	2

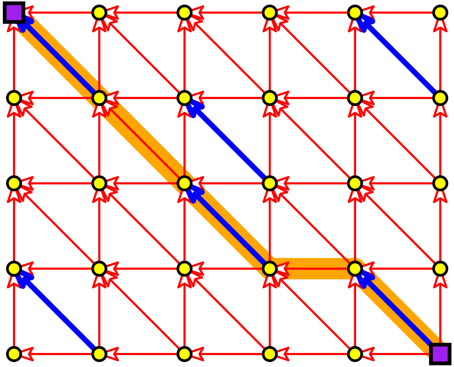
D	R	E	A	D
D	E	E		D



Example: DEED and DREAD

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3	3	3	2	2	2

D	R	E	A	D
D	E	E		D



THE END

...

(for now)

14.2.4

Dynamic programming algorithm for edit-distance

As part of the input...

The cost of aligning a character against another character

Σ: Alphabet

We are given a **cost** function (in a table):

$$\forall b, c \in \Sigma \qquad COST[b][c] = \text{ cost of aligning } b \text{ with } c.$$
$$\forall b \in \Sigma \qquad COST[b][b] = 0$$

δ : price of deletion of insertion of a single character

Memoizing the Recursive Algorithm (Explicit Memoization)

Input: Two strings

$A[1 \ldots m]$

$B[1 \ldots n]$

```
EditDistance(A, B)
    int  M[0..m][0..n]
    ∀i, j  M[i][j] ← ∞
    return edEMI(m, n)
```

```
edEMI( i , j )        // A[1...i], B[1...j]
    if M[i][j] < ∞
        return M[i][j]    // stored value

    if i = 0 or j = 0
        M[i][j] = (i + j)δ
        return M[i][j]

    m1 = δ + edEMI( i − 1 , j )
    m2 = δ + edEMI( i , j − 1)

    m3 = COST[A[i]][B[j]]
        +  edEMI( i − 1 , j − 1 )

    M[i][j] = min(m1, m2, m3)
    return M[i][j]
```

Dynamic program for edit distance

Removing Recursion to obtain Iterative Algorithm

$\boldsymbol{EDIST}(A[1..m], B[1..n])$

    $int\quad M[0..m][0..n]$
    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**

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

Analysis

1. Running time is $O(mn)$.

Dynamic program for edit distance

Removing Recursion to obtain Iterative Algorithm

$$
\begin{array}{l}
\mathbf{EDIST}(A[1..m], B[1..n]) \\
\quad \mathbf{int} \quad M[0..m][0..n] \\
\quad \textbf{for } i = 1 \texttt{ to } m \textbf{ do } M[i,0] = i\delta \\
\quad \textbf{for } j = 1 \texttt{ to } n \textbf{ do } M[0,j] = j\delta \\
\\
\quad \textbf{for } i = 1 \texttt{ to } m \textbf{ do} \\
\quad\quad \textbf{for } j = 1 \texttt{ to } n \textbf{ do} \\
\quad\quad\quad M[i][j] = \min \begin{cases} COST\left[A[i]\right]\left[B[j]\right] + M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}
\end{array}
$$

Analysis

1. Running time is $O(mn)$.

Dynamic program for edit distance

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**

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

Analysis

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

THE END

...

(for now)

14.2.5

Reducing space for edit distance

Matrix and DAG of computation of edit distance

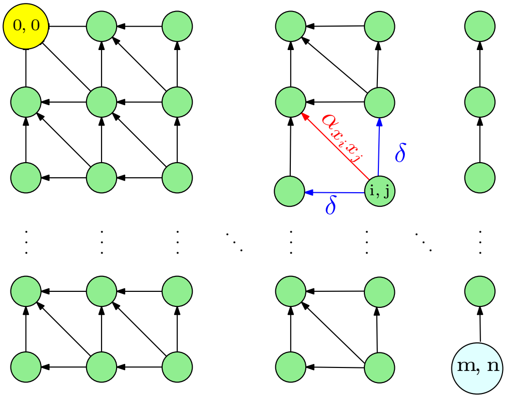


Figure: Iterative algorithm in previous slide computes values in row order.

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 $M(i,j)$

Example: DEED vs. BREAD filled by column

	ε	D	R	E	A	D
ε						
D						
E						
E						
D						

Example: DEED vs. BREAD filled by column

	ε	D	R	E	A	D
ε	0	1	2	3	4	5
D	1					
E	2					
E	3					
D	3					

Example: DEED vs. BREAD filled by column

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0				
E	2	1				
E	3	2				
D	3	3				

Example: DEED vs. BREAD filled by column

	ε	D	R	E	A	D
ε	0	1	2	3	4	5
D	1	0	1			
E	2	1	1			
E	3	2	2			
D	3	3	3			

Example: DEED vs. BREAD filled by column

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2		
E	2	1	1	1		
E	3	2	2	1		
D	3	3	3	2		

Example: DEED vs. BREAD filled by column

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	
E	2	1	1	1	2	
E	3	2	2	1	2	
D	3	3	3	2	2	

Example: DEED vs. BREAD filled by column

	ε	*D*	*R*	*E*	*A*	*D*
ε	0	1	2	3	4	5
D	1	0	1	2	3	4
E	2	1	1	1	2	3
E	3	2	2	1	2	3
D	3	3	3	2	2	2

Computing in column order to save space

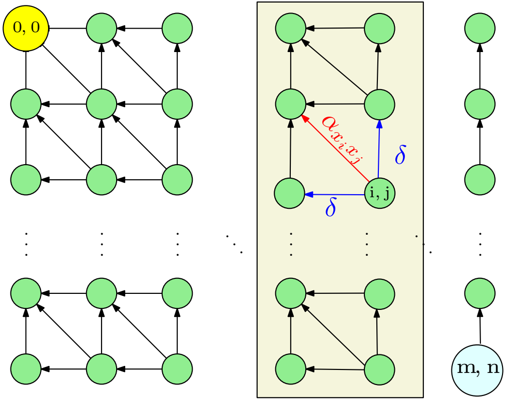


Figure: $\boldsymbol{M(i,j)}$ only depends on previous column values. Keep only two columns and compute in column order.

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.

THE END

...

(for now)

14.2.6
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.

$ABAZDC$
$BACBAD$

$ABAZDC$
$BACBAD$

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.

LCS Problem

Definition

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

$ABAZDC$
$BACBAD$

$ABAZDC$
$BACBAD$

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.

LCS Problem

Definition

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

$$\begin{array}{c} ABAZDC \\ BACBAD \end{array} \qquad \begin{array}{c} ABAZDC \\ BACBAD \end{array}$$

Example

LCS between ABAZDC and BACBAD is 4 via ABAD

Derive a dynamic programming algorithm for the problem.

LCS recursive definition

$A[1..n], B[1..m]$: Input strings.

$$LCS(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ \max\begin{pmatrix} LCS(i-1,j), \\ LCS(i,j-1) \end{pmatrix} & A[i] \neq B[j] \\ \max\begin{pmatrix} LCS(i-1,j), \\ LCS(i,j-1), \\ 1 + LCS(i-1,j-1) \end{pmatrix} & A[i] = B[j] \end{cases}$$

Similar to edit distance... $O(nm)$ time algorithm $O(m)$ space.

LCS recursive definition

$A[1..n], B[1..m]$: Input strings.

$$LCS(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ \max\begin{pmatrix} LCS(i-1,j), \\ LCS(i,j-1) \end{pmatrix} & A[i] \neq B[j] \\ \max\begin{pmatrix} LCS(i-1,j), \\ LCS(i,j-1), \\ 1 + LCS(i-1,j-1) \end{pmatrix} & A[i] = B[j] \end{cases}$$

Similar to edit distance... $O(nm)$ time algorithm $O(m)$ space.

Longest common subsequence is just edit distance for the two sequences...

A, B: input sequences
Σ: "alphabet" all the different values in A and B

$$\forall b, c \in \Sigma : b \neq c \qquad COST[b][c] = +\infty.$$
$$\forall b \in \Sigma \qquad COST[b][b] = 1$$

1 : price of deletion of insertion of a single character

Length of longest common subsequence $= m + n - \mathrm{ed}(A, B)$

Longest common subsequence is just edit distance for the two sequences...

$\boldsymbol{A}, \boldsymbol{B}$: input sequences
Σ: "alphabet" all the different values in $\boldsymbol{A}$ and $\boldsymbol{B}$

$$\forall b, c \in \Sigma : b \neq c \qquad COST[b][c] = +\infty.$$
$$\forall b \in \Sigma \qquad COST[b][b] = 1$$

$1:$ price of deletion of insertion of a single character

Length of longest common subsequence $= m + n - \mathrm{ed}(A, B)$

THE END

...

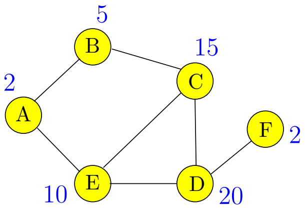
(for now)

14.3
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

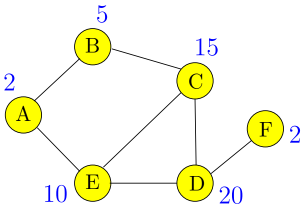


Maximum weight independent set in above graph: $\{B, D\}$

Maximum Weight Independent Set 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

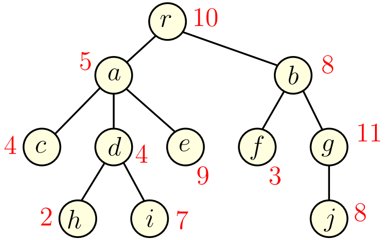


Maximum weight independent set in above graph: $\{B, D\}$

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. 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?

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. 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?

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. 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?

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? $O(n)$

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? $O(n)$

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? $O(n)$

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? $O(n)$

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.

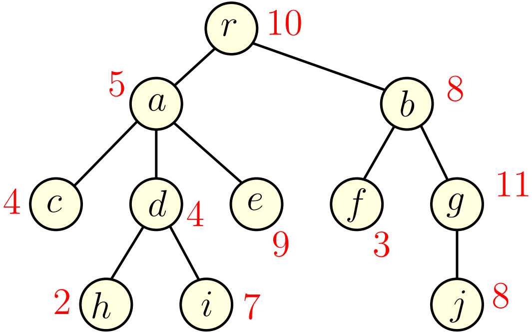
Case $r \notin \mathcal{O}$: Then $\mathcal{O}$ contains an optimum solution for each subtree of T hanging at a child of r.

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 $T(u)$

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

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. 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? Post-order traversal of a tree.

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? Post-order traversal of a tree.

Iterative Algorithm

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.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

```
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( Σ_{v_j child of v_i} M[v_j],
                      w(v_i) + Σ_{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.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

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 *)

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.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

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 *)

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.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

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**

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

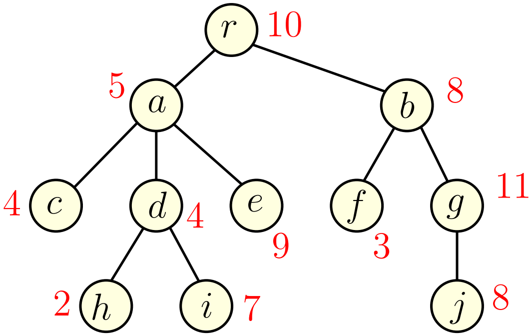
  return $M[v_n]$ (* Note: v_n is the root of T *)

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.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Example



THE END

…

(for now)

14.4
Dynamic programming and DAGs

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.