CSE 548: Algorithms

Dynamic Programming

R. Sekar

Overview

- Another approach for *optimization problems*, more general and versatile than greedy algorithms.
- *Optimal substructure* The optimal solution contains optimal solutions to subproblems.
- *Overlapping subproblems.* Typically, the same subproblems are solved repeatedly.
- Solve subproblems in *a certain order*, and *remember solutions* for later reuse.

Topics

DAGs and Dynamic Programming

- *Canonical way to represent dynamic programming*

  Nodes in the DAG represent subproblems

  Edges capture dependencies between subproblems

  Topological sorting solves subproblems in the right order

  Remember subproblem solutions to avoid recomputation

DAGs and Dynamic Programming

- *Canonical way to represent dynamic programming*

  Nodes in the DAG represent subproblems

  Edges capture dependencies between subproblems

  Topological sorting solves subproblems in the right order

  Remember subproblem solutions to avoid recomputation

- Many bottom-up computations on trees/dags *are* instances of dynamic programming

DAGs and Dynamic Programming

- *Canonical way to represent dynamic programming*

  Nodes in the DAG represent subproblems

  Edges capture dependencies between subproblems

  Topological sorting solves subproblems in the right order

  Remember subproblem solutions to avoid recomputation

- Many bottom-up computations on trees/dags *are* instances of dynamic programming
  - applies to trees of recursive calls (w/ duplication), e.g., Fib

DAGs and Dynamic Programming

- *Canonical way to represent dynamic programming*

  Nodes in the DAG represent subproblems

  Edges capture dependencies between subproblems

  Topological sorting solves subproblems in the right order

  Remember subproblem solutions to avoid recomputation

- Many bottom-up computations on trees/dags *are* instances of dynamic programming
  - applies to trees of recursive calls (w/ duplication), e.g., Fib
- For problems in other domains, DAGs are implicit, as is the topological sort.

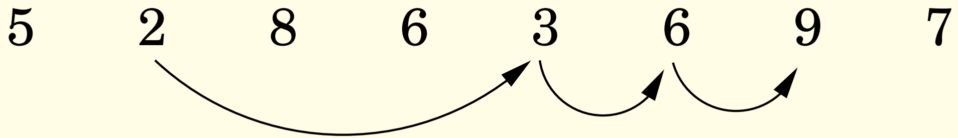
Longest Increasing Subsequence

Definition

Given a sequence $a_1, a_2, \ldots, a_n$, its LIS is a sequence

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

that maximizes k subject to $i_j < i_{j+1}$ and $a_{i_j} \leq a_{i_{j+1}}$.



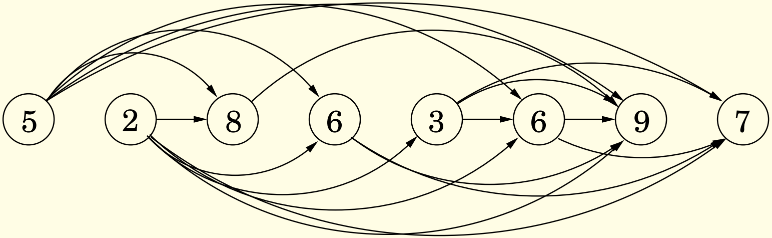
Casting LIS problem using a DAG

Nodes: represent elements in the sequence

Edges: connect an element to all followers that are larger

Topological sorting: sequence already topologically sorted

Remember: Using an array $L[1..n]$



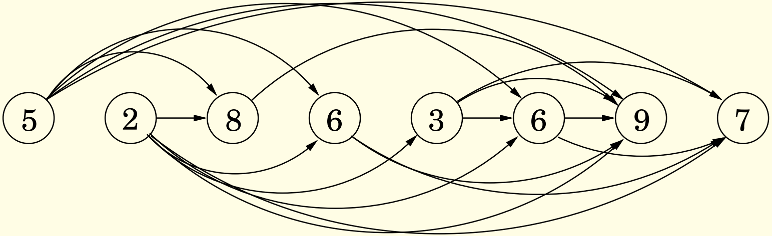
Algorithm for LIS

$LIS(E)$

for $j = 1$ **to** n **do**

$L[j] = 1 + max_{(i,j) \in E} L[i]$

return $max_{j=1}^{n} L[j]$



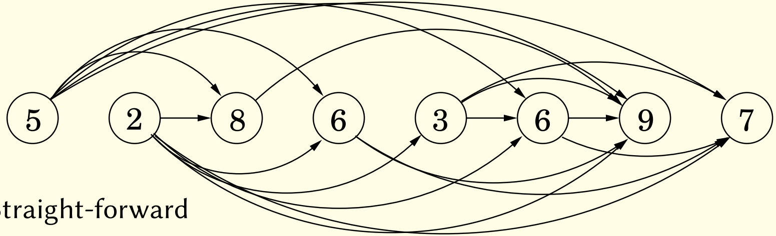
Algorithm for LIS

$LIS(E)$

for $j = 1$ **to** n **do**

$L[j] = 1 + max_{(i,j) \in E} L[i]$

return $max_{j=1}^{n} L[j]$



Correctness: Straight-forward

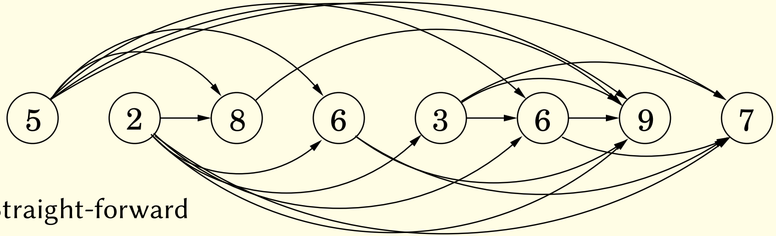
Algorithm for LIS

$LIS(E)$

for $j = 1$ **to** n **do**

$L[j] = 1 + max_{(i,j) \in E} L[i]$

return $max_{j=1}^{n} L[j]$



Correctness: Straight-forward

Complexity: What is it? Can it be improved?

Knapsack Problem (Recap)

- You have a choice of items you can pack in the sack
- Maximize value of sack, subject to a weight limit of W

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

Knapsack Problem (Recap)

- You have a choice of items you can pack in the sack
- Maximize value of sack, subject to a weight limit of W

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

Fractional knapsack: Fractional quantities acceptable

0-1 knapsack: Take all of one item or none at all

Knapsack w/ repetition: Take any integral number of items.

Knapsack Problem (Recap)

- You have a choice of items you can pack in the sack
- Maximize value of sack, subject to a weight limit of W

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

Fractional knapsack: Fractional quantities acceptable

0-1 knapsack: Take all of one item or none at all

Knapsack w/ repetition: Take any integral number of items.

No polynomial solution for the last two, but dynamic programming can solve them in *pseudo-polynomial* time of $O(nW)$.

Knapsack w/ repetition: Identify subproblems

- Consider subproblems by reducing the weight
  - Compute $K(W)$ in terms of $K(W')$ for $W' < W$

Knapsack w/ repetition: Identify subproblems

- Consider subproblems by reducing the weight
  - Compute $K(W)$ in terms of $K(W')$ for $W' < W$
- Which W' values to consider?
  - Since the ith item has a weight w_i, we should consider only $W - w_i$ for different i.

Knapsack w/ repetition: Identify subproblems

- Consider subproblems by reducing the weight
  - Compute $K(W)$ in terms of $K(W')$ for $W' < W$
- Which W' values to consider?
  - Since the ith item has a weight w_i, we should consider only $W - w_i$ for different i.
- *Optimal substructure:* If $K(W)$ is the optimal solution and it includes item i, then $K(W) = K(W - w_i) + v_i$

Knapsack w/ repetition

KnapWithRep(w, v, n, W)

$K[0] = 0$

for $w = 1$ to W **do**

$K[w] = max_{1 \leq i \leq n, w[i] \leq w}(K[w - w[i]] + v[i])$

return $K[W]$

Knapsack w/ repetition

KnapWithRep(w, v, n, W)

$K[0] = 0$

for $w = 1$ to W **do**

$K[w] = max_{1 \leq i \leq n, w[i] \leq w}(K[w - w[i]] + v[i])$

return $K[W]$

- Fills the array K from left-to-right

Knapsack w/ repetition

KnapWithRep(w, v, n, W)

$K[0] = 0$

for $w = 1$ to W **do**

$K[w] = max_{1 \leq i \leq n, w[i] \leq w}(K[w - w[i]] + v[i])$

return $K[W]$

- Fills the array K from left-to-right
  - If you construct the dag explicitly, you will see that we are looking for the longest path!

Knapsack w/ repetition

KnapWithRep(w, v, n, W)

$K[0] = 0$

for $w = 1$ to W **do**

$K[w] = max_{1 \leq i \leq n, w[i] \leq w}(K[w - w[i]] + v[i])$

return $K[W]$

- Fills the array K from left-to-right
  - If you construct the dag explicitly, you will see that we are looking for the longest path!
- *Runtime:* Outer loop iterates W times, *max* takes $O(n)$ time, for a total of $O(nW)$ time
  - *Not polynomial:* input size logarithmic (not linear) in W.

0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

Key idea: Define 2-d array $K[u, j]$ which computes optimal value for weight u achievable using items $1..j$

0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

Key idea: Define 2-d array $K[u, j]$ which computes optimal value for weight u achievable using items $1..j$

- $K[u, j]$ can be computed from $K[\_, j-1]$

0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

Key idea: Define 2-d array $K[u, j]$ which computes optimal value for weight u achievable using items $1..j$

- $K[u, j]$ can be computed from $K[\_, j-1]$
  - Either item j is not included in the optimal solution. Then $K[u, j] = K[u, j-1]$

0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

Key idea: Define 2-d array $K[u, j]$ which computes optimal value for weight u achievable using items $1..j$

- $K[u, j]$ can be computed from $K[\_, j-1]$
  - Either item j is not included in the optimal solution. Then $K[u, j] = K[u, j-1]$
  - Or, j is included, so $K[u, j] = v[j] + K[u - w[j], j-1]$

0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

Key idea: Define 2-d array $K[u, j]$ which computes optimal value for weight u achievable using items $1..j$

- $K[u, j]$ can be computed from $K[\_, j - 1]$
  - Either item j is not included in the optimal solution. Then $K[u, j] = K[u, j-1]$
  - Or, j is included, so $K[u, j] = v[j] + K[u-w[j], j-1]$
- So, fill up the array K as j goes from 1 to n

0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

Key idea: Define 2-d array $K[u, j]$ which computes optimal value for weight u achievable using items $1..j$

- $K[u, j]$ can be computed from $K[\_, j-1]$
  - Either item j is not included in the optimal solution. Then $K[u, j] = K[u, j-1]$
  - Or, j is included, so $K[u, j] = v[j] + K[u - w[j], j-1]$
- So, fill up the array K as j goes from 1 to n
- For each j, fill K as u goes from 1 to W

0-1 Knapsack Algorithm

$Knap01(w, v, n, W)$

$K[u, 0] = K[0, j] = 0, \forall 1 \leq u \leq W, 1 \leq j \leq n$

for $j = 1$ to n **do**

  for $u = 1$ to W **do**

    if $w[j] > u$ **then** $K[u, j] = K[u, j-1]$

    else $K[u, j] = max(K[u, j-1],$

          $K[u-w[j], j-1] + v[j])$

return $K[W, n]$

0-1 Knapsack Algorithm

Knap01(w, v, n, W)

$K[u, 0] = K[0, j] = 0, \forall 1 \leq u \leq W, 1 \leq j \leq n$

for $j = 1$ to n **do**

  for $u = 1$ to W **do**

    if $w[j] > u$ **then** $K[u, j] = K[u, j-1]$

    else $K[u, j] = max(K[u, j-1],$

        $K[u-w[j], j-1] + v[j])$

return $K[W, n]$

Runtime: As compared to unbounded knapsack, we have a nested loop here, but the inner loop now executes in $O(1)$ time. So runtime is still $O(nW)$

Recursive formulation of Dynamic programming

- Recursive formulation can often simplify algorithm presentation, avoiding need for explicit scheduling
  - Dependencies between subproblems can be left implicit an equation such as $K[w] = K[w - w[j]] + v[j]$
  - A call to compute $K[w]$ will automatically result in a call to compute $K[w - w[j]]$ because of dependency
  - *Can avoid solving (some) unneeded subproblems*

Recursive formulation of Dynamic programming

- Recursive formulation can often simplify algorithm presentation, avoiding need for explicit scheduling
  - Dependencies between subproblems can be left implicit an equation such as $K[w] = K[w - w[j]] + v[j]$
  - A call to compute $K[w]$ will automatically result in a call to compute $K[w - w[j]]$ because of dependency
  - *Can avoid solving (some) unneeded subproblems*
- *Memoization:* Remember solutions to function calls so that repeat invocations can use previously returned solutions

Recursive 0-1 Knapsack Algorithm

$BestVal01(u, j)$

if $u = 0$ **or** $j = 0$ **return** 0

if $w[j] > u$ **return** $BestVal01(u, j-1)$

else return $max(BestVal01(u, j-1), v[j] + BestVal01(u-w[j], j-1))$

- Much simpler in structure than iterative version
- Unneeded entries are not computed, e.g. $BestVal01(3, \_)$ when all weights involved are even
- *Exercise:* Write a recursive version of ChainMM.

Note: m_i's give us the dimension of matrices, specifically, M_i is an $m_{i-1} \times m_i$ matrix

Recursive 0-1 Knapsack Algorithm

***BestVal*01(u, j)**

if $u = 0$ **or** $j = 0$ **return** 0

if $w[j] > u$ **return** $BestVal01(u, j-1)$

else return $max(BestVal01(u, j-1), v[j] + BestVal01(u-w[j], j-1))$

- Much simpler in structure than iterative version
- Unneeded entries are not computed, e.g. $BestVal01(3, \_)$ when all weights involved are even
- *Exercise:* Write a recursive version of ChainMM.

Note: m_i's give us the dimension of matrices, specifically, M_i is an $m_{i-1} \times m_i$ matrix

Complexity: $O(n^3)$

Key step in Dyn. Prog.: Identifying subproblems

i. **The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.**

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is therefore linear.

ii. **The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.**

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

The number of subproblems is $O(mn)$.

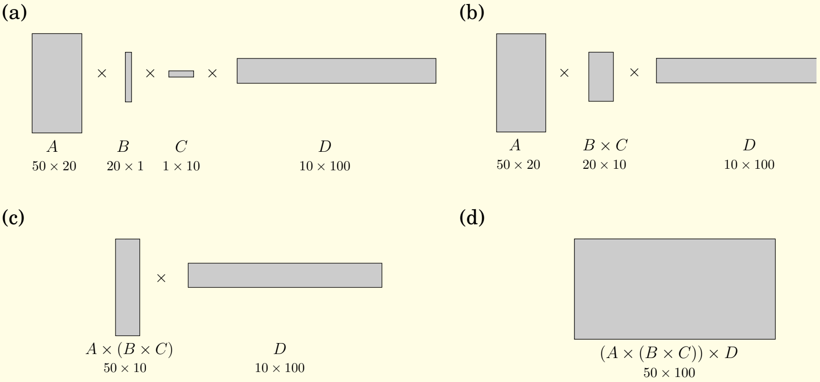
iii. **The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.**

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is $O(n^2)$.

iv. **The input is a rooted tree. A subproblem is a rooted subtree.**

Chain Matrix Multiplication



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

Chain MM: Formulating Optimal Solution

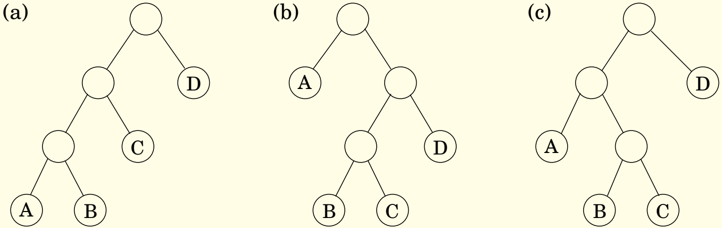
Consider outermost multiplication: $(M_1 \times \cdots \times M_j) \times (M_{j+1} \times \cdots \times M_n)$ — we could compute j using dynamic programming

Optimal substructure: Note that the optimal solution for $(M_1 \times \cdots \times M_j) \times (M_{j+1} \times \cdots \times M_n)$ must rely on optimal solutions to $M_1 \times \cdots \times M_j$ and $M_{j+1} \times \cdots \times M_n$ — or else we could improve the overall solution still

Cost function: This suggests a cost function $C[k, l]$ to denote the optimal cost of $M_k \times \cdots \times M_l$

Chain MM: Formulating Optimal Solution

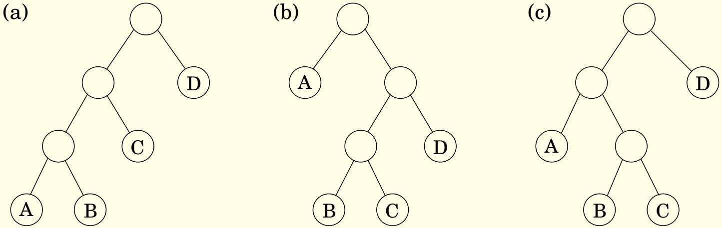
Figure 6.7 **(a)** $((A \times B) \times C) \times D$; **(b)** $A \times ((B \times C) \times D)$; **(c)** $(A \times (B \times C)) \times D$.



- Subproblems correspond to one of the subtrees

Chain MM: Formulating Optimal Solution

Figure 6.7 (a) $((A \times B) \times C) \times D$; **(b)** $A \times ((B \times C) \times D)$; **(c)** $(A \times (B \times C)) \times D$.



- Subproblems correspond to one of the subtrees
- Since order of multiplications can't be changed, each subtree must correspond to a "substring" of multiplications, i.e., $M_k \times \cdots \times M_l$

Chain MM Algorithm

chainMM(m, n)

$C[i, i] = 0 \; \forall 1 \leq i \leq n$
for $s = 1$ to $n - 1$ **do**
  for $k = 1$ to $n - s$ **do**
    $l = k + s$
    $C[k, l] = min_{k \leq i < l}(C[k, i] + C[i + 1, l] + m_{k-1} * m_i * m_l)$
return $C[1, n]$

- Recall: subproblems correspond to substrings: $M_k \times \cdots \times M_l$
- We iterate in increasing order of substring length
  - s goes from 1 to $n - 1$ and represents substring length minus 1.
- Substrings of same lengths are considered left to right,
  - k goes from 1 to $n - s$ and represents the starting position of substring

Chain MM Algorithm

chainMM(m, n)

$C[i, i] = 0 \; \forall 1 \leq i \leq n$

for $s = 1$ to $n - 1$ **do**

  for $k = 1$ to $n - s$ **do**

    $l = k + s$

    $C[k, l] = min_{k \leq i < l}(C[k, i] + C[i + 1, l] + m_{k-1} * m_i * m_l)$

return $C[1, n]$

Note: m_i's give us the dimension of matrices, specifically, M_i is an $m_{i-1} \times m_i$ matrix

Chain MM Algorithm

chainMM(m, n)

$C[i, i] = 0 \; \forall 1 \leq i \leq n$

for $s = 1$ to $n - 1$ **do**

  for $k = 1$ to $n - s$ **do**

    $l = k + s$

    $C[k, l] = min_{k \leq i < l}(C[k, i] + C[i + 1, l] + m_{k-1} * m_i * m_l)$

return $C[1, n]$

Note: m_i's give us the dimension of matrices, specifically, M_i is an $m_{i-1} \times m_i$ matrix

Complexity: $O(n^3)$

Subsequence

Definition

A sequence $a[1..m]$ is a subsequence of $b[1..n]$ occurring at position r if there exist $i_1, \ldots, i_k$ such that $a[r..(r+l-1)] = b[i_1]b[i_2]\cdots b[i_l]$, where $i_j < i_{j+1}$

Subsequence

Definition

A sequence $a[1..m]$ is a subsequence of $b[1..n]$ occurring at position r if there exist $i_1, ..., i_k$ such that $a[r..(r+l-1)] = b[i_1]b[i_2]\cdots b[i_l]$, where $i_j < i_{j+1}$

The relative order of elements is preserved in a subsequence, but unlike a substring, the elements needs not be contiguous.

Example: *BDEFHJ* is a subsequence of *ABCDEFGHIJK*

Longest Common Subsequence

Definition (LCS)

The LCS of two sequences $x[1..m]$ and $y[1..n]$ is the longest sequence $z[1..k]$ that is a subsequence of both x and y.

Longest Common Subsequence

Definition (LCS)

The LCS of two sequences $x[1..m]$ and $y[1..n]$ is the longest sequence $z[1..k]$ that is a subsequence of both x and y.

Example: $BEHJ$ is a common subsequence of $A\underline{B}CD\underline{E}FG\underline{H}I\underline{J}KLM$ and $AA\underline{B}BX\underline{E}J\underline{HJ}Z$

Longest Common Subsequence

Definition (LCS)

The LCS of two sequences $x[1..m]$ and $y[1..n]$ is the longest sequence $z[1..k]$ that is a subsequence of both x and y.

Example: $BEHJ$ is a common subsequence of $A\underline{B}CD\underline{E}FG\underline{H}I\underline{J}KLM$ and $AA\underline{B}BX\underline{E}J\underline{HJ}Z$

By aligning elements of z with the corresponding elements of x and y, we can compare x and y

Longest Common Subsequence

Definition (LCS)

The LCS of two sequences $x[1..m]$ and $y[1..n]$ is the longest sequence $z[1..k]$ that is a subsequence of both x and y.

Example: $BEHJ$ is a common subsequence of $A\underline{B}CD\underline{E}FG\underline{H}I\underline{J}KLM$ and $AA\underline{B}BX\underline{E}J\underline{HJ}Z$

By aligning elements of z with the corresponding elements of x and y, we can compare x and y

$$
\begin{array}{llllllllll}
x: & P & R & O & F & - & E & S & S & O & R \\
z: & P & R & O & F & - & E & S & - & - & R \\
y: & P & R & O & F & F_{ins} & E & S & -_{del} & U_{sub} & R
\end{array}
$$

to identify *edit* operations (insert/delete/substitute) operations needed to map x to y

Edit (Levenshtein) distance

Definition (ED)

Given sequences x and y and functions I, D and S that associate costs with each insert, delete and substitute operations, what is the minimum cost of any the edit sequence that transforms x into y.

Edit (Levenshtein) distance

Definition (ED)

Given sequences x and y and functions I, D and S that associate costs with each insert, delete and substitute operations, what is the minimum cost of any the edit sequence that transforms x into y.

Applications

- Spell correction (Levenshtein automata)
- `diff`
- In the context of version control, reconcile/merge concurrent updates by different users.
- DNA sequence alignment, evolutionary trees and other applications in computational biology

Towards a dynamic programming solution (1)

What subproblems to consider?

Towards a dynamic programming solution (1)

What subproblems to consider?

- Just like the LIS problem, we proceed from left to right, i.e., compute $L[j]$ as j goes from1 to n

Towards a dynamic programming solution (1)

What subproblems to consider?

- Just like the LIS problem, we proceed from left to right, i.e., compute $L[j]$ as j goes from1 to n
- But there are two strings x and y for LCS, so the subproblems correspond to prefixes of both x and y — there are $O(mn)$ such prefixes.

Towards a dynamic programming solution (1)

What subproblems to consider?

- Just like the LIS problem, we proceed from left to right, i.e., compute $L[j]$ as j goes from1 to n
- But there are two strings x and y for LCS, so the subproblems correspond to prefixes of both x and y — there are $O(mn)$ such prefixes.

E X P O N E N	T I A L
P O L Y N	O M I A L

The subproblem above can be represented as $E[7, 5]$.

$E[i, j]$ represents the edit distance of $x[1..i]$ and $y[1..j]$

Towards a dynamic programming solution (2)

For $E[k, l]$, consider the following possibilities:

Towards a dynamic programming solution (2)

For $E[k, l]$, consider the following possibilities:

- $x[k] = y[l]$: in this case, $E[k, l] = E[k-1, l-1]$ — the edit distance has not increased as we extend the string by one character, since these characters match

Towards a dynamic programming solution (2)

For $E[k, l]$, consider the following possibilities:

- $x[k] = y[l]$: in this case, $E[k, l] = E[k-1, l-1]$ — the edit distance has not increased as we extend the string by one character, since these characters match
- $x[k] \neq y[l]$: Three possibilities

Towards a dynamic programming solution (2)

For $E[k, l]$, consider the following possibilities:

- $x[k] = y[l]$: in this case, $E[k, l] = E[k-1, l-1]$ — the edit distance has not increased as we extend the string by one character, since these characters match
- $x[k] \neq y[l]$: Three possibilities
  - extend $E[k-1, l]$ by deleting $x[k]$:
    - $E[k, l] = E[k-1, l] + DC(x[k])$

Towards a dynamic programming solution (2)

For $E[k, l]$, consider the following possibilities:

- $x[k] = y[l]$: in this case, $E[k, l] = E[k-1, l-1]$ — the edit distance has not increased as we extend the string by one character, since these characters match
- $x[k] \neq y[l]$: Three possibilities
  - extend $E[k-1, l]$ by deleting $x[k]$:
    - $E[k, l] = E[k-1, l] + DC(x[k])$
  - extend $E[k, l-1]$ by inserting $y[l]$:
    - $E[k, l] = E[k, l-1] + IC(y[l])$

Towards a dynamic programming solution (2)

For $E[k, l]$, consider the following possibilities:

- $x[k] = y[l]$: in this case, $E[k, l] = E[k-1, l-1]$ — the edit distance has not increased as we extend the string by one character, since these characters match
- $x[k] \neq y[l]$: Three possibilities
  - extend $E[k-1, l]$ by deleting $x[k]$:
    - $E[k, l] = E[k-1, l] + DC(x[k])$
  - extend $E[k, l-1]$ by inserting $y[l]$:
    - $E[k, l] = E[k, l-1] + IC(y[l])$
  - extend $E[k-1, l-1]$ by substituting $x[k]$ with $y[l]$:
    - $E[k, l] = E[k-1, l-1] + SC(x[k], y[l])$

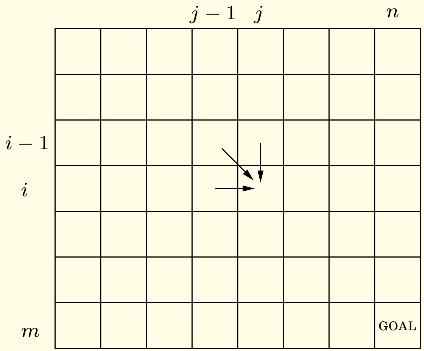
Towards a dynamic programming solution (3)

$$
\begin{aligned}
E[k,l] = \quad & min(\quad E[k-1,l] + DC(x[k]), && // \downarrow \\
& \qquad E[k,l-1] + IC(y[l]), && // \rightarrow \\
& \qquad E[k-1,l-1] + SC(x[k],y[l])) && // \searrow \\
E[0,l] = \quad & \sum_{i=1}^{l} IC(y[i]) \\
E[k,0] = \quad & \sum_{i=1}^{k} DC(x[i])
\end{aligned}
$$

Edit distance $= E[m,n]$

(Recall: m and n are lengths of strings x and y)

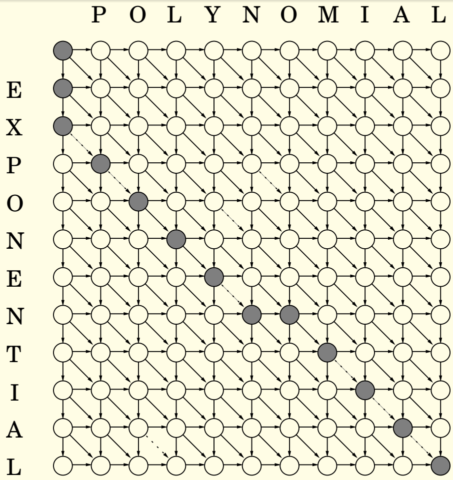
Towards a dynamic programming solution (4)



		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

$$\begin{aligned} E[k,l] = \quad & min(E[k-1,l] + DC(x[k]), && // \downarrow \\ & E[k,l-1] + IC(y[l]), && // \rightarrow \\ & E[k-1,l-1] + SC(x[k],y[l])) && // \searrow \end{aligned}$$

Towards a dynamic programming solution (5)



$$E[k,l] = min(E[k-1,l] + DC(x[k]), E[k,l-1] + IC(y[l]), E[k-1,l-1] + SC(x[k],y[l]))$$

Variations

Approximate prefix:

Is y approx. prefix of x?

Variations

Approximate prefix:

Is y approx. prefix of x? Decide based on

$$max_{1 \leq k \leq m} E[k, n]$$

Variations

Approximate prefix:

Is y approx. prefix of x? Decide based on

$$max_{1 \leq k \leq m} E[k, n]$$

Approximate suffix:

Variations

Approximate prefix:

Is y approx. prefix of x? Decide based on

$$max_{1 \leq k \leq m} E[k, n]$$

Approximate suffix:

Initialize $E[k, 0] = 0$, use $E[m, n]$ to determine if y is an approximate suffix of x

Variations

Approximate prefix:

Is y approx. prefix of x? Decide based on

$$max_{1 \leq k \leq m} E[k, n]$$

Approximate suffix:

Initialize $E[k, 0] = 0$, use $E[m, n]$ to determine if y is an approximate suffix of x

Approximate substring:

Variations

Approximate prefix:

Is y approx. prefix of x? Decide based on

$$max_{1\leq k\leq m}E[k, n]$$

Approximate suffix:

Initialize $E[k, 0] = 0$, use $E[m, n]$ to determine if y is an approximate suffix of x

Approximate substring:

Initialize $E[k, 0] = 0$, use $max_{1\leq k\leq m}E[k, n]$ to decide if y is an approx. substring of x.

More variations

Supporting transpositions:

More variations

Supporting transpositions:

Use a fourth term within *min*:

$$E[k-2, l-2] + TC(x[k-1]x[k], y[l-1]y[l])$$

where TC is a small value for transposed characters, and ∞ otherwise.

Similarity Vs Edit-distance

Edit-distance cannot be interpreted on its own, and needs to take into account the lengths of strings involved.

Similarity can stand on its own.

$$
\begin{aligned}
S[k,l] = \quad & max(S[k-1,l] - DC(x[k]), && // \downarrow \\
& S[k,l-1] - IC(y[l]), && // \rightarrow \\
& S[k-1,l-1] - SC(x[k],y[l])) && // \searrow
\end{aligned}
$$

$$S[0,l] =$$

$$S[k,0] =$$

- $SC(r,r)$ should be negative, while IC and DC should be positive.
- Formulations in biology are usually based on similarity

Similarity Vs Edit-distance

Edit-distance cannot be interpreted on its own, and needs to take into account the lengths of strings involved.

Similarity can stand on its own.

$$
\begin{aligned}
S[k,l] = \quad & max(S[k-1,l] - DC(x[k]), && // \downarrow \\
& \qquad S[k,l-1] - IC(y[l]), && // \rightarrow \\
& \qquad S[k-1,l-1] - SC(x[k],y[l])) && // \searrow \\
S[0,l] = \quad & -\sum_{i=1}^{l} IC(y[i]) \\
S[k,0] = \quad & -\sum_{i=1}^{k} DC(x[i])
\end{aligned}
$$

- $SC(r,r)$ should be negative, while IC and DC should be positive.
- Formulations in biology are usually based on similarity

Global alignment (DNA, proteins, ...)

- Similar to edit distance, but uses similarity scores

Global alignment (DNA, proteins, ...)

- Similar to edit distance, but uses similarity scores
- Gaps are scored differently: a contiguous sequence of n deletions does not get penalized as much as n times a single deletion. (same applies to insertions.)

Global alignment (DNA, proteins, ...)

- Similar to edit distance, but uses similarity scores
- Gaps are scored differently: a contiguous sequence of n deletions does not get penalized as much as n times a single deletion. (same applies to insertions.)
- Captures the idea that large deletions/insertions are much more likely in nature than many small ones. (Think of chromosomal crossover during meiotic cell division.)

Global alignment (DNA, proteins, ...)

- Similar to edit distance, but uses similarity scores
- Gaps are scored differently: a contiguous sequence of n deletions does not get penalized as much as n times a single deletion. (same applies to insertions.)
- Captures the idea that large deletions/insertions are much more likely in nature than many small ones. (Think of chromosomal crossover during meiotic cell division.)
- Obvious formulation to support such gap metrics will lead to more expensive algorithms: $S[k, l]$ depends on $S[k-d, l]$ and $S[k, l-i]$ for any $d < k$ and $i < l$. But a more careful formulation can get back to quadratic time

Global alignment (DNA, proteins, ...)

- Similar to edit distance, but uses similarity scores
- Gaps are scored differently: a contiguous sequence of n deletions does not get penalized as much as n times a single deletion. (same applies to insertions.)
- Captures the idea that large deletions/insertions are much more likely in nature than many small ones. (Think of chromosomal crossover during meiotic cell division.)
- Obvious formulation to support such gap metrics will lead to more expensive algorithms: $S[k, l]$ depends on $S[k-d, l]$ and $S[k, l-i]$ for any $d < k$ and $i < l$. But a more careful formulation can get back to quadratic time
- Quadratic time still too slow for sequence alignment.

Local alignment

- Aimed at identifying local regions of similarity, specifically, the best matches between subsequences of x and y

Local alignment

- Aimed at identifying local regions of similarity, specifically, the best matches between subsequences of x and y
- $S[i, j]$ now represents the best alignment of some suffix of $x[1..i]$ and $y[1..j]$.

Local alignment

- Aimed at identifying local regions of similarity, specifically, the best matches between subsequences of x and y
- $S[i, j]$ now represents the best alignment of some suffix of $x[1..i]$ and $y[1..j]$.
- A new term is introduced within *max*, namely, zero. This means that costs can never become negative.
- In other words, a subsequence does not incur costs because of mismatches preceding the subsequence.

Local alignment

- Aimed at identifying local regions of similarity, specifically, the best matches between subsequences of x and y
- $S[i, j]$ now represents the best alignment of some suffix of $x[1..i]$ and $y[1..j]$.
- A new term is introduced within *max*, namely, zero. This means that costs can never become negative.
- In other words, a subsequence does not incur costs because of mismatches preceding the subsequence.
- This change enables regions of similarity to stand out as positive scores.

Local alignment

- Aimed at identifying local regions of similarity, specifically, the best matches between subsequences of x and y
- $S[i,j]$ now represents the best alignment of some suffix of $x[1..i]$ and $y[1..j]$.
- A new term is introduced within *max*, namely, zero. This means that costs can never become negative.
- In other words, a subsequence does not incur costs because of mismatches preceding the subsequence.
- This change enables regions of similarity to stand out as positive scores.
- Initialize $F[i,0] = F[0,j] = 0$

Improvements to ED Algorithm

Linear-space: $O(mn)$ is not so good for large m, n.

Improvements to ED Algorithm

Linear-space: $O(mn)$ is not so good for large m, n.

- Slow in terms of runtime
- (Possibly) unacceptable in terms of space usage

Improvements to ED Algorithm

Linear-space: $O(mn)$ is not so good for large m, n.

Slow in terms of runtime

(Possibly) unacceptable in terms of space usage

- If we are only interested in ED, we can use linear space: retain only the last column computed.
- But if want the actual edits, we need $O(mn)$ space with the algorithms discussed so far.

Linear-space algorithms developed to overcome this problem.

Improvements to ED Algorithm

Linear-space: $O(mn)$ is not so good for large m, n.

Slow in terms of runtime

(Possibly) unacceptable in terms of space usage

- If we are only interested in ED, we can use linear space: retain only the last column computed.
- But if want the actual edits, we need $O(mn)$ space with the algorithms discussed so far.

Linear-space algorithms developed to overcome this problem.

Better overall performance: $O(md)$ space and runtime if the max. distance $\leq d$.

In the interest of time, we won't cover these extensions. They are fairly involved, but not necessarily hard.

LCS application: UNIX `diff`

Each line is considered a "character:"

- Number of lines far smaller than number of characters
- Difference at the level of lines is easy to convey to users
- Much higher degree of confidence when things line up. Leads to better results on programs.

  But does not work that well on document types where line breaks are not meaningful, e.g., text files where each paragraph is a line.

Aligns lines that are preserved.

- The edits are then printed in the familiar "diff" format.

LCS applications: version control, patch,...

Software patches often distributed as "diffs." Programs such as `patch` can apply these patches to source code or any other file.

LCS applications: version control, patch,...

Software patches often distributed as "diffs." Programs such as `patch` can apply these patches to source code or any other file.

Concurrent updates in version control systems are resolved using LCS.

LCS applications: version control, patch,...

Software patches often distributed as "diffs." Programs such as `patch` can apply these patches to source code or any other file.

Concurrent updates in version control systems are resolved using LCS.

- Let x be the version in the repository
- Suppose that user A checks it out, edits it to get version y
- Meanwhile, B also checks out x, edits it to z.
- If $x \longrightarrow y$ edits target a disjoint set of locations from those targeted by the $x \longrightarrow z$ edits, both edits can be committed; otherwise a conflict is reported.

Summary

- A general approach for *optimization problems*
- *Applicable in the presence of:*
  - Optimal substructure
  - A natural ordering among subproblems
  - Numerous subproblems (often, exponential), but only some (polynomial number) are distinct