cs473: Algorithms

Lecture 6: Dynamic Programming

Michael A. Forbes Chandra Chekuri

University of Illinois at Urbana-Champaign

September 12, 2019

Overview

logistics:

- pset2 out, due W10 — can submit in *groups* of ≤ 3

last time:

- shortest paths
  - with negative lengths
  - all-pairs

today:

- dynamic programming *optimized*
  - edit distance
  - longest increasing subsequence

Dynamic Programming

dynamic programming:

- develop recursive algorithm
- understand structure of subproblems
  - names of subproblems
  - number of subproblems
  - dependency graph amongst subproblems
- memoize (implicitly, or explicitly)
- analysis (time, space)
- further optimization

remarks:

- memoizing a recursive algorithm does not necessarily lead to an efficient algorithm (e.g., knapsack problem) — you need the *right* recursion
- recognizing that dynamic programming applies to a problem can be non-obvious

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. The **edit distance** between x and y is the minimum number of insertions, deletions and substitutions required to transform x into y.

Example

$\underline{\text{m}}\text{oney} \to \text{bone}\underline{\text{y}} \to \text{bon}\underline{\text{e}} \to \text{bo}\underline{\text{n}}\text{a} \to \text{bo}\underline{\ \ }\text{a} \to \text{boba} \implies \text{edit distance} \leq 5$

remarks:

- edit distance ≤ 4
- intermediate strings can be arbitrary in $\Sigma^\star$

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. An **alignment** is a sequence M of pairs of indices (i, j) such that

- an index could be empty, such as $(, 4)$ or $(5,)$
- each index appears exactly once per coordinate
- no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$, *or* $i > i'$ and $j > j'$

The **cost** of an alignment is the number of pairs (i, j) where $x_i \neq y_j$.

Example

```
mon ey
bo ba
```

$M = \{(1, 1), (2, 2), (3,), (, 3), (4, 4), (5,)\}$, cost 5

Edit Distance (III)

question: given two strings $x, y \in \Sigma^\star$, compute their edit distance

Lemma

The edit distance between two strings $x, y \in \Sigma^\star$ is the minimum cost of an alignment.

Proof.

Exercise. □

question: given two strings $x, y \in \Sigma^\star$, compute the minimum cost of an alignment

remarks:

- can *also* ask to compute the alignment itself
- widely solved in practice, e.g., the BLAST heuristic for DNA edit distance

Edit Distance (IV)

Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\operatorname{dist}(x \circ a, y \circ b) = \min \begin{cases} \operatorname{dist}(x, y) + \mathbb{1}[\![a \neq b]\!] \\ \operatorname{dist}(x, y \circ b) + 1 \\ \operatorname{dist}(x \circ a, y) + 1 \end{cases} .$$

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- a aligns to b, with cost $\mathbb{1}[\![a \neq b]\!]$
- a is deleted, with cost 1
- b is deleted, with cost 1

$\square$

Edit Distance (V)

iterative algorithm:

dist$(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

```
for 0 ≤ i ≤ n
    d[i][0] = i
for 0 ≤ j ≤ m
    d[0][j] = j
for 0 ≤ i ≤ n
    for 0 ≤ j ≤ m
```

$$d[i][j] = \min \begin{cases} d[i-1][j-1] + \mathbb{1}[\![x_i \neq y_j]\!] \\ d[i-1][j] + 1 \\ d[i][j-1] + 1 \end{cases}$$

```
return d[n][m]
```

correctness: clear

complexity:

- $O(nm)$ time
- space
  - clearly $O(nm)$
  - *better:* only store $d[\mathrm{cur}][\cdot]$ and $d[\mathrm{prev}][\cdot] \implies O(m)$

question: are we done?

Edit Distance (VI)

Corollary

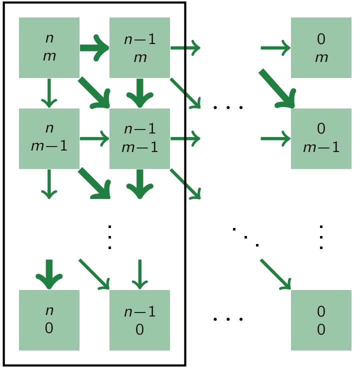
Given two strings $x, y \in \Sigma^\star$ *can compute the minimum cost alignment in* $O(nm)$*-time and* $O(nm)$*-space.*

Proof.

Exercise. *Hint:* follow *how* each subproblem was solved. $\square$

Edit Distance (VII)

dependency graph:



computing the alignment:

- *how* update rule is computed yields a pointer for each (i, j)
- one pointer per optimal choice — multiple pointers are possible
- *any* path from (n, m) to boundary yields optimal alignment
- compute path via graph search

saving space:

- only keep most recent two columns

$\Longrightarrow$ we lost the pointers!

question: compute the alignment in $O(n + m)$ space?

Edit Distance, Better

Lemma

Let $x, y \in \Sigma^\star$ *be strings, with* $n = |x|$ *and* $m = |y|$*. Then for any* $1 \le i \le n$*,*

$$\mathrm{dist}(x, y) = \min_{1 \le j \le m} \{\mathrm{dist}(x_{\le i}, y_{\le j}) + \mathrm{dist}(x_{>i}, y_{>j})\} \; .$$

Proof.

$\underline{\le}$: Fix j. Let $A_\le$ and $A_>$ be alignments respectively between $x_{\le i}, y_{\le j}$ and $x_{>i}, y_{>j}$, with respective costs $\mathrm{dist}(x_{\le i}, y_{\le j})$ and $\mathrm{dist}(x_{>i}, y_{>j})$. Then $A_\le \circ A_>$ is an alignment between x and y of cost $\mathrm{dist}(x_{\le i}, y_{\le j}) + \mathrm{dist}(x_{>i}, y_{>j})$.

$\underline{=}$: Any alignment A between x and y will align $x_{\le i}$ to some prefix $y_{\le j}$ of y in an alignment $A_\le$, and align $x_{>i}$ to the suffix $y_{>j}$ in an alignment $A_>$, and hence for this j we have $\mathrm{dist}(x, y) = \mathrm{dist}(x_{\le i}, y_{\le j}) + \mathrm{dist}(x_{>i}, y_{>j})$. $\square$

Edit Distance, Better (II)

Definition

Let $x, y \in \Sigma^\star$ be strings, with $n = |x|$ and $m = |y|$. Then for any $1 \leq i \leq n$, define $\text{meet}_i(x, y)$ to be the $j \in [m]$ where $x_{\leq i}$ aligns to $y_{\leq j}$ in an optimal alignment. That is,

$$\text{meet}_i(x, y) = \min\{j : \text{dist}(x, y) = \text{dist}(x_{\leq i}, y_{\leq j}) + \text{dist}(x_{>i}, y_{>j})\} .$$

remark: previous lemma asserts such a j exists

meet $(i, x_1x_2\cdots x_n, y_1y_2\cdots y_m)$
    `for` $1 \leq j \leq m$
        `compute` $\text{dist}(x_{\leq i}, y_{\leq j})$
    `for` $1 \leq j \leq m$
        `compute` $\text{dist}(x_{>i}, y_{>j})$
    `output min` j `st` $\text{dist}(x, y) =$
        $\text{dist}(x_{\leq i}, y_{\leq j}) + \text{dist}(x_{>i}, y_{>j})$

correctness: clear

complexity:

- **dist**$(x_{\leq i}, y)$ *already* computes $\text{dist}(x_{\leq i}, y_{\leq j})$ for *all* j
  - $O(nm)$ time, $O(m)$ space
- **dist**$(\text{reverse}(x_{>i}), \text{reverse}(y))$ *already* computes $\text{dist}(x_{>i}, y_{>j})$ for *all* j

$\Longrightarrow$ $O(nm)$ time, $O(m)$ space

Edit Distance, Better (III)

divide and conqueror:

```
dist-align(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    if n = 1
        use dist(x, y)
    if m = 1
        use dist(x, y)
    j = meet(n − 1, x, y)
    A_≤ = dist-align(x_{≤n−1}, y_{≤j})
    A_> = dist-align(x_{>n−1}, y_{>j})
    return A_≤ ∘ A_>
```

correctness: clear

complexity:

- base cases
  - $O(m)$ time, $O(1)$ space
  - $O(n)$ time, $O(1)$ space
- $\text{meet}_{n-1}(x, y)$
  - $O(nm)$ time, $O(n+m)$ space
- space recurrence
  - $S(n, m) \leq \max\{O(n+m), S(n-1, m), S(1, m)\}$

  $\Longrightarrow\ S(n, m) \leq O(n+m)$
- time recurrence
  - $T(n, m) \leq O(nm) + T(n-1, m) + T(1, m)$

  $\Longrightarrow\ T(n, m) \leq O(n^2 m)$

question: can we do better?

Edit Distance, Better (IV)

divide and conqueror:

dist-align'$(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$
  `if` $n = 1$
    `use` **dist**(x, y)
  `if` $m = 1$
    `use` **dist**(x, y)
  $j =$ **meet**$(\lfloor \frac{n}{2} \rfloor, x, y)$
  $A_{\leq} =$ **dist-align'**$(x_{\leq \lfloor \frac{n}{2} \rfloor}, y_{\leq j})$
  $A_{>} =$ **dist-align'**$(x_{> \lfloor \frac{n}{2} \rfloor}, y_{> j})$
  `return` $A_{\leq} \circ A_{>}$

correctness: clear

complexity:

- base cases: $O(n + m)$ time, $O(1)$ space
- $\text{meet}_{\lfloor \frac{n}{2} \rfloor}(x, y)$: $O(nm)$ time, $O(n + m)$ space
- space recurrence
  - $S(n, m) \leq \max\{O(n + m), S(\lfloor \frac{n}{2} \rfloor, m), S(n - \lfloor \frac{n}{2} \rfloor, m)\}$

  $\Longrightarrow$ $S(n, m) \leq O(n + m)$
- time recurrence
  - $T(n, m) \leq O(nm) + T(\lfloor \frac{n}{2} \rfloor, j) + T(n - \lfloor \frac{n}{2} \rfloor, m - j)$
  - guess $T(n, m) \leq \alpha \cdot nm$
  - $T(n, m) \lesssim \beta \cdot nm + \alpha \cdot \frac{n}{2} \cdot j + \alpha \cdot \frac{n}{2} \cdot (m - j) = (\beta + \frac{\alpha}{2})nm$

  $\Longrightarrow$ valid as long as $\alpha \geq 2\beta$

  $\Longrightarrow$ $T(n, m) \leq O(nm)$

$\Longrightarrow$ computing actual alignment in $O(nm)$-time and $O(n + m)$-space.

Longest Increasing Subsequence

Definition

A **sequence** of integers, of **length** n, is an ordered list $a_1, a_2, \ldots, a_n \in \mathbb{Z}$. The sequence is **increasing** if $a_1 < a_2 < \cdots < a_n$.
A **subsequence** of $a_1, a_2, \ldots, a_n$ is any sequence of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_m}$, where $1 \leq i_1 < \cdots < i_m \leq n$. The subsequence is **increasing (IS)** if $a_{i_1} < \cdots < a_{i_n}$.

Example

- 02139947200854008540943059472061801 — sequence
- $021399472008540085409430594720\underline{61801}$ — *sub*sequence
- $02\underline{1}399\underline{4}72008\underline{5}4008540\underline{9}43059472061801$ — *increasing* subsequence
- $\underline{02}\underline{1}\underline{3}99\underline{4}72008\underline{5}400\underline{8}540\underline{9}43059472061801$ — *longer* increasing subsequence

Longest Increasing Subsequence (II)

Definition

The **longest increasing subsequence problem (LIS)** is to, given a sequence $a_1, a_2, \ldots, a_n \in \mathbb{Z}$, compute the (length of) the longest increasing subsequence.

goal: solve with dynamic programming

- identify subproblems
- develop recursion
- memoize
- analyze
- optimize *time*

remark: without loss of generality the a_i are *distinct*, up to a cost of $\Theta(n \log n)$ in runtime (**exercise**)

Longest Increasing Subsequence (III)

Lemma

For a sequence $\overline{a} = a_1, a_2, \ldots, a_n$*, define* $\mathsf{LIS}(\overline{a})$ *to be the length of the longest increasing subsequence. Define* $\mathsf{LIS}^\star(\overline{a})$ *to be the length of the longest increasing subsequence that* **contains the last element** a_n*. Then*

1. $\mathsf{LIS}(a_1, a_2, \ldots, a_n) = \max_{1 \le i \le n} \mathsf{LIS}^\star(a_1, a_2, \ldots, a_i)$.
2. $\mathsf{LIS}^\star(a_1, a_2, \ldots, a_n) = \max_{i : a_i < a_n} \{1 + \mathsf{LIS}^\star(a_1, a_2, \ldots, a_i), 1\}$.

Proof.

1. Clear.
2. For i with $a_i < a_n$, an $\mathsf{IS}^\star$ $a_{i_1} < \cdots < a_{i_{m-1}} < a_{i_m = i}$ of $\overline{a}_{\le i}$ can append a_n to yield an $\mathsf{IS}^\star$ $a_{i_1} < \cdots < a_{i_{m-1}} < a_i < a_n$ of $\overline{a}$, and *every* $\mathsf{IS}^\star$ of $\overline{a}$ can be decomposed this way, *or* by taking the singleton sequence a_n. Now take maximums. □

Longest Increasing Subsequence (IV)

Lemma

Define $\mathrm{LIS}^{\star}(\overline{a})$ *to be the length of the longest increasing subsequence that contains the last element* a_n*. Then* $\mathrm{LIS}^{\star}(a_1, a_2, \ldots, a_n) = \max_{i:a_i<a_n} \{1 + \mathrm{LIS}^{\star}(\overline{a}_{\leq i}), 1\}$.

Example

02139947200854008540943059472061801

1. $\underline{0}2139947200854008540943059472061801$ — $\mathrm{LIS}^{\star}(a_1) = 1$
2. $\underline{02}139947200854008540943059472061801$ — $\mathrm{LIS}^{\star}(a_1, a_2) = 2$
3. $\underline{0}2\underline{1}39947200854008540943059472061801$ — $\mathrm{LIS}^{\star}(a_1, \ldots, a_3) = 2$
4. $\underline{0}2\underline{13}9947200854008540943059472061801$ — $\mathrm{LIS}^{\star}(a_1, \ldots, a_4) = 3$
5. $\underline{0}2\underline{139}947200854008540943059472061801$ — $\mathrm{LIS}^{\star}(a_1, \ldots, a_5) = 4$
6. $\underline{0}2\underline{13}9\underline{9}47200854008540943059472061801$ — $\mathrm{LIS}^{\star}(a_1, \ldots, a_6) = 4$

Longest Increasing Subsequence (V)

iterative algorithm:

```
LIS(a_1, a_2, ..., a_n):
    for 1 ≤ i ≤ n
        L*[i] = 1
    L = 0
    for 1 ≤ i ≤ n
        for 1 ≤ j < i
            if a_j < a_i
                L*[i] = max{L*[i], 1 + L*[j]}
        L = max{L, L*[i]}
    return L
```

correctness: clear

complexity:

- $O(n)$ space
- $O(n^2)$ time — *do better?*

Longest Increasing Subsequence, Faster

$$\mathrm{LIS}^\star(a_1, a_2, \ldots, a_i) = \max_{i: a_j < a_i} \{1 + \mathrm{LIS}^\star(a_1, a_2, \ldots, a_j), 1\} .$$

This recursive step does too much — all (a_j, a_i) are compared! Use sorting?
idea: define subproblem based on *length* of increasing subsequences

Definition

For sequence $a_1, a_2, \ldots, a_n$, define the **end of increasing subsequence** $\mathrm{EIS}(\ell, \overline{a})$ to be the minimum a_i such that there is an increasing sequence of length ℓ that terminates at a_i, that is,

$$\mathrm{EIS}(\ell, \overline{a}) := \min_{i: a_{i_1} < a_{i_2} < \cdots < a_{i_\ell = i}} a_i .$$

$\mathrm{EIS}(\ell, \overline{a}) = \infty$ if $\ell > \mathrm{LIS}(\overline{a})$.

intuition: prefer the '*smallest*' IS of each size

Longest Increasing Subsequence, Faster (II)

Definition

For sequence $a_1, a_2, \dots, a_n$, define $\mathrm{EIS}(\ell, \overline{a})$ to be the minimum a_i such that there is an increasing sequence of length ℓ that terminates at a_i. $\mathrm{EIS}(\ell, \overline{a}) = \infty$ if $\ell > \mathrm{LIS}(\overline{a})$.

Lemma

$\mathrm{LIS}(\overline{a}) = \max_{\ell : \mathrm{EIS}(\ell, \overline{a}) < \infty} \ell$.

Proof.

Clear. $\square$

Longest Increasing Subsequence, Faster (III)

Definition

For sequence $a_1, a_2, \ldots, a_n$, define $\mathrm{EIS}(\ell, \overline{a})$ to be the minimum a_i such that there is an increasing sequence of length ℓ that terminates at a_i. $\mathrm{EIS}(\ell, \overline{a}) = \infty$ if $\ell > \mathrm{LIS}(\overline{a})$.

Lemma

For sequence $a_1, a_2, \ldots, a_n$, $\mathrm{EIS}(\ell, \overline{a}) < \mathrm{EIS}(\ell + 1, \overline{a})$, *for all* ℓ. *That is,* $\mathrm{EIS}(\cdot, \overline{a})$ *is a strictly sorted sequence.*

Proof.

Let $a_{i_1} < a_{i_2} < \cdots < a_{i_\ell}$ be a witness for $\mathrm{EIS}(\ell, \overline{a}) = a_{i_\ell}$, and let $a_{i'_1} < a_{i'_2} < \cdots < a_{i'_\ell} < a_{i'_{\ell+1}}$ be a witness for $\mathrm{EIS}(\ell + 1, \overline{a}) = a_{i'_{\ell+1}}$. Then as $a_{i'_1} < a_{i'_2} < \cdots < a_{i'_\ell}$ is length-ℓ increasing sequence we have that $\mathrm{EIS}(\ell, \overline{a}) \leq a_{i'_\ell} < a_{i'_{\ell+1}} = \mathrm{EIS}(\ell + 1, \overline{a})$. $\square$

Longest Increasing Subsequence, Faster (IV)

Lemma

$\mathrm{EIS}(\ell, (a_1, \ldots, a_n, a_{n+1})) =$

1. $\mathrm{EIS}(\ell, \bar{a})$, *if* $\mathrm{EIS}(\ell, \bar{a}) < a_{n+1}$
2. $\mathrm{EIS}(\ell, \bar{a})$, *if* $\mathrm{EIS}(\ell - 1, \bar{a}) > a_{n+1}$
3. a_{n+1}, *if* $\mathrm{EIS}(\ell, \bar{a}) > a_{n+1}$ **and** $\mathrm{EIS}(\ell - 1, \bar{a}) < a_{n+1}$

Proof.

1. Clear.
2. Clear.
3. Exists increasing sequence of length ℓ terminating at a_{n+1}

   iff exists increasing sequence of length $\ell - 1$ terminating at $a_i < a_{n+1}$, for some i

   iff exists increasing sequence of length $\ell - 1$ terminating at $\mathrm{EIS}(\ell - 1, \bar{a}) < a_{n+1}$ □

Longest Increasing Subsequence, Faster (V)

Lemma

For a fixed $\overline{a}$, $\mathrm{EIS}(\ell, \overline{a})$ *strictly increases with* ℓ.

Lemma

$\mathrm{EIS}(\ell, (a_1, \ldots, a_n, a_{n+1})) =$

1. $\mathrm{EIS}(\ell, \overline{a})$, *if* $\mathrm{EIS}(\ell, \overline{a}) < a_{n+1}$ **or** $\mathrm{EIS}(\ell - 1, \overline{a}) > a_{n+1}$
2. a_{n+1}, *if* $\mathrm{EIS}(\ell, \overline{a}) > a_{n+1}$ **and** $\mathrm{EIS}(\ell - 1, \overline{a}) < a_{n+1}$

Corollary

- $\mathrm{EIS}(\ell, (\overline{a}, a_{n+1})) \neq \mathrm{EIS}(\ell, \overline{a})$ *for* **exactly one** *value of* ℓ
- *This value of* ℓ *can be found by binary search.*

remarks:

- uses *distinctness* of the a_i
- boundary cases need attention, e.g., $\mathrm{EIS}(\ell, \overline{a}) = \infty$, or $\ell - 1 = 0$

Longest Increasing Subsequence, Faster (VI)

LIS'$(a_1, a_2, \ldots, a_n)$:
```
    for 1 ≤ ℓ ≤ n
        E[ℓ] = ∞
    for 1 ≤ i ≤ n
        ℓ = min{k : E[k] > a_i}
        E[ℓ] = a_i
    for 1 ≤ i ≤ n
        if E[i] < ∞
            L = i
    return L
```

correctness: clear

complexity:

- $O(n)$ space
- time
  - $E[\cdot]$ remains sorted throughout

  $\Longrightarrow$ $O(\log n)$ time to compute $\min\{k : E[k] > a_i\}$

  $\Longrightarrow$ $O(n \log n)$ total runtime

remarks:

- making a_i distinct costs $\Theta(n \log n)$ extra time
- can compute actual subsequence in same time bound, using back pointers (**exercise**)

Overview (II)

logistics:

- pset2 out, due W10 — can submit in *groups* of ≤ 3

today:

- dynamic programming *optimized*
  - edit distance
  - longest increasing subsequence

next time:

- randomized algorithms

TOC