CS 580: Algorithm Design and Analysis

Jeremiah Blocki
Purdue University
Spring 2018

Announcement: Homework 3 due February 15th at 11:59PM
Midterm Exam: Wed, Feb 21 (8PM-10PM) @ MTHW 210

Recap: Dynamic Programming

Key Idea: Express optimal solution in terms of solutions to smaller sub problems

Example 1: Weighted Interval Scheduling
- **OPT(j) is optimal solution considering only jobs 1,…,j**
- **OPT(j) = max{ v_j + OPT(p(j)), OPT(j-1)}**
- Case 1: Optimal solution includes job j with value v_j
  - Add job j and eliminate incompatible jobs p(j)+1,…,j-1
- Case 2: Optimal solution does not include job j

Example 2: Segmented Least Squares
- **Fit points to a sequence of several line segments**
- **Goal: Minimize E+cL**
  - **E squared error**
  - **L number of lines**
- **OPT(j) = min{e(i,j)+OPT(i-1)+c: i< j+1}**

OPT(j) is optimal solution considering only jobs 1,…,j

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.
- Given n objects and a "knapsack."
- Item i weighs w_i > 0 kilograms and has value v_i > 0.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

W = 11

Greedy: repeatedly add item with maximum ratio v_i / w_i.
Ex: { 5, 2, 1 } achieves only value = 35 ⇒ greedy not optimal.

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }
- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, W, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

7

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

8

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.
- Not polynomial in input size!
  - Only need $\log_2 W$ bits to encode each weight
  - Problem can be encoded with $O(n \log_2 W)$ bits
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

9

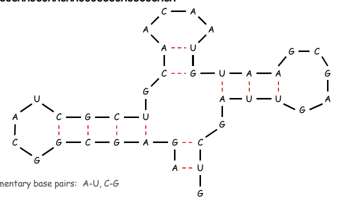
6.5 RNA Secondary Structure

RNA Secondary Structure

RNA. String $B = b_1 b_2 \ldots b_n$ over alphabet { A, C, G, U }.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA



C A A A A U C G U A A G C G A U U G A U G A U A C G C G C G C G A U C G A U G

complementary base pairs: A-U, C-G

11

RNA Secondary Structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:
- [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.

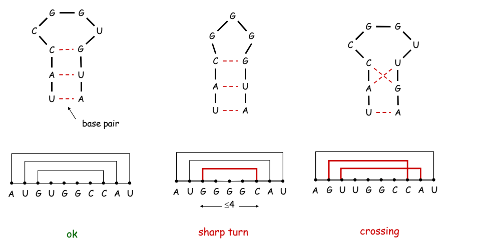
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.
↖ approximate by number of base pairs

Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

12

RNA Secondary Structure: Examples

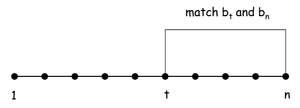
Examples.



13

RNA Secondary Structure: Subproblems

First attempt. OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_j$.



Difficulty. Results in two sub-problems.
- Finding secondary structure in: $b_1 b_2 \ldots b_{t-1}$. ← OPT(t-1)
- Finding secondary structure in: $b_{t+1} b_{t+2} \ldots b_{n-1}$.

need more sub-problems

14

Dynamic Programming Over Intervals

Notation. OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

- Case 1. If $i \geq j - 4$.
  - OPT(i, j) = 0 by no-sharp turns condition.
- Case 2. Base b_j is not involved in a pair.
  - OPT(i, j) = OPT(i, j-1)
- Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.
  - non-crossing constraint decouples resulting sub-problems
  - $\text{OPT}(i, j) = 1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \}$

take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements

Remark. Same core idea in CKY algorithm to parse context-free grammars.

15

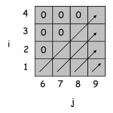
Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?
A. Do shortest intervals first.

```
RNA(b_1,…,b_n) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute M[i, j]

   return M[1, n]
}
```

using recurrence



Running time. $O(n^3)$.

16

Dynamic Programming Summary

Recipe.
- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.
- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy

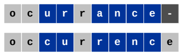
CKY parsing algorithm for context-free grammar has similar structure

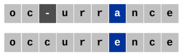
Top-down vs. bottom-up: different people have different intuitions.

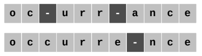
17

6.6 Sequence Alignment

String Similarity


6 mismatches, 1 gap


1 mismatch, 1 gap


0 mismatches, 3 gaps

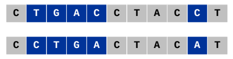
How similar are two strings?
- `ocurrance`
- `occurrence`

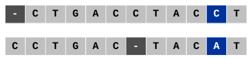
19

Edit Distance

Applications.
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
- Gap penalty δ; mismatch penalty α_{pq}.
- Cost = sum of gap and mismatch penalties.


$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$


$2\delta + \alpha_{CA}$

20

Sequence Alignment

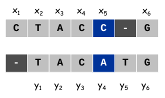
Goal: Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.

Def. The pair x_i-y_j and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i\,:\,x_i \text{ unmatched}} \delta + \sum_{j\,:\,y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG vs. TACATG.
Sol: $M = x_2$-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.


x_1 x_2 x_3 x_4 x_5 x_6
C T A C C - G
- T A C A T G
y_1 y_2 y_3 y_4 y_5 y_6

21

Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.
- Case 1: OPT matches x_i-y_j.
  - pay mismatch for x_i-y_j + min cost of aligning two strings $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$
- Case 2a: OPT leaves x_i unmatched.
  - pay gap for x_i and min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$
- Case 2b: OPT leaves y_j unmatched.
  - pay gap for y_j and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

22

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x₁x₂...xₘ, y₁y₂...yₙ, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xᵢ, yⱼ] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

Analysis. $\Theta(mn)$ time and space.
English words or sentences: $m, n \le 10$.
Computational biology: m = n = 100,000. 10 billions ops OK, but 10GB array?

23

6.7 Sequence Alignment in Linear Space

Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

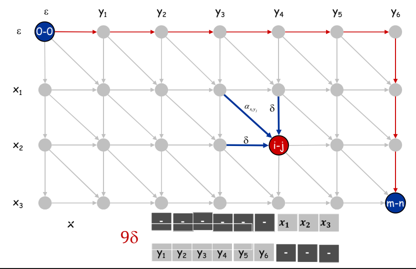
Easy. Optimal value in O(m + n) space and O(mn) time.
- Compute OPT(i, •) from OPT(i-1, •).
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal alignment in O(m + n) space and O(mn) time.
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

25

Sequence Alignment: Linear Space

Edit distance graph.
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).

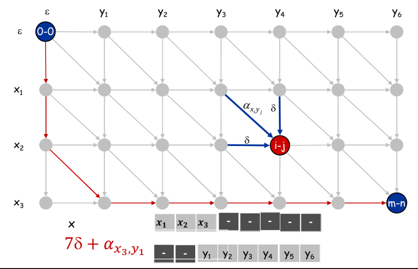


ε
y_1 y_2 y_3 y_4 y_5 y_6
x_1 x_2 x_3
0-0
$\alpha_{x_i y_j}$
δ
i-j
m-n
x
9δ
- - - - - - x_1 x_2 x_3
y_1 y_2 y_3 y_4 y_5 y_6 - - -

7

Sequence Alignment: Linear Space

Edit distance graph.
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).

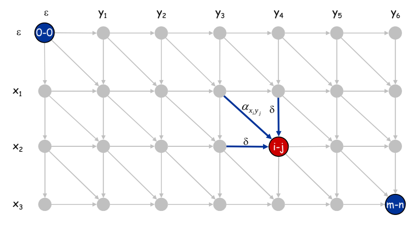


ε
y_1 y_2 y_3 y_4 y_5 y_6
x_1 x_2 x_3
0-0
$\alpha_{x_i y_j}$
δ
i-j
m-n
x
$7\delta + \alpha_{x_3, y_1}$
x_1 x_2 x_3 - - - - -
- - y_1 y_2 y_3 y_4 y_5 y_6

9

Sequence Alignment: Linear Space

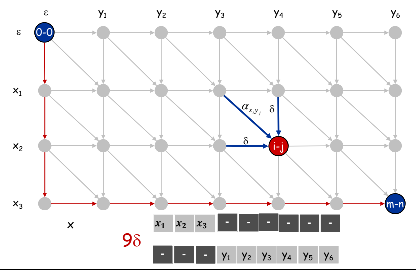
Edit distance graph.
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).



6

Sequence Alignment: Linear Space

Edit distance graph.
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).

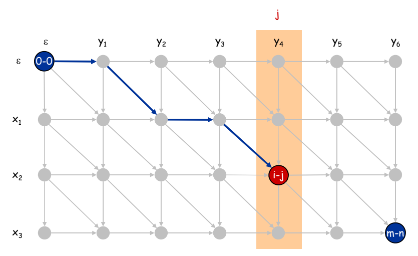


ε
y_1 y_2 y_3 y_4 y_5 y_6
x_1 x_2 x_3
0-0
$\alpha_{x_i y_j}$
δ
i-j
m-n
x
9δ
x_1 x_2 x_3 - - - - - -
- - - y_1 y_2 y_3 y_4 y_5 y_6

8

Sequence Alignment: Linear Space

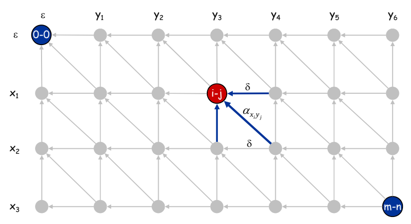
Edit distance graph.
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Can compute f (•, j) for any j in O(mn) time and O(m + n) space.



30

Sequence Alignment: Linear Space

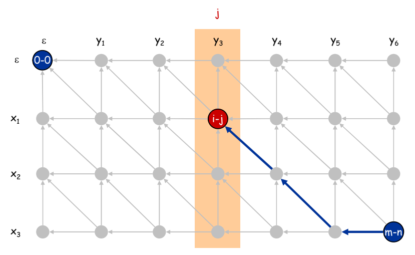
Edit distance graph.
- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute by reversing the edge orientations and inverting the roles of (0, 0) and (m, n)



31

Sequence Alignment: Linear Space

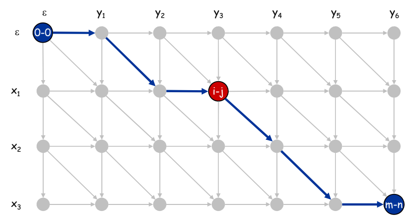
Edit distance graph.
- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute g(•, j) for any j in O(mn) time and O(m + n) space.



32

Sequence Alignment: Linear Space

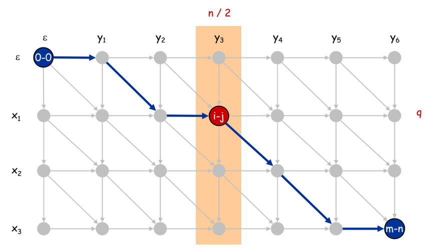
Observation 1. The cost of the shortest path that uses (i, j) is f(i, j) + g(i, j).



33

Sequence Alignment: Linear Space

Observation 2. let q be an index that minimizes f(q, n/2) + g(q, n/2). Then, the shortest path from (0, 0) to (m, n) uses (q, n/2).

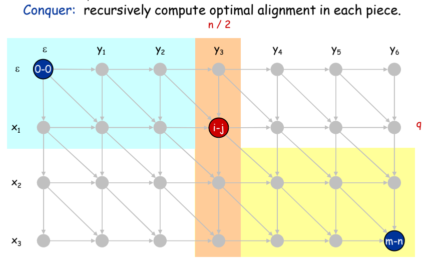


34

Sequence Alignment: Linear Space

Divide: find index q that minimizes f(q, n/2) + g(q, n/2) using DP.
- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.



35

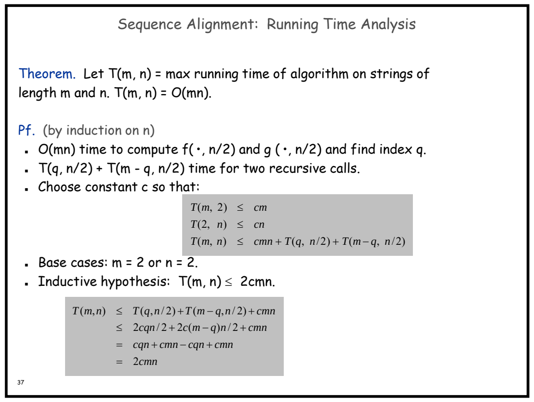
Sequence Alignment: Running Time Analysis Warmup

Theorem. Let T(m, n) = max running time of algorithm on strings of length at most m and n. T(m, n) = O(mn log n).

$$T(m,n) \le 2T(m, n/2) + O(mn) \Rightarrow T(m,n) = O(mn \log n)$$

Remark. Analysis is not tight because two sub-problems are of size (q, n/2) and (m - q, n/2). In next slide, we save log n factor.

36



Sequence Alignment: Running Time Analysis

Theorem. Let T(m, n) = max running time of algorithm on strings of length m and n. T(m, n) = O(mn).

Pf. (by induction on n)

- O(mn) time to compute f(•, n/2) and g (•, n/2) and find index q.
- T(q, n/2) + T(m - q, n/2) time for two recursive calls.
- Choose constant c so that:

$$\begin{aligned} T(m,\ 2) &\le cm \\ T(2,\ n) &\le cn \\ T(m,\ n) &\le cmn + T(q,\ n/2) + T(m-q,\ n/2) \end{aligned}$$

- Base cases: m = 2 or n = 2.
- Inductive hypothesis: T(m, n) ≤ 2cmn.

$$\begin{aligned} T(m,n) &\le T(q,n/2) + T(m-q,n/2) + cmn \\ &\le 2cqn/2 + 2c(m-q)n/2 + cmn \\ &= cqn + cmn - cqn + cmn \\ &= 2cmn \end{aligned}$$

37