The Design and Analyse of Algorithms

Lecture 13 Dynamic Programming II

Zhenbo Wang

Department of Mathematical Sciences, Tsinghua University

Content

Knapsack Problem

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

Greedy Algorithms

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Table 1: knapsack instance with W = 11

Ex. $\{1, 2, 5\}$ has value 35; $\{3, 4\}$ has value 40.

Ex. $\{3, 5\}$ has value 46 (but exceeds weight limit).

- *Greedy by value*. Repeatedly add item with maximum v_i.
- *Greedy by weight*. Repeatedly add item with minimum w_i.
- *Greedy by ratio*. Repeatedly add item with maximum ratio v_i / w_i.
- *Observation*. None of greedy algorithms is optimal.

Dynamic Programming

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

Case 1. OPT does not select item i.

OPT selects best of $\{1, 2, \cdots, i-1\}$ using weight limit w.

Case 2. OPT selects item i.

OPT selects best of $\{1, 2, \cdots, i-1\}$ 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

$KNAPSACK(n, W, w_1, \cdots, w_n, v_1, \cdots, v_n)$

```
 1: for w = 0 to W do
 2:     M[0, w] ← 0.
 3: end for
 4: for i = 1 to n do
 5:     for w = 1 to W do
 6:         if w_i > w then
 7:             M[i, w] ← M[i − 1, w].
 8:         else
 9:             M[i, w] ← max{M[i − 1, w], v_i + M[i − 1, w − w_i]}.
10:         end if
11:     end for
12: end for
13: return M[n, W].
```

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

Knapsack Problem: Running Time

Theorem 1

There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

Pf. Takes O(1) time per table entry.

There are $\Theta(nW)$ table entries.

After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] > M[i-1, w]$. □

Remarks. Not polynomial in input size!

Decision version of knapsack problem is *NP-COMPLETE*. [CHAPTER 8]

There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]

RNA Secondary Structure

- *RNA*. String $B = b_1 b_2 \cdots b_n$ 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.

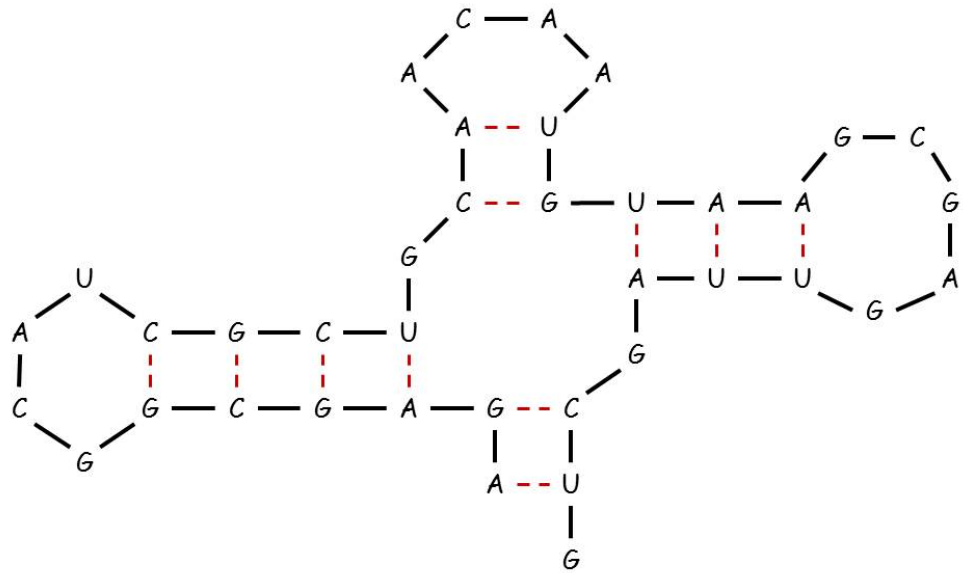


Figure 1: RNA secondary structure for *GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA*

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 $G - C$.

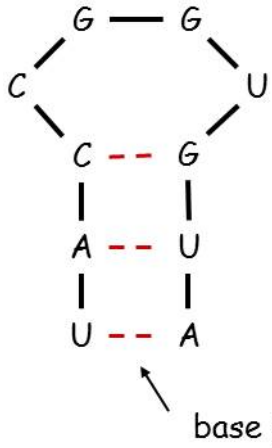
  [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.

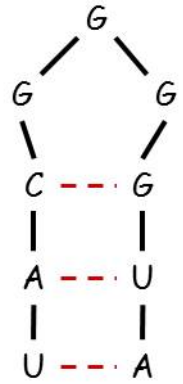
  [Non-crossing] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.

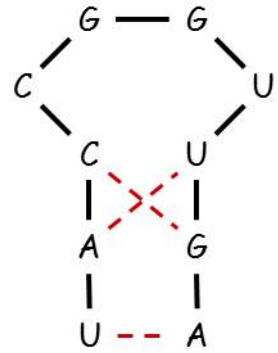
- *Free energy*. Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy (approximate by number of base pairs).

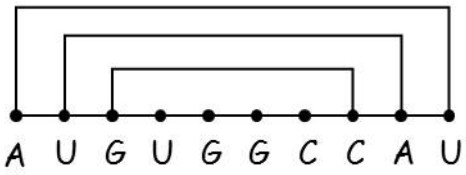
Goal. Given an RNA molecule $B = b_1 b_2 \cdots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

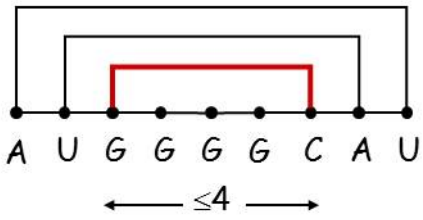




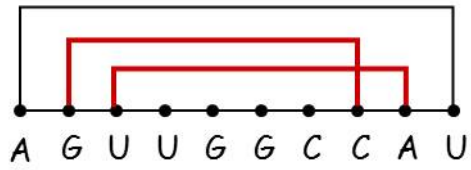




ok



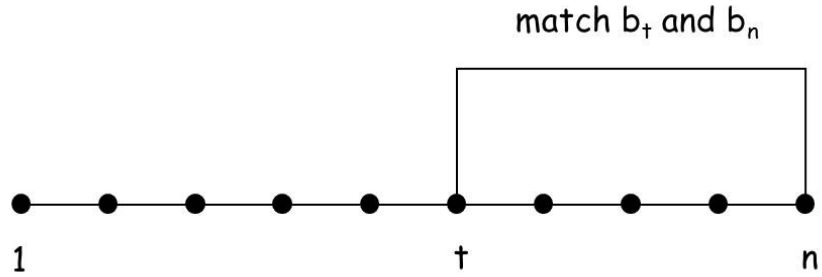
sharp turn



crossing

RNA Secondary Structure: Subproblems

- *First attempt*. $OPT(j) =$ maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \cdots b_j$.
- *Choice*. Match b_t and b_n.



- *Difficulty*. Results in two subproblems but one of wrong form.

  Find secondary structure in $b_1 b_2 \cdots b_{t-1}$. $\leftarrow OPT(t-1)$

  Find secondary structure in $b_{t+1} b_{t+2} \cdots b_{n-1}$. $\leftarrow$ need more subproblems.

Dynamic Programming Over Intervals

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

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 $i \leq t < j - 4$.

Noncrossing constraint decouples resulting subproblems.

$OPT(i, j) = 1 + \max_t \{OPT(i, t - 1) + OPT(t + 1, j - 1)\}$.

Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do shortest intervals first.

$RNA(n, b_1, \cdots, b_n)$

```
1: for k = 5 to n – 1 do
2:     for i = 1 to n – k do
3:         j ← i + k.
4:         Compute M[i, j] using formula.
5:     end for
6: end for
7: return M[1, n].
```

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

Dynamic Programming Summary

- *Outline*.

  Polynomial number of subproblems.

  Solution to original problem can be computed from subproblems.

  Natural ordering of subproblems from smallest to largest.

- *Techniques*.

  Binary choice: weighted interval scheduling.

  Multiway choice: segmented least squares.

  Adding a new variable: knapsack problem.

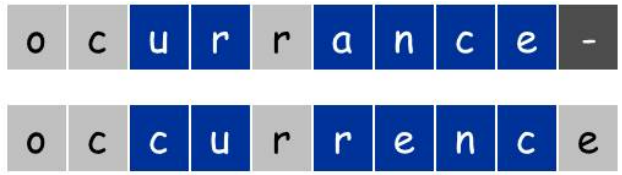
  Dynamic programming over intervals: RNA secondary structure.

- *Top-down vs. bottom-up*. Different people have different intuitions.

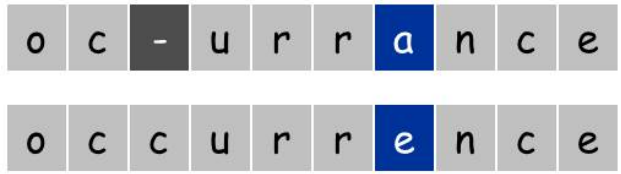
String Similarity

Q. How similar are two strings?

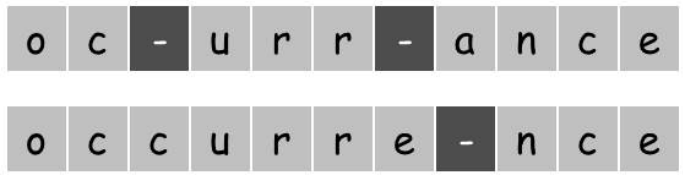
Ex. Ocurrance and occurrence.



6 mismatches, 1 gap



1 mismatch, 1 gap



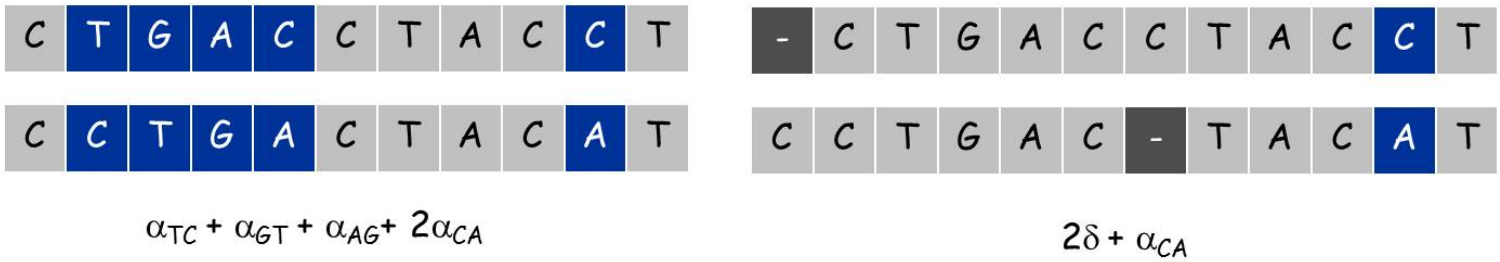
0 mismatches, 3 gaps

Edit Distance

- *Edit distance*. [Levenshtein 1966, Needleman-Wunsch 1970]

  Gap penalty δ; mismatch penalty α_{pq}.

  Cost $=$ sum of gap and mismatch penalties.



- *Applications*: speech recognition; computational biology.

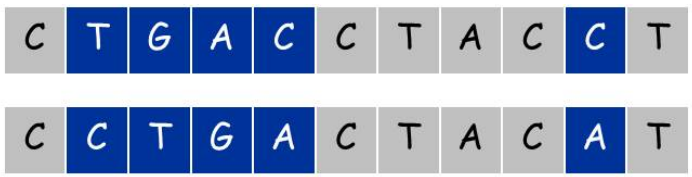
Sequence Alignment

Goal. Given two strings $x_1 x_2 \cdots x_m$ and $y_1 y_2 \cdots y_n$ find min cost alignment.

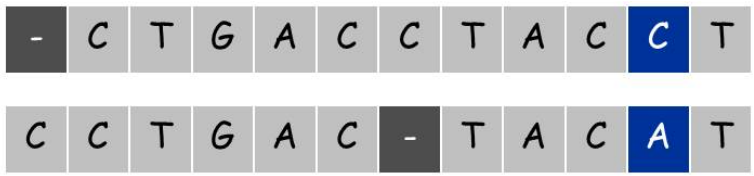
Def. An alignment M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

Def. The cost of an alignment M is:

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



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



$2\delta + \alpha_{CA}$

Sequence Alignment: Problem Structure

Def. $OPT(i, j) =$ min cost of aligning strings $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$.

Case 1. OPT matches $x_i - y_j$.

Pay mismatch for $x_i - y_j +$ min cost of aligning $x_1 x_2 \cdots x_{i-1}$ and $y_1 y_2 \cdots y_{j-1}$.

Case 2a. OPT leaves x_i unmatched.

Pay mismatch for $x_i +$ min cost of aligning $x_1 x_2 \cdots x_{i-1}$ and $y_1 y_2 \cdots y_j$.

Case 2b. OPT leaves y_j unmatched.

Pay mismatch for $y_j +$ min cost of aligning $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_{j-1}$.

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

Sequence Alignment: Algorithm

SEQUENCE – ALIGNMENT$(m, n, x_1, \cdots, x_m, y_1, \cdots, y_n, \delta, \alpha)$

```
1: for i = 0 to m do
2:     M[i, 0] ← iδ.
3: end for
4: for j = 0 to m do
5:     M[0, j] ← jδ.
6: end for
7: for i = 1 to m do
8:     for j = 1 to n do
9:         M[i, j] ← min{α[x_i, y_j] + M[i − 1, j − 1], δ + M[i − 1, j], δ +
           M[i, j − 1]}.
10:    end for
11: end for
12: return M[m, n].
```

Sequence Alignment: Analysis

Theorem 2

The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length m and n in $\Theta(mn)$ time and $\Theta(mn)$ space.

Pf. Algorithm computes edit distance.

Can trace back to extract optimal alignment itself. □

Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

A. Easy to compute optimal value in $O(mn)$ time and $O(m + n)$ space.

Compute $OPT(i, \cdot)$ from $OPT(i - 1, \cdot)$.

But, no longer easy to recover optimal alignment itself.

Theorem 3 (Hirschberg 1975)

There exist an algorithm to find an optimal alignment in $O(mn)$ time and $O(m + n)$ space.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Homework

- Read Chapter 6 of the textbook.
- Exercises 20 & 27 in Chapter 6.