EE360C: Algorithms

Dynamic Programming (2/4)

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Segmented Least Squares

Segmented Least Squares

Least Squares

- Foundational problem in statistic and numerical analysis
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

Solution

The solution from calculus tells us that the minimum error is achieved when:

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}$$

and

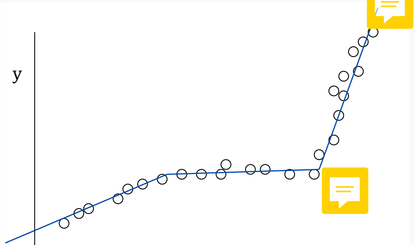
$$b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Points lie roughly on a sequence of several line segments
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, find a sequence of lines that minimizes $f(x)$

Question

What is a reasonable choice for $f(x)$ to balance accuracy (goodness of fit) and parsimony (number of lines)



Segmented Least Squares

The Tradeoff

We have to tradeoff the number of lines for the summed error

Find a sequence of lines that minimizes

- The sum of the sums of the squared errors E in each segment
- The number of lines L

This results in a tradeoff function $E + cL$ for some constant c

Dynamic Programming: Multiway Choice

Notation

- $OPT(j) =$ minimum cost for points $p_1, p_2, \ldots, p_j$
- $e(i, j) =$ minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$

To Compute $OPT(j)$

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i
- Cost = $e(i, j) + c + OPT(i-1)$

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \le i \le j}(e(i, j) + c + OPT(i-1)) & \textit{otherwise} \end{cases}$$

Segmented Least Squares: Algorithm

```
INPUT: n, p₁,…,p_N , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error e_ij for
         the segment p_i,…, p_j

   for j = 1 to n
      M[j] = min_{1 ≤ i ≤ j} (e_ij + c + M[i-1])

   return M[n]
}
```

Running Time

- Bottleneck: computing $e(i, j)$ for $O(n^2)$ pairs, $O(n)$ per pair using the previous formula

Knapsack

The 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 a capacity of W kilograms
- Goal: fill the knapsack so as to maximize the total value

Greedy?

Remember the greedy algorithm we explored was not optimal...

Knapsack: A False Start

Definition: $OPT(i)$ is the max profit for the subset of items $1, \ldots, i$

- Case 1: *OPT* does not select item i
  - *OPT* selects the best of $1, 2, \ldots, i - 1$
- 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

We need more subproblems!

Dynamic Programming: Adding a Variable

Definition: $OPT(i, w)$ is the max profit of items $1, \ldots, i$ with weight limit w

- Case 1: OPT does not select item i
  - OPT selects best of $\{1, 2, \ldots, i-1\}$ using weight limit w
- Case 2: OPT selects item i
  - new weight limit $w - w_i$
  - OPT selects best of $\{1, 2, \ldots, i-1\}$ 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: Bottom Up

```
Input: n, w1,…,wN, v1,…,vN

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

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

return M[n, W]
```

Running Time

- $\theta(nW)$
- Is this a polynomial-time algorithm?
- Nope. It's "pseudo-polynomial"

RNA Secondary Structure

RNA Secondary Structure

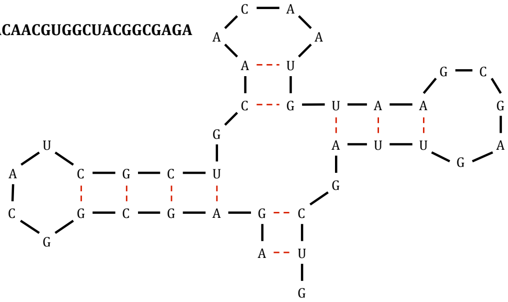
RNA

String $B = b_1 b_2 \ldots b_n$ over the 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 often essential to the function of the molecule. Legal base pairs are (A, U) or (C, G).

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



RNA Secondary Structure

Secondary Structure

A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- **Watson and Crick:** S is a matching and each pair in S is a Watson-Crick component (matching A to U or C to G)
- **No Sharp Turns:** The ends of each pair are separated by at least 4 intervening bases. That is, 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, the we cannot have $i < k < j < l$

Free Energy

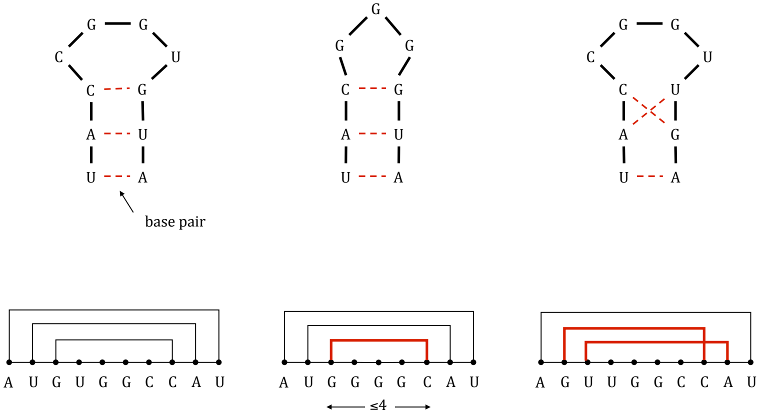
The usual hypothesis is that an RNA molecule will form the secondary structure that has the optimum total *free energy*, which we approximate by the number of base pairs.

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.

RNA Secondary Structure: Examples

Examples.



ok | sharp turn | crossing

Dynamic Programming Over Intervals

Notation

$OPT(i, j)$ is the 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 the 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 $i \leq t < j - 4$
  - The non crossing constraint decouples the resulting subproblems
  - $OPT(i.j) = 1 + \max_t(OPT(i, t - 1) + OPT(t + 1, j - 1))$ such that $i \leq t < j - 4$ and (b_j, b_t) is a Watson and Crick pair

Bottom Up Dynamic Programming Over Intervals

Question

What order to solve the subproblems?

Answer

Do the 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
}
```

Dynamic Programming Summary

Recipe

- Characterize the structure of the problem
- Recursively define the value of an optimal solution
- Compute the value of the optimal solution
- Construct the 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

Top-down vs. Bottom-up?

Different people have different intuitions

Questions