Dynamic Programming*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course: Shanghai Jiao Tong University

* Special thanks is given to *Prof. Kevin Wayne@Princeton* for sharing his slides, and also given to Mr. Chao Wang from CS2014@SJTU and Mr. Hongjian Cao from CS2015@SJTU for producing this lecture.

Outline

1. Introduction
   - Introduction
2. Basic Methodology
   - Weighted Interval Scheduling
   - Segmented Least Squares
   - Knapsack Problem
3. More Examples
   - RNA Secondary Structure

Outline

1. Introduction
   - Introduction
2. Basic Methodology
   - Weighted Interval Scheduling
   - Segmented Least Squares
   - Knapsack Problem
3. More Examples
   - RNA Secondary Structure

Algorithmic Paradigms

Greedy: Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer: Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming Applications

Areas

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ...

Some famous dynamic programming algorithms

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

Outline

Weighted Interval Scheduling Problem

Job j starts at s_j, finishes at f_j, and has weight or value v_j .

Two jobs compatible if they don't overlap.

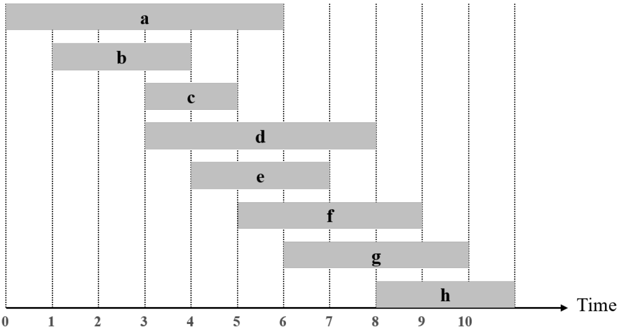
Goal: find maximum weight subset of mutually compatible jobs.

Weighted Interval Scheduling Problem

Job j starts at s_j, finishes at f_j, and has weight or value v_j .

Two jobs compatible if they don't overlap.

Goal: find maximum weight subset of mutually compatible jobs.



Unweighted Interval Scheduling Review

Recall: Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Unweighted Interval Scheduling Review

Recall: Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation: Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Weighted Interval Scheduling

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

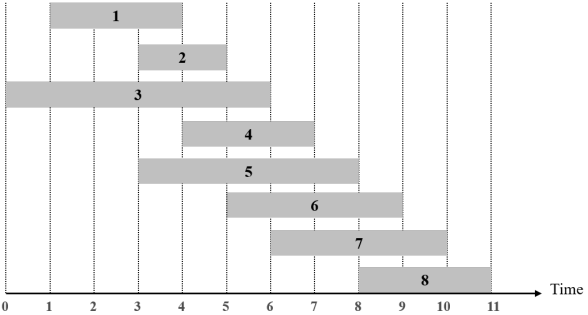
Definition: $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Weighted Interval Scheduling

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Definition: $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Example: $p(8) = 5, p(7) = 3, p(2) = 0$.



Binary Choice

Greedy template: $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \cdots, j$.

Optimal substructure:

Case 1: OPT selects job j.

- collect profit v_j
- can't use incompatible jobs $\{p(j) + 1, p(j) + 2, \cdots, j - 1\}$
- must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \cdots, p(j)$

Case 2: OPT does not select job j.

- must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \cdots, j - 1$

Binary Choice

Greedy template: $OPT(j)$ = value of optimal solution to the problem consisting of job requests $1, 2, \cdots, j$.

Optimal substructure:

Case 1: OPT selects job j.

- collect profit v_j
- can't use incompatible jobs $\{p(j)+1, p(j)+2, \cdots, j-1\}$
- must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \cdots, p(j)$

Case 2: OPT does not select job j.

- must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \cdots, j-1$

$$OPT(j) = \begin{cases} 0, & j = 0, \\ \max\{v_j + OPT(p(j)), OPT(j-1)\}, & otherwise \end{cases}$$

Brute Force Algorithm

Algorithm 1: Brute Force

Input: n; $s_1, \cdots, s_n$; $f_1, \cdots, f_n$; $v_1, \cdots, v_n$;

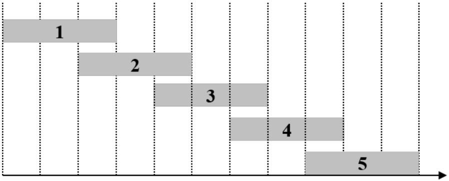
Sort jobs by finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p(1), p(2), \cdots, p(n)$;

Function `Compute-Opt` (j):
    if j=0 **then**
        return 0;
    else
        return
            $\max\{v_j+$ Compute-Opt$(p(j))$, Compute-Opt$(j-1)\}$;

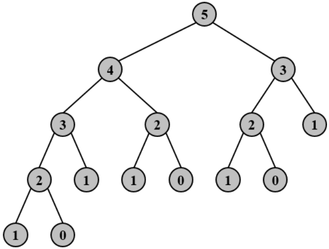
Brute Force Algorithm

Observation: Recursive algorithm fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

Example: Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Memoization

Store results of each sub-problem in a cache; lookup as needed.

Algorithm 2: Memoization

Input: $n; s_1, \cdots, s_n; f_1, \cdots, f_n; v_1, \cdots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p(1), p(2), \cdots, p(n)$;
for $j = 1 \rightarrow n$ **do**
    $M[j]$ = empty;
$M[0] = 0$;

Function `M-Compute-Opt` (j) **:**
    if *M[j] is empty* **then**
        $M[j] =$
            $\max\{v_j +$ M-Compute-Opt$(p(j))$, M-Compute-Opt$(j-1)\}$;
    return $M[j]$;

Running Time

Claim: Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.
- M-Compute-Opt(j): each invocation takes $O(1)$ time and either
  - returns an existing value $M[j]$
  - fills in one new entry $M[j]$ and makes two recursive calls
- Progress measure Φ = number nonempty entries of $M[\cdot]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - increases Φ by 1 $\Rightarrow$ at most $2n$ recursive calls.
- Overall running time of M-Compute-Opt(n) is $O(n)$.

Remark: $O(n)$ if jobs are pre-sorted by start and finish times.

Finding a Solution from the OPT Value

Algorithm 3: Post-Processing

Run M-Compute-Opt(n);
Run Find-Solution(n);
Function `Find-Solution` (j) **:**
  if $j = 0$ **then**
    output nothing;
  else if $v_j + M[p(j)] > M[j-1]$ **then**
    print j;
    Find-Solution ($p(j)$);
  else
    Find-Solution($j-1$);

- $\#$ of recursive calls $1 \leq n \Rightarrow O(n)$;

Bottom-Up Dynamic Programming

Algorithm 4: Memoization

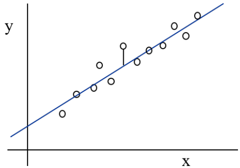
Input: n; $s_1, \cdots, s_n$; $f_1, \cdots, f_n$; $v_1, \cdots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p(1), p(2), \cdots, p(n)$;
Function `Iterative-Compute-Opt()`**:**
  M[0] = 0;
  for $j = 1 \to n$ **do**
    $M[j] = \max\{v_j + M[p(j)], M[j-1]\}$;

Outline

Segmented Least Squares

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



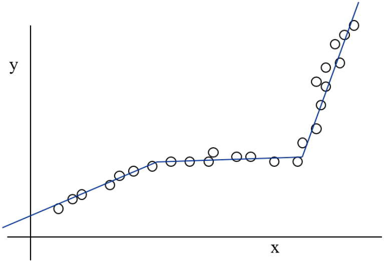
Solution: Calculus $\Rightarrow$ min error is achieved when

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

Segmented Least Squares

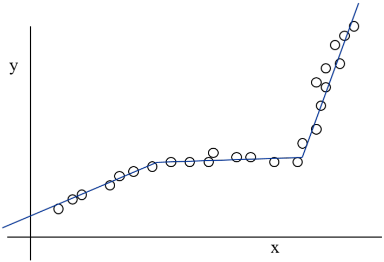
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ with $x_1 < x_2 < \cdots < x_n$, find a sequence of lines that minimizes $f(x)$.

Question: What's a reasonable choice for $f(x)$ to balance accuracy (goodness of fit) and parsimony (number of lines)?



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) , $\cdots$, (x_n, y_n) with $x_1 < x_2 < \cdots < x_n$, 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
- Tradeoff function: $E + cL$, for some constant $c > 0$.



Multiway Choice

Notation:

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

Compute $OPT(j)$:

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

$$OPT(j) = \begin{cases} 0, & j = 0, \\ \min\limits_{1 \le i \le j} \{e(i,j) + c + OPT(i-1)\}, & otherwise \end{cases}$$

Segmented Least Squares

Algorithm 5: Memoization

Input: $n; p_1, \cdots, p_N; c$

```
Function Iterative-Compute-Opt():
    M[0] = 0;
    for j = 1 → n do
        for i = 1 → j do
            compute the least square error e_ij for the segment
              p_i, ..., p_j;
    for j = 1 → n do
        M[j] = min_{1≤i≤j}{e_ij + c + M[i − 1]};
    return M[n];
```

Segmented Least Squares

Algorithm 6: Memoization

Input: $n; p_1, \cdots, p_N; c$

Function `Iterative-Compute-Opt()`**:**

  $M[0] = 0$;

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

    for $i = 1 \to j$ **do**

      compute the least square error e_{ij} for the segment $p_i, \cdots, p_j$;

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

    $M[j] = \min_{1 \leq i \leq j}\{e_{ij} + c + M[i-1]\}$;

  return $M[n]$;

Running time: $O(n^3)$ (can be improved to $O(n^2)$ by pre-computing.)

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

Outline

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.

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.

Example: $\{3, 4\}$ has value 40.

W = 11

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

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.

Example: $\{3, 4\}$ has value 40.

W = 11

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

Greedy: repeatedly add item with maximum ratio v_i/w_i.

Example: $\{5, 2, 1\}$ achieves only value = 35 $\Rightarrow$ greedy not optimal.

False Start

Definiton:$OPT(i)$= max profit subset of items $1, \cdots, i$.

Case 1: OPT does not select item i.

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

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!

Adding a New Variable

Definiton: $OPT(i)$ = 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.

- new weight $limit = w - w_i$
- OPT selects best of using $\{1, 2, \cdots, i-1\}$ this new weight limit

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

Bottom-Up Algorithm (Fill up an n-by-W array)

Algorithm 7: Knapsack Problem Algorithm using n-by-W Array

Input: $n, W, w_1, \cdots, w_N, v_1, \cdots, v_N$

for $w = 0 \to W$ **do**
    $M[0, w] = 0;$
for $i = 1 \to n$ **do**
    for $w = 1 \to W$ **do**
        if $w_i > w$ **then**
            $M[i, w] = M[i - 1, w];$
        else
            $M[i, w] = \max\{M[i - 1, w], v_i + M[i - 1, w - w_i]\}$
return $M[n, W];$

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

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

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

W = 11

OPT: $\{4, 3\}$

value $= 22 + 18 = 40$

Running Time

Running time: $\Theta(nW)$.

- Not polynomial in input size!
- "Pseudo-polynomial".
- Decision version of Knapsack is NP-complete.

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

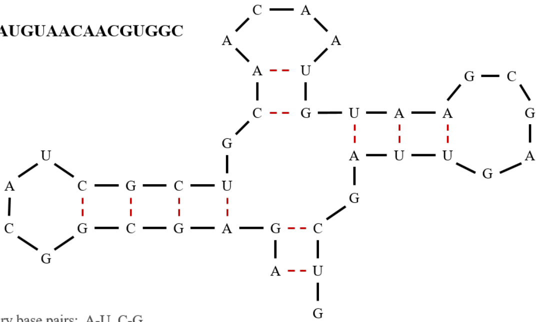
Outline

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.

Example: **GUCGAUUGAGCGAAUGUAACAACGUGGC UACGGCGAGA**



complementary base pairs: A-U, C-G

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.

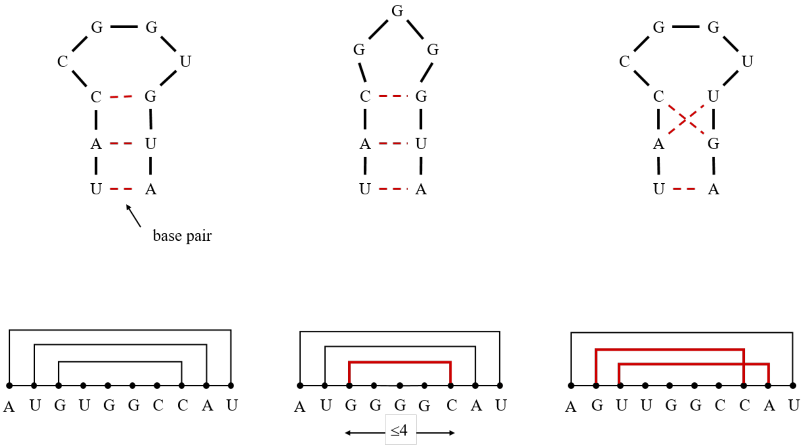
[No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.

[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.

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

Examples



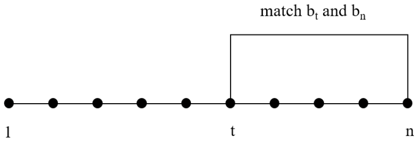
ok

sharp turn

crossing

Subproblems

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



Difficulty: Results in two sub-problems.

- Finding secondary structure in: $b_1 b_2 \cdots b_{t-1}$.
- Finding secondary structure in: $b_{t+1} b_{t+2} \cdots b_{n-1}$.

Dynamic Programming Over Intervals

Notation: $OPT(j)$ = 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 $i \leq t < j - 4$.

- non-crossing constraint decouples resulting sub-problems
- $OPT(i,j) = 1 + \max_t\{OPT(i,t-1) + OPT(t+1,j-1)\}$

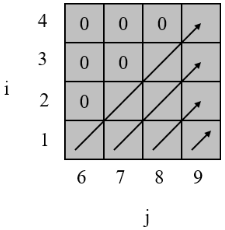
Remark: Same core idea in CKY algorithm to parse context-free grammars.

Bottom Up Dynamic Programming Over Intervals

Question: What order to solve the sub-problems?

Answer: Do shortest intervals first.

```
RNA(b1,…,bn) {
  for k = 5, 6, …, n-1
    for i = 1, 2, …, n-k
      j = i + k
      Compute M[i, j]
                          using recurrence
  return M[1, n]
}
```



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

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 interval

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