6. Dynamic Programming I

- weighted interval scheduling
- segmented least squares
- knapsack problem
- RNA secondary structure

Algorithmic paradigms

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

Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

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.
- De Boor for evaluating spline curves.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.
- ...

---

Lecture slides by Kevin Wayne
Copyright © 2003 Pearson-Addison Wesley
Copyright © 2013 Kevin Wayne
http://www.cs.princeton.edu/~wayne/algorithm-design
6. Dynamic Programming I

- weighted interval scheduling
- segmented least squares
- knapsack problem
- RNA secondary structure

**Earliest-finish-time first algorithm**

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

**Recall.** Greedy algorithm is correct if all weights are 1.

**Observation.** Greedy algorithm fails spectacularly for weighted version.

**Weighted interval scheduling**

**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.

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

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

**Ex.** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$. 
Dynamic programming: binary choice

Notation. \( OPT(j) \) = value of optimal solution to the problem consisting of job requests \( 1, 2, \ldots, j \).

Case 1. \( OPT \) selects job \( j \).
- Collect profit \( v_j \).
- Can't use incompatible jobs \( \{ p(j) + 1, p(j) + 2, \ldots, j - 1 \} \).
- Must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, p(j) \).

Case 2. \( OPT \) does not select job \( j \).
- Must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, j - 1 \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left\{ v_j + OPT(p(j)), OPT(j - 1) \right\} & \text{otherwise}
\end{cases}
\]

Weighted interval scheduling: brute force

Input: \( n, s[1..n], f[1..n], v[1..n] \)
Sort jobs by finish time so that \( f[1] \leq f[2] \leq \ldots \leq f[n] \).
Compute \( p[1], p[2], \ldots, p[n] \).

\[
\text{Compute-Opt}(j)
\begin{align*}
&\text{if } j = 0 \\
&\quad \text{return } 0. \\
&\text{else} \\
&\quad \text{return } \max(v[j] + \text{Compute-Opt}(p[j]), \text{Compute-Opt}(j-1)).
\end{align*}
\]

Weighted interval scheduling: memoization

Memoization. Cache results of each subproblem; lookup as needed.

Input: \( n, s[1..n], f[1..n], v[1..n] \)
Sort jobs by finish time so that \( f[1] \leq f[2] \leq \ldots \leq f[n] \).
Compute \( p[1], p[2], \ldots, p[n] \).

for \( j = 1 \) to \( n \)
\[
M[j] \leftarrow \text{empty}. \\
M[0] \leftarrow 0.
\]

\[
\text{M-Compute-Opt}(j)
\begin{align*}
&\text{if } M[j] \text{ is empty} \\
&\quad M[j] \leftarrow \max(v[j] + \text{M-Compute-Opt}(p[j]), \text{M-Compute-Opt}(j-1)). \\
&\text{return } M[j].
\end{align*}
\]

Observation. Recursive algorithm fails spectacularly because of redundant subproblems \( \Rightarrow \) exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.
Weighted interval scheduling: running time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.
- 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
- (i) returns an existing value $M[j]$
- (ii) fills in one new entry $M[j]$ and makes two recursive calls

Progress measure $\Phi = \# \text{nonempty entries of } M[\cdot]$.
- Initially $\Phi = 0$, throughout $\Phi \leq n$.
- (ii) increases $\Phi$ by 1 ⇒ at most $2n$ recursive calls.

Overall running time of M-COMPUTE-OPT($n$) is $O(n)$.

Remark. $O(n)$ if jobs are presorted by start and finish times.

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?
A. Make a second pass.

```
Find-Solution(j)
if j = 0
    return \emptyset.
else if (v[j] + M[p[j]] > M[j-1])
    return \{ j \} \cup Find-Solution(p[j]).
else
    return Find-Solution(j-1).
```

Analysis. # of recursive calls $\leq n$ ⇒ $O(n)$.

Weighted interval scheduling: bottom-up

Bottom-up dynamic programming. Unwind recursion.

```
BOTTOM-UP($n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$)
Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$.
$M[0] \leftarrow 0$.
FOR $j = 1$ TO $n$
    $M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \}$.
```

6. Dynamic Programming I

- weighted interval scheduling
- segmented least squares
- knapsack problem
- RNA secondary structure

SECTION 6.3
Least squares

- 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. Calculus $\Rightarrow$ min error is achieved when

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

Segmented least squares

Given $n$ points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$ and a constant $c > 0$, find a sequence of lines that minimizes $f(x) = E + cL$:
- $E =$ the sum of the sums of the squared errors in each segment.
- $L =$ the number of lines.

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 $i$.
- $Cost = e(i,j) + c + OPT(i-1)$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$
Segmented least squares algorithm

\[ \text{SEGMENTED-LEAST-SQUARES} (n, p_1, \ldots, p_n, c) \]

\begin{verbatim}
FOR j = 1 TO n
    FOR i = 1 TO j
        Compute the least squares \( e(i, j) \) for the segment \( p_i, p_{i+1}, \ldots, p_j \).
    \end{verbatim}

\begin{verbatim}
M[0] \leftarrow 0.

FOR j = 1 TO n
    M[j] \leftarrow \min_{1 \leq i \leq j} \{ e(i, j) + c + M[i-1] \}.
\end{verbatim}

RETURN \( M[n] \).

Segmented least squares analysis

**Theorem.** [Bellman 1961] The dynamic programming algorithm solves the segmented least squares problem in \( O(n^3) \) time and \( O(n^2) \) space.

**Pf.**

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

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

**Remark.** Can be improved to \( O(n^2) \) time and \( O(n) \) space by precomputing various statistics. How?

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.

\[
\begin{array}{ccc}
  i & v_i & w_i \\
  1 & 1 & 1 \\
  2 & 6 & 2 \\
  3 & 18 & 5 \\
  4 & 22 & 6 \\
  5 & 28 & 7 \\
\end{array}
\]

\( \text{knapsack instance (weight limit W = 11)} \)

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: false start

Def. \( OPT(i) = \max \text{ profit subset of items } 1, \ldots, i. \)

Case 1. \( OPT \) does not select item \( i. \)
- \( OPT \) selects best of \( \{ 1, 2, \ldots, i-1 \}. \)

Case 2. \( OPT \) selects item \( i. \)
- Selecting 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 subproblems!

Knapsack problem: bottom-up

**Knapsack** \((n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)\)

**FOR** \( w = 0 \) \textbf{TO} \( W \)
\( M[0, w] \leftarrow 0. \)

**FOR** \( i = 1 \) \textbf{TO} \( n \)
**FOR** \( w = 1 \) \textbf{TO} \( W \)
\( \text{IF } (w_i > w) \text{ } M[i, w] \leftarrow M[i-1, w]. \)
\( \text{ELSE } M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}. \)

**RETURN** \( M[n, W]. \)

Dynamic programming: adding a new variable

Def. \( OPT(i, w) = \max \text{ profit subset of items } 1, \ldots, i \) 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.

\[
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w_i > w \\
\max \{ OPT(i-1, w), v_i + OPT(i-1, w-w_i) \} & \text{otherwise}
\end{cases}
\]
Theorem. 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! \( \text{"pseudo-polynomial"} \)
- Decision version of knapsack problem is \text{NP}-\text{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 \( B = b_1b_2...b_n \) 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.

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy.

Goal. Given an RNA molecule \( B = b_1b_2...b_n \), find a secondary structure \( S \) that maximizes the number of base pairs.
RNA secondary structure

**Examples.**

![Diagrams of RNA secondary structures with examples of base pairs, sharp turns, and crossings.]

- **Base pair:**
  - G—G
  - C—C
  - A—U

- **Sharp turn:** (≤ 4 intervening bases)
  - G—G

- **Crossing:**
  - G—C
  - C—U

**Dynamic programming over intervals**

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

**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 $b_i$ 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.** In which order to solve the subproblems?
**A.** Do shortest intervals first.

**RNA** $(n, b_1, \ldots, b_n)$

**FOR** $k = 5 \text{ TO } n - 1$

**FOR** $i = 1 \text{ TO } n - k$

\[ j \leftarrow i + k. \]

**Compute $M[i, j]$ using formula.**

**RETURN** $M[1, n]$.

**Theorem.** The dynamic programming algorithm solves the RNA secondary substructure problem in $O(n^3)$ time and $O(n^2)$ space.
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, with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solution to smaller subproblems.

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.