Dynamic Programming

CMSC 451, Summer 2009

Homework

• Homework 2: 5.3, 5.5, 6.1, 6.24
• Due Monday by 9:40am

• Reminder: Write-up of your group project algorithm also due Monday by 9:40am.

• Use the forum for questions to me that others would also benefit from
## 6 Dynamic Programming
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.

sub-problems whose results can be reused several times

---

Dynamic Programming History

**Bellman.** [1950s] Pioneered the systematic study of dynamic programming.

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

"it’s impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

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.

6.1 Weighted Interval Scheduling
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.

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: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

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

**Ex:** \( p(8) = 5, p(7) = 3, p(2) = 0 \).

What are the other \( p \) values?

**Designing an Optimal Solution**

**Proof of optimality.**

If we can design an algorithm that searches through all possible solutions (the entire solution space) and finds the maximum, then it is optimal...
Designing an Optimal Solution

Either the optimal solution contains the last job, or it doesn’t. How can we use this to find an optimal solution? Can we create a recursive description of an algorithm using this idea?

Dynamic Programming: Binary Choice

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

- **Case 1:** $OPT$ selects job $j$.
  - collect profit $v_j$
  - can’t use incompatible jobs \{ $p(j) + 1, p(j) + 2, ..., j - 1$ \}
  - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$

- **Case 2:** $OPT$ does not select job $j$.
  - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., j-1$

$$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left\{ v_j + OPT(p(j)), \ OPT(j-1) \right\} & \text{otherwise}
\end{cases}$$
Weighted Interval Scheduling: Brute Force

Brute force algorithm.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

**Compute-Opt(j)** {
  if \( j = 0 \)
    return 0
  else
    return \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems \( \Rightarrow \) exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances, grows like the Fibonacci sequence.
Weighted Interval Scheduling: Memoization

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

**Input:** \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

**Sort** jobs by finish times so that \( f_1 \leq f_2 \leq \cdots \leq f_n \).

**Compute** \( p(1), p(2), \ldots, p(n) \)

\[
\text{for } j = 1 \text{ to } n \\
\text{ } M[j] = \text{empty} \\
\text{ } M[0] = 0 \\
\text{ } M\text{-Compute-Opt}(j) \\
\quad \text{if (} M[j] \text{ is empty)} \\
\quad \text{ } M[j] = \text{max}(v_j + M\text{-Compute-Opt}(p(j)), M\text{-Compute-Opt}(j-1)) \\
\quad \text{return } M[j] \\
\]

---

Weighted Interval Scheduling: 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.
- **\( M\text{-Compute-Opt}(\cdot) \):** 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 \( \Phi = \# \) nonempty entries of \( M[\cdot] \),**
  - initially \( \Phi = 0 \), throughout \( \Phi \leq n \).
  - (ii) increases \( \Phi \) by 1 \( \Rightarrow \) at most \( 2n \) recursive calls.
- **Overall running time of \( M\text{-Compute-Opt}(n) \) is \( O(n) \).**

**Remark.** \( O(n) \) if jobs are pre-sorted by start and finish times.
Weighted Interval Scheduling: Example

Input: \( n, s_1, \ldots, s_n, \epsilon_1, \ldots, \epsilon_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

for \( j = 1 \) to \( n \)
   \( M[j] = \) empty
   \( M[0] = 0 \)

\( M\)-Compute-Opt\( (j) \) {
   if \( M[j] \) is empty
      \( M[j] = \max(v_j + M\)-Compute-Opt\( (p(j))\), \( M\)-Compute-Opt\( (j-1)\))
   return \( M[j] \)
}

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms compute the optimal value. What if we want the solution itself?

A. Do some post-processing.

Run \( M\)-Compute-Opt\( (n) \)
Run \( \text{Find-Solution}(n) \)

Find-Solution\( (j) \) {
   if \( (j = 0) \)
      output nothing
   else if \( (v_j + M[p(j)] > M[j-1]) \)
      print \( j \)
      Find-Solution\( (p(j)) \)
   else
      Find-Solution\( (j-1) \)
}

- \# of recursive calls \( \leq n \) \( \Rightarrow \) \( O(n) \).
Weighted Interval Scheduling: Bottom-Up

**Bottom-up dynamic programming.** Unwind recursion.

**Input:** \( n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq ... \leq f_n \).

Compute \( p(1), p(2), ..., p(n) \)

Iterative-Compute-Opt {
    \( M[0] = 0 \)
    for \( j = 1 \) to \( n \)
        \( M[j] = \max(v_j + M[p(j)], M[j-1]) \)
}

Weighted Interval Scheduling: Bottom-up Example

**Input:** \( n, s_1, ..., s_n, f_1, ..., f_n, v_1, ..., v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq ... \leq f_n \).

Compute \( p(1), p(2), ..., p(n) \)

Iterative-Compute-Opt {
    \( M[0] = 0 \)
    for \( j = 1 \) to \( n \)
        \( M[j] = \max(v_j + M[p(j)], M[j-1]) \)
}
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.

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**W = 11**

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

Dynamic Programming: False Start

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

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

- **Case 2:** \( \text{OPT} \) 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.** \( OPT(i, w) = \text{max profit subset of items 1, \ldots, i with weight limit} \ w \).

- **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

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

Knapsack Problem: Bottom-Up

**Knapsack.** Fill up an \( n \)-by-\( W \) array.

\[
\begin{align*}
\text{Input: } n, \ W, \ w_1, \ldots, w_n, \ v_1, \ldots, v_n \\
\text{for } w = 0 \text{ to } W & \quad M[0, w] = 0 \\
\text{for } i = 1 \text{ to } n \\
\quad \text{for } w = 1 \text{ to } W & \quad \text{if } (w_i > w) \\
\quad & \quad \quad \quad M[i, w] = M[i-1, w] \\
\quad & \quad \text{else} \\
\quad & \quad \quad \quad M[i, w] = \max\{M[i-1, w], v_i + M[i-1, w-w_i]\} \\
\text{return } M[n, W]
\end{align*}
\]