Dynamic Programming Examples

Imran Rashid

University of Washington

February 27, 2008
1 Weighted Interval Scheduling
Lecture Outline

1. Weighted Interval Scheduling
2. Knapsack Problem
Lecture Outline

1. Weighted Interval Scheduling
2. Knapsack Problem
3. String Similarity
Lecture Outline

1. Weighted Interval Scheduling
2. Knapsack Problem
3. String Similarity
4. Common Errors with Dynamic Programming
Algorithmic Paradigms

- **Greed.** Build up a solution incrementally, myopically optimizing some local criterion.

- **Divide-and-conquer.** Break up a problem into two 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, systems, ...

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

1. Weighted Interval Scheduling
2. Knapsack Problem
3. String Similarity
4. Common Errors with Dynamic Programming
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.
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.

Can Greedy work when there are weights?
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.

Can Greedy work when there are weights?
- Greedy fails for ordering either by finish time or by weight.
Weighted Interval Scheduling

- Notation. Label jobs by finishing time: $f_1, f_2, \ldots f_n$.
- Def. $p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.
- Ex: $p(8) = 5, p(7) = 3, p(2) = 0$.
Dynamic Programming: Binary Choice

- Notation. \( \text{OPT}(j) = \) value of optimal solution to the problem consisting of job requests 1, 2, ..., \( j \).
  - Case 1: OPT selects job \( j \).
    - 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, ..., \( j - 1 \)
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 \( f_1 \leq f_2 \leq \ldots \leq f_n \)
Compute \( p(1), p(2), \ldots, p(n) \)

procedure \text{Compute-OPT}(j)
  
  if \( j = 0 \) then return 0
  
  else
    
    return \( \max(v_j + \text{Compute-OPT}(p(j)), \text{Compute-OPT}(j - 1)) \)
  
  end if

end procedure
Weighted Interval Scheduling: Brute Force

- Observation. Recursive algorithm fails spectacularly because of redundant sub-problems exponential algorithms.
- Ex. Number of recursive calls for family of “layered” instances grows like Fibonacci sequence.

\[ p(1) = 0, \ p(j) = j - 2 \]
Weighted Interval Scheduling: Memoization

Input $n, s_1, \ldots s_n, f_1, \ldots f_n, v_1, \ldots, v_n$
Sort jobs by finish times so $f_1 \leq f_2 \leq \ldots \leq f_n$
Compute $p(1), p(2), \ldots, p(n)$
for $i = 1 \ldots n$ do
  $M[i] \leftarrow$ empty
end for
$M[0] \leftarrow 0$
procedure $M\text{-OPT}(j)$
  if $M[j]$ is empty then
    $M[j] \leftarrow \max(v_j + M\text{-OPT}(p(j)), M\text{-OPT}(j - 1))$
  end if
return $M[j]$
end procedure
Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p()$: $O(n)$ after sorting by start time.
- $\text{M-OPT}(j)$: each invocation takes $O(1)$ time and either
  1. returns an existing value $M[j]$
  2. fills in one new entry $M[j]$ and makes two recursive calls
- Progress Measure: $\Theta$ number of empty cells in $M$
  - $\Theta \leq n$ always
  - max 2 recursive calls at any level $\Rightarrow \leq 2n$ recursive calls total
- $\text{M-OPT}(n)$ is $O(n)$
- Overall, $O(n \log n)$, or $O(n)$ if presorted by start & finish times
Weighted Interval Scheduling: Iterative

- Bottom Up Iteration

---

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

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

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

**procedure** \( \text{ITER-OPT}(j) \)

\[
M[0] \leftarrow 0
\]

for \( i = 1 \ldots n \) do

\[
M[i] \leftarrow \max(v_i + M[p(i)], M[i-1])
\]

end for

return \( M[j] \)

end procedure
Outline

1. Weighted Interval Scheduling
2. Knapsack Problem
3. String Similarity
4. Common Errors with Dynamic Programming
Knapsack Problem

- Given $n$ objects and a knapsack
- Object $i$ has weight $w_i$ and value $v_i$.
- Knapsack has maximum weight $W$
- Goal: fill knapsack to maximize total value

Example Instance

- Knapsack max weight $W = 11$.
- Packing items $\{3, 4\}$ gives total value 40.

Can we use greedy?

Greedy by value/weight ratio is sub-optimal. In the example, it would pack $\{5, 2, 1\}$, which only has value 35.
Def. $\text{OPT}(i) = \text{max value subset of items } 1, \ldots, i$.

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

Conclusion. Need more sub-problems!
Knapsack Subproblems: first try

- Def. $OPT(i) = \text{max value subset of items } 1, \ldots, i$.
  - Case 1: OPT does not select item $i$.
    - OPT selects best of $\{1, 2, \ldots, i-1\}$
  - Case 2: OPT selects item $i$.
    - 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!
Knapsack Subproblems: second try

- Def. \( OPT(i, S) = \max \text{ value subset of items } 1, \ldots, i, \text{ using items in the set } S. \)
- Works, but ...
Def. $\text{OPT}(i, S) = \text{max value subset of items } 1, \ldots, i,$ using items in the set $S$.

Works, but ...

... $2^n$ subproblems! We haven’t saved any work
Def. $OPT(i, S) = \text{max value subset of items } 1, \ldots, i,$ using items in the set $S$.

Works, but ...

... $2^n$ subproblems! we haven’t saved any work

Do we really need to know all of items chosen? Just need to know if we can stick in item $i$ ...
Knapsack Subproblems: third time’s a charm

- Only need to know the weight already in the knapsack
- Def. $OPT(i, w) = \max$ value subset of items 1, \ldots, $i$ weighing no more than $w$.
  - Case 1: OPT does not select item $i$.
    - OPT selects best of $\{1, 2, \ldots, i-1\}$ weighing no more than $w$.
  - Case 2: OPT selects item $i$.
    - $w' = w - w_i$
    - OPT adds item $i$ to optimal solution from 1, \ldots, $i-1$ weighing no more than $w'$, the new weight limit.

- The Recurrence:

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

1. Weighted Interval Scheduling
2. Knapsack Problem
3. String Similarity
4. Common Errors with Dynamic Programming
String Similarity

How similar are two strings?

1. occurrence
2. occurrence

- 5 mismatches, 1 gap
- 1 mismatch, 1 gap
- 0 mismatches, 3 gaps
String Edit Distance

- Applications
  - Basis for “diff”
  - Speech Recognition
  - Computational Biology

- Edit Distance
  - Gap Penalty $\delta$; mismatch-penalty $\alpha_{pq}$
  - Cost = sum of gap and mismatch penalties

\[
\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA} \\
2\delta + \alpha_{CA}
\]
**Goal** Given two strings $X = x_1x_2\ldots x_m$ and $Y = y_1y_2\ldots y_n$ find alignment of minimum cost.

**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 pair $(x_i, y_j)$ and $(x_{i'}, y_{j'})$ cross if $i < i'$ but $j > j'$.

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

- mismatch
- gap
Sequence Alignment Subproblems

- **Def** $OPT(i, j) = \text{min cost of aligning strings } x_1x_2\ldots x_i \text{ and } y_1y_2\ldots y_j$. 
  - Case 1. $OPT$ matches $(x_i, y_j)$. Pay mismatch for $(x_i, y_j)$ + min cost aligning substrings $x_1x_2\ldots x_{i-1}$ and $y_1y_2\ldots y_{j-1}$.
  - Case 2a. $OPT$ leaves $x_i$ unmatched. Pay gap for $x_i$ and min cost of aligning $x_1x_2\ldots x_{i-1}$ and $y_1y_2\ldots y_j$.
  - Case 2b. $OPT$ leaves $y_i$ unmatched. Pay gap for $y_i$ and min cost of aligning $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_{j-1}$. 
Sequence Alignment Subproblems

- **Def** $OPT(i, j) = \min \text{ cost of aligning strings } x_1x_2 \ldots x_i \text{ and } y_1y_2 \ldots y_j$.

- **Case 1.** $OPT$ matches $(x_i, y_j)$. Pay mismatch for $(x_i, y_j)$ + min cost aligning substrings $x_1x_2 \ldots x_{i-1}$ and $y_1y_2 \ldots y_{j-1}$.

- **Case 2a.** $OPT$ leaves $x_i$ unmatched. Pay gap for $x_i$ and min cost of aligning $x_1x_2 \ldots x_{i-1}$ and $y_1y_2 \ldots y_j$.

- **Case 2b.** $OPT$ leaves $y_i$ unmatched. Pay gap for $y_i$ and min cost of aligning $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_{j-1}$.

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

- Runtime: $\Theta(mn)$
- Space: $\Theta(mn)$
- English words: $m, n \leq 10$
- Biology: $m, n \approx 10^5$
  - $10^{10}$ operations OK ...
  - 10 GB array is a problem
  - Can cut space down to $O(m + n)$ (see Section 6.7)
Outline

1. Weighted Interval Scheduling
2. Knapsack Problem
3. String Similarity
4. Common Errors with Dynamic Programming
Consider this Dynamic Programming “solution” to the Travelling Salesman Problem

Order the points $p_1, \ldots, p_n$ arbitrarily.

\[
\begin{array}{l}
\text{for } i = 1, \ldots, n \text{ do} \\
\quad \text{for } j = 1, \ldots, i \text{ do} \\
\quad \quad \text{Take optimal solution for points } p_1, \ldots, p_{i-1}, \text{ and put point } p_i \text{ right after } p_j. \\
\quad \text{end for} \\
\text{end for} \\
\end{array}
\]

Keep optimal of all the attempts above.

The runtime of this algorithm is $\Theta(n^2)$. Is it really this easy?
The runtime of this algorithm is $\Theta(n^2)$. Is it really this easy?
The runtime of this algorithm is $\Theta(n^2)$. Is it really this easy?

NO. We don’t have the “principle of optimality”.

Why should the optimal solution for points $p_1, \ldots, p_i$ be based on the optimal solution for $p_1, \ldots, p_{i-1}$??
The runtime of this algorithm is $\Theta(n^2)$. Is it really this easy?

We have not bothered to prove the optimality for many of the problems we considered, because it is “clear”. But be sure to check.
What if we changed the previous algorithm to keep track of all ordering of points \( p_1, \ldots, p_i \)? The optimal solution for \( p_1, \ldots, p_{i+1} \) must come from one of those, right?
What if we changed the previous algorithm to keep track of all ordering of points $p_1, \ldots, p_i$? The optimal solution for $p_1, \ldots, p_{i+1}$ must come from one of those, right?

Sure, that would work.
What if we changed the previous algorithm to keep track of all ordering of points $p_1, \ldots, p_i$? The optimal solution for $p_1, \ldots, p_{i+1}$ must come from one of those, right?

But now you’re doing $n!$ work.