Dynamic Programming Examples

Imran Rashid

University of Washington

February 27, 2008

Lecture Outline

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

Algorithmic Paradigms

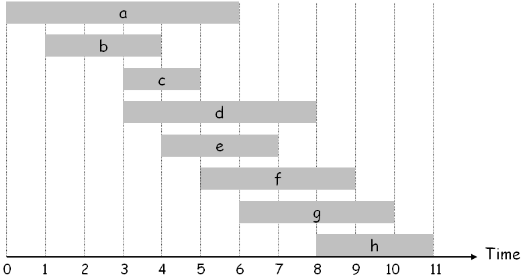
- Greed. Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming. Break up a problem 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.

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.

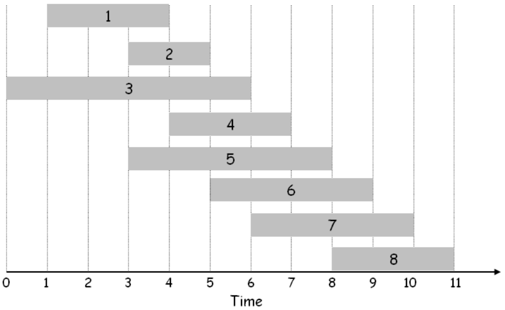


Unweighted Interval Scheduling Review

- 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.
- Can Greedy work when there are weights?

Weighted Interval Scheduling

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



i	$p(i)$
0	-
1	0
2	0
3	0
4	1
5	0
6	2
7	3
8	5

Dynamic Programming: Binary Choice

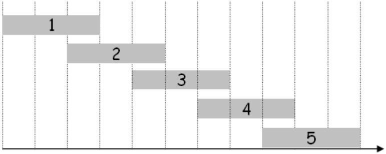
- Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.
  - Case 1: OPT selects job j.
    - can't use incompatible jobs $\{p(j)+1, p(j)+2, ..., j-1\}$
    - must include optimal solution to problem consisting of remaining compatible jobs $1, 2, ..., p(j)$
  - 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

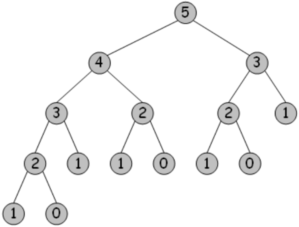
- Brute force algorithm.

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

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 $p()$: $O(n)$ 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: Θ 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

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?

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

Knapsack Subproblems: first try

- Def. $OPT(i) =$ 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.

Knapsack Subproblems: second try

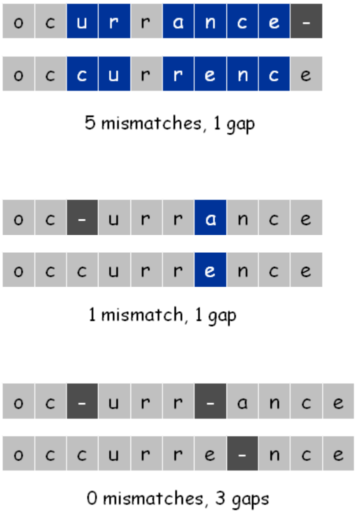
- Def. $OPT(i, S) =$ max value subset of items $1, \ldots, i$, using items in the set S.
- Works, but ...

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.

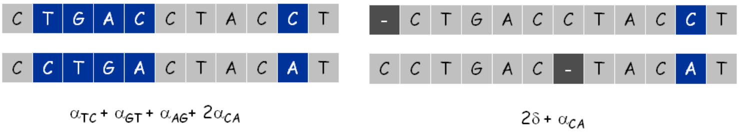
String Similarity

- How similar are two strings?
  1. ocurrance
  2. occurrence



String Edit Distance

- Applications
  - Basis for "diff"
  - Speech Recognition
  - Computational Biology
- Edit Distance
  - Gap Penalty δ; mismatch-penalty α_{pq}
  - Cost $=$ sum of gap and mismatch penalties



Sequence Alignment

- **Goal** Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ 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 f $i < i'$ but $j > j'$.

Sequence Alignment Subproblems

- **Def** $OPT(i, j)$ = min cost of aligning strings $x_1x_2 \ldots x_i$ 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 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)

Dynamic Programming and TSP(1)

- Consider this Dyanmic Programming "solution" to the Travelling Salesman Problem

---

Order the points $p_1, \ldots, p_n$ arbitrarily.
for $i = 1, \ldots n$ **do**
    for $j = 1, \ldots i$ **do**
        Take optimal solution for points $p_1, \ldots p_{i-1}$, and put point p_i right after p_j.
    end for
    Keep optimal of all the attempts above.
end for

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

Dynamic Programming and TSP (2)

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

Dynamic Programming and TSP (3)

- What if we changed the previous algorithm to keep track of all ordering of points $p_1, \ldots, p_i$? The optimal solution for $p_1, \ldots, p_{i+1}$ <u>must</u> come from one of those, right?