

Algorithm Design and Analysis

CSE
565

LECTURE 18 Dynamic Programming

- Weighted Interval Scheduling

Sofya Raskhodnikova

10/8/2007

S. Raskhodnikova; based on slides by K. Wayne.

CSE
565

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.

10/8/2007

S. Raskhodnikova; based on slides by K. Wayne.

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"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography*.

Dynamic Programming Applications

Areas.

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

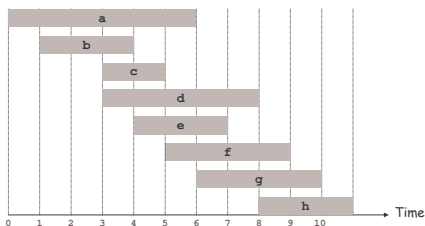
Some famous dynamic programming algorithms.

- 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.

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.

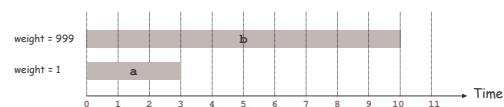


Unweighted Interval Scheduling Review

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.

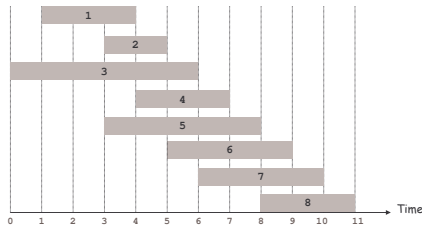


Weighted Interval Scheduling

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

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 $1, 2, \dots, j$.

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

optimal substructure

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

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

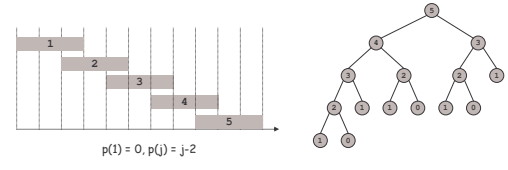
```

Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
Compute  $p(1), p(2), \dots, p(n)$ 
Compute-Opt( $j$ ) {
  if ( $j = 0$ )
    return 0
  else
    return  $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$ 
}
    
```

Weighted Interval Scheduling: Brute Force

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 Fibonacci sequence.



Review Question

Prove that the solution to the recurrence $T(n) = T(n-1) + T(n-2) + \Theta(1)$ is exponential in n .

Weighted Interval Scheduling: Memoization

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

```

Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
Compute  $p(1), p(2), \dots, p(n)$ 

for  $j = 1$  to  $n$ 
   $M[j] = \text{empty}$ 
 $M[0] = 0$ 

M-Compute-Opt( $j$ ) {
  if ( $M[j]$  is empty)
     $M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$ 
  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.
- $M\text{-Compute-Opt}(j)$: 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 $\Phi = \#$ nonempty entries of $M[\cdot]$.
 - initially $\Phi = 0$, throughout $\Phi \leq n$.
 - (ii) increases Φ 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: Finding a Solution

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

A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run 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:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
Compute  $p(1), p(2), \dots, p(n)$ 

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