EE360C: Algorithms

Dynamic Programming (1/4)

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Introduction

A Review of Algorithmic Techniques

Greedy

Build up a solution incrementally, myopically optimizing some local criterion

Divide and Conquer

Break up a problem into two (or more) subproblems, solve each subproblem independently, and combine the solutions to the subproblems to form a solution to the original problem

Dynamic Programming

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

A History

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

Etymology

- Dynamic programming = planning over time
- Secretary of Defense at the time was hostile to mathematical research
- Bellman sought an impressive name to avoid confrontation
    - “something not even a Congressman could object to”

Dynamic Programming

Dynamic Programming is an algorithm design technique that, like divide and conquer, relies on solutions to sub problems to solve a problem

- in dynamic programming, however, the subproblems are not independent
- a dynamic programming algorithm solves each of these subproblems just once, saving time in comparison to a traditional divide and conquer approach

Dynamic programming is commonly used for **optimization problems** in which many possible solutions exist; the algorithm helps us find one of the possibilities

- every solution to the problem has a value
- the algorithm helps us find a solution with the optimal value

Dynamic Programming Structure

Dynamic programming algorithms typically involve the following steps:

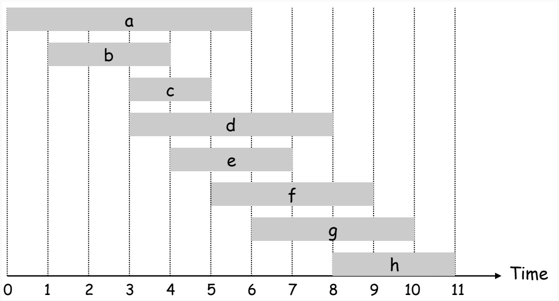
- characterize the structure of an optimal solution
- recursively define the value of an optimal solution
- compute the value of an optimal solution in a bottom-up fashion
- construct an optimal solution from the computed information

Weighted Interval Scheduling

Weighted Interval Scheduling

The weighted interval scheduling problem:

- Job j starts at s_j, finishes at f_j, and has weight or value v_j
- Two jobs are *compatible* if they do not overlap
- Goal: find maximum weight subset of mutually compatible jobs

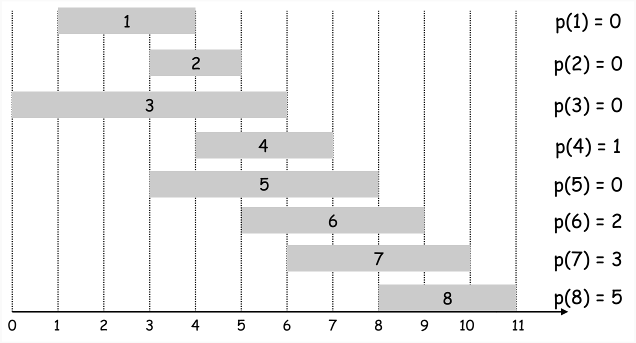


Unweighted Interval Scheduling

Recall that greedy algorithm that works perfectly (optimally) if all the weights are 1. The greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Back to Weighted Interval Scheduling

Label (and sort) jobs by finishing time ($f_1 \leq f_2 \leq \ldots \leq f_n$).
Define $p(j)$ to be the largest index $i < j$ such that job i is compatible with job j.



Weighted Interval Scheduling

Consider an optimal solution $\mathcal{O}$.

- **Case 1:** $\mathcal{O}$ selects job j.
    - then $\mathcal{O}$ can't include any of the jobs in the range $p(j) + 1, p(j) + 2, \ldots j - 1$
    - $\mathcal{O}$ must include the optimal solution to the problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$. If it didn't, we could replace $\mathcal{O}$ choice of requests with a better one from $1, 2, \ldots, p(j)$ with no danger of overlapping requests n.
- **Case 2:** $\mathcal{O}$ does not select job j.
    - $\mathcal{O}$ must include the optimal solution to the problem consisting of remaining compatible jobs $1, \ldots j - 1$.

Weighted Interval Scheduling

Let $\mathcal{O}_j$ be the optimal solution to the problem consisting of requests $1, \ldots, j$, and let $\mathrm{OPT}(j)$ denote the value of this solution

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

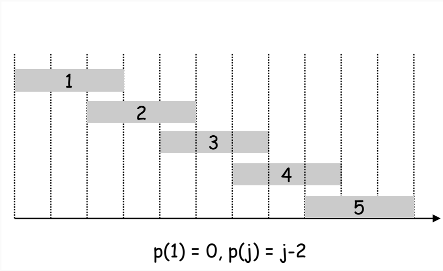
Weighted Interval Scheduling: Brute Force

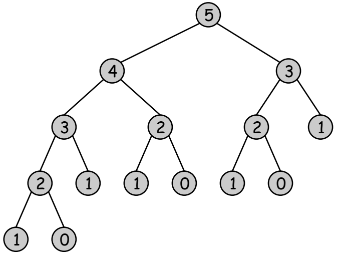
```
Input: $n$, $s_1,\dots,s_n$ , $f_1,\dots,f_n$ , $v_1,\dots,v_n$

Sort jobs by finish times so that $f_1 \le f_2 \le \dots \le f_n$.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max($v_j$ + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Weighted Interval Scheduling: Brute Force is Bad

The recursive algorithm fails spectacularly because of redundant sub-problems (it’s an exponential algorithm)

- the number of recursive calls for a family of “layered” instances grows like the Fibonacci sequence





Weighted Interval Scheduling: Memoization

Memoization

store the results of each sub-problem in a cache; look them up 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 $f_1 \le f_2 \le \dots \le f_n$.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty   ⟵ global array
M[j] = 0

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

Weighted Interval Scheduling: Memoization is Good

What is the running time of the memoized algorithm? $O(n \lg n)$

- Sort by finish time: $O(n \lg n)$
- Computing $p(\cdot)$: $O(n)$ after sorting by start time
- `M-Compute-Opt(j)`: each invocation takes $O(1)$ time and either:
    - returns an existing value `M[j]`
    - fills in one new entry `M[j]` and makes two recursive calls
- Define a progress measure Φ as the number of nonempty entries in `M[]`.
    - initially $\Phi = 0$; throughout $\Phi \leq n$
    - a call to `M-Compute-Opt(j)` increases Φ by at most 1; Φ cannot be increased more than n times; there are at most $2n$ recursive calls
- so the overall running time of `M-Compute-Opt(n)` is $O(n)$

Weighted Interval Scheduling: Finding a Solution

This algorithm has only computed the optimal value (i.e., the weight of the optimal schedule). What if we want the solution itself?

```
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)
}
```

Bottom-Up Weighted Interval Scheduling

To do this via dynamic programming, basically, we unwind the recursion...

```
Input: n, $s_1,\ldots,s_n$ , $f_1,\ldots,f_n$ , $v_1,\ldots,v_n$

Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$.

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])
}
```

Dynamic Programming Generalities

When Dynamic Programming Applies

For an optimization problem to be a good candidate for dynamic programming, it should exhibit:

- optimal substructure
- overlapping problems

Optimal Substructure

A problem exhibits the optimal substructure property if the optimal solution contains within it optimal solutions to subproblems

- be careful, though, optimal substructure may also make a problem a good candidate for a greedy solution

We use the optimal substructure by then solving the subproblems bottom-up, combining the subproblem solutions as we move up.

Overlapping Subproblems

If the recursive solution is going to solve the same subproblem multiple times, then we're wasting computation

- this is in contrast to problems that generate unique subproblems at every split, for which divide and conquer is a good fit
- one alternative for overlapping subproblems is *memoization*
    - it's like dynamic programming, but it fills the table in using a top-down approach that's more like traditional recursion
    - you create a table to use in a recursive algorithm, and the recursive calls fill in the table as they return
    - results can then be just computed once and looked up thereafter
- more traditional dynamic programming is "bottom up"

Questions