Lecture 19 – Dynamic Programming

Reading: KT Sections 6.1 and 6.2

Weighted Interval Scheduling problem
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.

Let’s try solving it

- Will greedy work?

- How about divide and conquer?

- Is there some structure in the problem that we can exploit?
Let’s define a few notions

**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.** $p(8) = 5, p(7) = 3, p(2) = 0$.

More notations

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

**Goal.** $OPT(n)$ = value of optimal solution to the original problem.

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

**Case 2.** $OPT(j)$ does not select job $j$.
- Must include optimal solution to problem consisting of remaining jobs $1, 2, \ldots, j - 1$.

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

**BRUTE-FORCE** \((n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)\)

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

Compute \(p[1], p[2], \ldots, p[n]\).

**RETURN** \(\text{COMPUTE-OPT}(n)\).

What’s the complexity of this algorithm?

**COMPUTE-OPT** \((j)\)

**If** \(j = 0\)

**RETURN** \(0\).

**Else**

**RETURN** \(\max \{ v_j + \text{COMPUTE-OPT}(p[j]), \text{COMPUTE-OPT}(j-1) \}\).

Memoization

**Top-down dynamic programming (memoization).** Cache result of each subproblem; lookup as needed.

**TOP-DOWN** \((0, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)\)

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

Compute \(p[1], p[2], \ldots, p[n]\).

\(M[0] \leftarrow 0\). — global array \(M\)

**RETURN** \(\text{M-COMPUTE-OPT}(n)\).

**M-COMPUTE-OPT** \((j)\)

**If** \(M[j] = \text{uninitialized}\)

\(M[j] \leftarrow \max \{ v_j + \text{M-COMPUTE-OPT}(p[j]), \text{M-COMPUTE-OPT}(j-1) \}\).

**RETURN** \(M[j]\).
Bottom-up dynamic programming

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

**Bottom-Up** \((n, s_1, \ldots, s_k; f_1, \ldots, f_k; v_1, \ldots, v_k)\)

Sort jobs by finish time so that \(f_1 \leq f_2 \leq \ldots \leq f_k\).

Compute \(p[1], p[2], \ldots, p[n]\).

\(M[0] \leftarrow 0\), \hspace{1cm} \text{previously computed values}

\textbf{FOR} \(j = 1\) \textbf{TO} \(n\)

\(M[j] \leftarrow \max\{ v_j + M[p[j]], M[j-1] \} \).

**Running time.** The bottom-up version takes \(O(n \log n)\) time.