Interval scheduling

Weighted Interval Scheduling

1. **Input** $R = \{r_1, r_2, \cdots, r_n\}$, each job r_i has interval (s_i, f_i) and weight/value v_i.
2. **Goal** Maximize the total value of jobs scheduled, i.e.

$$\sum_{i \in S} v_i$$

Approach

1. **Choose the highest value** Not optimal, consider

$$---------3---------$$

$$--2--\qquad --2--$$

2. **Choose by the maximum value/length ratio** Not optimal. consider

$$-----1------\quad -------1-------$$

$$--1--$$

3. **Choose earliest ending job** Not optimal, consider

$$--10--\quad --10--\quad --10--$$

$$-------100-------$$

4. **Choose job with least value/#conflict ratio** Not optimal, consider

$$----------21(7)----------$$

$$--10(10)--\quad --6(6)--\quad --2(2)--$$

Nothing really works, usually greedy algo are simple in nature, not involving complex ratios..

Dynamic Programming

Decompose a problem into subproblems, and combine solutions.

Example. scheduling Schedule a job $S \subseteq R$. Determine if $r_n \in S$.
Sort all jobs by finish time f_i. Define function $P : R \to R$ such that $P(r_i) = r_j$, where $f_j < f_i$ is the job with largest finish time such that r_j does not overlap with r_i. Hence all jobs between r_j and r_i overlap with r_i

1. **case 1** $r_n \in S$. Any job between $P(r_n)$ and r_n is not in S, next job we can possibly schedule is $P(r_n)$, next recursively search $\{r_1, \cdots, P(r_n)\}$
2. **case 2** $r_n \notin S$ next job we can possibly schedule is r_{n-1}, next recursively search $\{r_1, \cdots, r_{n-1}\}$

Solution. □

Let O_j be an optimal subproblem $\{r_1, \cdots, r_j\}$, where $1 \leq j \leq n$. Let OPT_j is total value of that optimal solution

$$\sum_{i \in S_{O_j}} v_i$$

1. If $r_j \in S_{O_j}$, $OPT_j = OPT_{P(j)} + v_j$
2. If $r_j \notin S_{O_j}$, $OPT_j = OPT_{j-1}$
3. together $OPT_j = max\{OPT_{j-1}, OPT_{P(j)} + v_j\}$
   (a) If $OPT_j > OPT_{P(j)} + v_j$
   (b) otherwise $r_j \in S_{O_j}$ otherwise $r_j \neq S_{O_j}$

$$\begin{aligned}
&Weighted - Interval - Scheduling - Help(j, P)\\
&\quad If \quad j == 0\\
&\qquad return \quad 0\\
&\quad else\\
&\qquad S_1 = Weighted - Interval - Scheduling(P(j), P)\\
&\qquad S_2 = Weighted - Interval - Scheduling(j-1, P)\\
&\quad return \quad max(S_1, S_2)
\end{aligned}$$

Complexity: $O(2^n)$. Since each iteration reduce a problem of size n to two subproblems of size $n-1$ with recurrene relation of $T(n) = 2T(n-1)$. We can avoid repeated computation by caching

$$
\begin{aligned}
&Weighted - Interval - Scheduling - Value(j, P, M)\\
&\quad If \quad M[j] \quad defined\\
&\qquad return \quad M[j]\\
&\quad else\\
&\qquad a = Weighted - Interval - Scheduling - Help(j, P)\\
&\qquad M[j] = a\\
&\qquad return \quad a
\end{aligned}
$$

Complexity: $O(n)$. Since require only one computation, with result stored in array A

$$
\begin{aligned}
&Weighted - Interval - Scheduling - Value(n, P, M)\\
&Weighted - Interval - Schedule(j, P, A)\\
&\quad If \quad M[j] == M[P(j)] + v_j\\
&\qquad print \quad r_j\\
&\qquad g = P(j)\\
&\quad Else\\
&\qquad g = j - 1\\
&\quad Return \quad Weighted - Interval - Schedule(g, P, M)
\end{aligned}
$$

Key points of dynamic programming

1. Optimal substructure
2. Recursion relation between subproblem and the main problem
3. Implementing recursion correctly will likely have an exponential growth (i.e. solving same subproblem over and over again, similar to brute force). We improve upon this using memoization, upon which the complexity drops from exponential to (sub-)polynomial.