Lecture Notes: Design and Analysis of Algorithms Overview, Interval Scheduling (Course: MIT 6.046J)

Thobias K. Høivik

April 9, 2025

Basic Terminology in Complexity

Definition 1 (P)**.** *We denote the class of problems solvable in polynomial time by P. $\Rightarrow O(n^k)$ for some constant k.*

Definition 2 (NP)**.** *We denote the class of problems whose solutions can be verified in polynomial time by NP.*

Example: Determining whether a cycle in a graph is a hamiltonian cycle is trivial, but finding one is not.

Definition 3 (NP-Complete)**.** *The problem is in NP and is as "difficult" as any problem in NP.*

A result of this definition is that if any $NP - Complete$ problem is shown to be solvable in polynomial time, NP collapses to P. The hamiltonian cycle problem is one such problem.

Interval Scheduling

Resources & requests $1, \ldots, n$. $s(i)$ start time, $f(i)$ finish time, $s(i) < f(i)$. We say two requests i, j are compatible if they don't overlap, i.e $f(i) \leq s(j) \vee f(j) \leq s(i)$. If we have a set of requests represented as intervals then we claim we can find the subset which has the highest number of compatible requests by using a greedy algorithm (an algorithm that makes locally optimal choices).

Definition 4 (Greedy Algorithm)**.** *A greedy algorithm is a myopic algorithm (informally: with no knowledge/consideration of future events and information) that processes the input one piece at a time with no apparanet lookahead.*

Greedy Interval Scheduling

1. Use a simple rule to select a request i.
2. Reject all requests that are incompatible with i.
3. Repeat until all requests are processed.

In this case a "simple rule" would be to choose the request i with earliest finish time $f(i)$.
Claim: Given a list of intervals L, greedy algorithm with earliest finish time rule produces k^* intervals, where k^* is maximum.

Proof. We prove this claim using an exchange argument via induction on the number of intervals selected by the greedy algorithm.
Base Case: The greedy algorithm picks the interval with the earliest finish time, say a_1, and let the optimal solution pick some interval o_1. Since the greedy algorithm selects the interval with the earliest finish time, we have $f(a_1) \leq f(o_1)$. If $a_1 = o_1$, then we are done. Otherwise, we can replace o_1 with a_1, and the resulting solution remains valid because a_1 finishes no later than o_1, and thus does not overlap with any intervals in the optimal schedule that follow o_1.
Inductive Hypothesis: Suppose the greedy algorithm selects $a_1, a_2, \ldots, a_r$, and there exists an optimal solution $o_1, o_2, \ldots, o_{k^*}$ such that we can replace the first r intervals in the optimal solution with the greedy ones without affecting feasibility.
Inductive Step: Consider the $(r+1)$-th interval selected by the greedy algorithm, a_{r+1}. Let o_j be the first interval in the optimal solution that starts after $f(a_r)$. Since the greedy algorithm selects the earliest finishing compatible interval, we have $f(a_{r+1}) \leq f(o_j)$. We can replace o_j with a_{r+1} in the optimal solution, and the schedule remains feasible. By repeating this process, we eventually construct an optimal solution that includes all intervals selected by the greedy algorithm.
Hence, the greedy algorithm selects $k^* = k$ intervals, which is optimal. □

Weighted Interval Scheduling

In this version of the interval scheduling problem, each interval (or request) i has an associated weight $w(i)$, representing its value or importance. The goal is to select a subset of non-overlapping intervals whose total weight is maximized.
Unlike the unweighted case (where we could use a greedy algorithm based on earliest finish time), there is **no greedy algorithm** that always works for the weighted version. Instead, we solve it using **dynamic programming**.
Key Idea: For each interval i, we consider two options:

- Include interval i in our solution.
- Skip interval i and move to the next one.

If we include interval i, we must skip all intervals that overlap with it. Therefore, we need to know which intervals come *after* it without overlapping.
We define subproblems as follows:

$$R^x = \{\text{requests } j \in R \mid s(j) \geq x\}$$

That is, R^x is the set of all intervals that start **at or after time** x. In particular, for interval i, the subproblem $R^{f(i)}$ is the set of all intervals that start after i finishes.
DP Recurrence: For the whole set of requests R, we guess the first interval i to include.

If we pick i, we get its weight $w(i)$ plus the optimal weight we can get from the remaining compatible intervals (i.e., those in $R^{f(i)}$).

$$opt(R) = \max_{1 \leq i \leq n} \left(w(i) + opt(R^{f(i)}) \right)$$

In practice, we sort the intervals by finish time and precompute for each interval i the index of the last interval that finishes before $s(i)$ (commonly denoted as $p(i)$). Then we define:

$$OPT(i) = \max\left(w(i) + OPT(p(i)),\ OPT(i-1)\right)$$

This recurrence means: either we take interval i and add its weight to the best solution ending before it, or we skip it and stick with the best solution for the first $i-1$ intervals. This version allows us to compute the solution efficiently using bottom-up dynamic programming.