Lecture 1: Introduction

6.006 pre-requisite:

- Data structures such as heaps, trees, graphs
- Algorithms for sorting, shortest paths, graph search, dynamic programming

Course Overview

This course covers several modules:

1. Divide and Conquer - FFT, Randomized algorithms
2. Optimization - greedy and dynamic programming
3. Network Flow
4. Intractibility (and dealing with it)
5. Linear programming
6. Sublinear algorithms, approximation algorithms
7. Advanced topics

Theme of today's lecture

Very similar problems can have very different complexity. Recall:

- **P**: class of problems solvable in polynomial time. $O(n^k)$ for some constant k.

  Shortest paths in a graph can be found in $O(V^2)$ for example.

- **NP**: class of problems verifiable in polynomial time.

  Hamiltonian cycle in a directed graph $G(V, E)$ is a simple cycle that contains each vertex in V.

  Determining whether a graph has a hamiltonian cycle is NP-complete but verifying that a cycle is hamiltonian is easy.

- **NP-complete**: problem is in NP and is as hard as any problem in NP.

  If any NPC problem can be solved in polynomial time, then every problem in NP has a polynomial time solution.

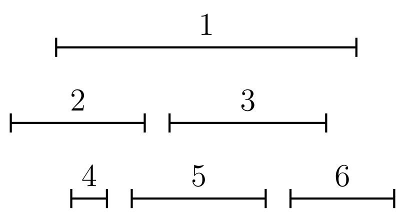
Interval Scheduling

Requests $1, 2, \ldots, n$, single resource

$s(i)$ start time, $f(i)$ finish time, $s(i) < f(i)$ (start time must be less than finish time for a request)

Two requests i and j are compatible if they don't overlap, i.e., $f(i) \leq s(j)$ or $f(j) \leq s(i)$.

In the figure below, requests 2 and 3 are compatible, and requests 4, 5 and 6 are compatible as well, but requests 2 and 4 are not compatible.



Goal: Select a compatible subset of requests of maximum size.

Claim: We can solve this using a greedy algorithm.

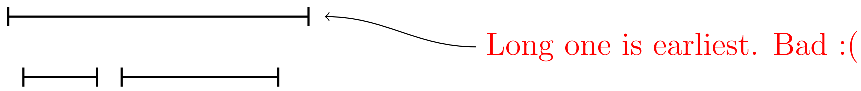
A greedy algorithm is a myopic algorithm that processes the input one piece at a time with no apparent look ahead.

Greedy Interval Scheduling

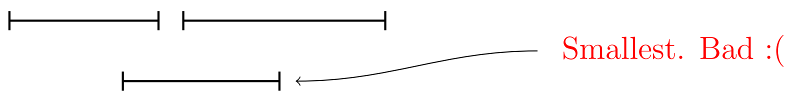
1. Use a simple rule to select a request i.
2. Reject all requests incompatible with i.
3. Repeat until all requests are processed.

Possible rules?

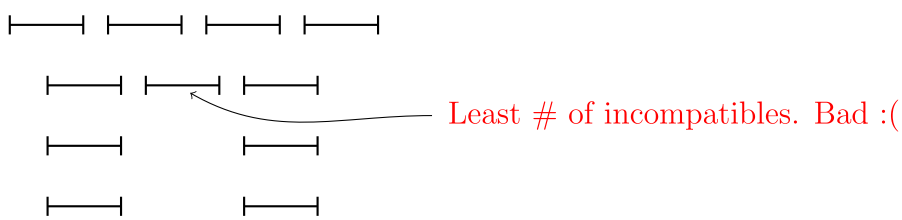
1. Select request that starts earliest, i.e., minimum $s(i)$.



2. Select request that is smallest, i.e., minimum $f(i) - s(i)$.



3. For each request, find number of incompatibles, and select request with minimum such number.



4. Select request with earliest finish time, i.e., minimum $f(i)$.

Claim 1. *Greedy algorithm outputs a list of intervals*

$$< s(i_1), f(i_1) >, < s(i_2), f(i_2) >, \ldots, < s(i_k), f(i_k) >$$

such that

$$s(i_1) < f(i_1) \leq s(i_2) < f(i_2) \leq \ldots \leq s(i_k) < f(i_k)$$

Proof. Simple proof by contradiction – if $f(i_j) > s(i_{j+1})$, interval j and $j+1$ intersect, which is a contradiction of Step 2 of the algorithm! □

Claim 2. *Given list of intervals L, greedy algorithm with earliest finish time produces k^* intervals, where k^* is optimal.*

Proof. Induction on k^*.

Base case: $k^* = 1$ – this case is easy, any interval works.

Inductive step: Suppose claim holds for k^* and we are given a list of intervals whose optimal schedule has $k^* + 1$ intervals, namely

$$S^*[1, 2, \ldots, k^* + 1] = < s(j_1), f(j_1) >, \ldots, < s(j_{k^*+1}), f(j_{k^*+1}) >$$

Say for some generic k, the greedy algorithm gives a list of intervals

$$S[1, 2, \ldots, k] =< s(i_1), f(i_1) >, \ldots, < s(i_k), f(i_k) >$$

By construction, we know that $f(i_1) \leq f(j_1)$, since the greedy algorithm picks the earliest finish time.

Now we can create a schedule

$$S^{**} =< s(i_1), f(i_1) >, < s(j_2), f(j_2) >, \ldots, < s(j_{k^*+1}), f(j_{k^*+1}) >$$

since the interval $< s(i_1), f(i_1) >$ does not overlap with the interval $< s(j_2), f(j_2) >$ and all intervals that come after that. Note that since the length of S^{**} is $k^* + 1$, this schedule is also optimal.

Now we proceed to define L' as the set of intervals with $s(i) \geq f(i_1)$.

Since S^{**} is optimal for L, $S^{**}[2, 3, \ldots, k^* + 1]$ is optimal for L', which implies that the optimal schedule for L' has k^* size.

We now see by our initial inductive hypothesis that running the greedy algorithm on L' should produce a schedule of size k^*. Hence, by our construction, running the greedy algorithm on L' gives us $S[2, \ldots, k]$.

This means $k - 1 = k^*$ or $k = k^* + 1$, which implies that $S[1, \ldots, k]$ is indeed optimal, and we are done.

□

Weighted Interval Scheduling

Each request i has weight $w(i)$. Schedule subset of requests that are non-overlapping with maximum weight.

A key observation here is that the greedy algorithm no longer works.

Dynamic Programming

We can define our sub-problems as

$$R^x = \{j \in R | s(j) \geq x\}$$

Here, R is the set of all requests.

If we set $x = f(i)$, then R^x is the set of requests later than request i.

Total number of sub-problems $= n$ (one for each request)

Only need to solve each subproblem once and memoize.

We try each request i as a possible first. If we pick a request as the first, then the remaining requests are $R^{f(i)}$.

Note that even though there may be requests compatible with i that are not in $R^{f(i)}$, we are picking i as the first request, i.e., we are going in order.

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

Total running time is $O(n^2)$ since we need $O(n)$ time to solve each sub-problem.

Turns out that we can actually reduce the overall complexity to $O(n \log n)$. We leave this as an exercise.

Non-identical machines

As before, we have n requests $\{1, 2, \ldots, n\}$. Each request i is associated with a start time $s(i)$ and finish time $f(i)$, m different machine types as well $\tau = \{T_1, \ldots, T_m\}$. Each request i is associated with a set $Q(i) \subseteq \tau$ that represents the set of machines that request i can be serviced on.

Each request has a weight of 1. We want to maximize the number of jobs that can be scheduled on the m machines.

This problem is in NP, since we can clearly check that a given subset of jobs with machine assignments is legal.

Can $k \leq n$ requests be scheduled? This problem is NP-complete.

Maximum number of requests that should be scheduled? This problem is NP-hard.

Dealing with intractability

1. Approximation algorithms: Guarantee within some factor of optimal in polynomial time.

2. Pruning heuristics to reduce (possible exponential) runtime on "real-world" examples.

3. Greedy or other sub-optimal heuristics that work well in practice but provide no guarantees.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015