Algorithms

Jane Doe

Table of Contents

1. Greedy Algorithms

1.1. Interval Scheduling

Lecture 1 Jan 1, 2025

1.1.1. Computational Problem: Interval Scheduling

Suppose we have a list of n tasks that we want to schedule on a single processor. Each activity is specified by its start and end times, and only one activity can be scheduled on the resource at a time. Note an activity cannot be paused, i.e., it uses the resource continuously between its start and end times.

Our goal is to schedule the maximum possible number of activities.

1.1.2. Remark

We formalize the previous problem as follows:

Call our set of activities $S = \{1, 2, ..., n\}$. The i-th activity is given by a tuple $(s(i), f(i))$ with $s(i) \leq f(i)$, for $i = 1, ..., n$, where $s(i)$ represents its start time and $f(i)$ represents its finish time.

Define a *feasible schedule* as a subset in which no two activities overlap. Thus, our goal is to find a feasible schedule of maximum size.

1.1.3. Example: Interval Scheduling

- Suppose we have 5 activities:

$$\{(3, 6), (1, 4), (4, 10), (6, 8), (0, 2)\}$$

- A feasible schedule cannot have two activities that overlap (in time).
- For instance, we cannot accept both (1, 4) and (3, 6).
- However, (3, 6) and (6, 8) are acceptable, because second only begins when first ends.
- The optimal solution in this example has size 3.

1.1.4. Remark

We have a few possible greedy strategies:

1. FIFO: pick the one that starts first, remove overlapping activities and repeat
2. Shortest first: pick the activity with the shortest duration, remove overlapping activities and repeat
3. Min overlap first: Count the number of other jobs that overlap, then choose one with smallest overlap count

But if we analyze these, we see simple counterexamples for the first one (an activity that's really long is first) and the third one (a shorter interval in the middle that overlaps with 2 which could fit together). We could also construct an example for teh second.

1.1.5. Algorithm: Earliest Finish Time (EFT)

The correct greedy strategy is to sort the process job in order by earliest finish time.

We sort jobs by order:

$$f(j_1) \leq f(j_2) \leq \cdots \leq f(j_n)$$

Pseudocode:

```
A = {1}; j = 1; // accept job 1
for i = 2 to n do
  if s(i) >= f(j) then
    A = A + {i}; j = i;
return A
```

1.1.6. Example: EFT

Activity	Start	Finish
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

- The greedy algorithm first chooses 1, then skips 2 and 3
- Next it chooses 4, and skips 5, 6, and 7
- It should choose 8 and 11 (verify)

1.1.7. Remark

- Suppose that OPT is an optimal solution for the problem. Ideally, we would like to show our algorithm always returns the same output, namely $A \equiv \text{OPT}$.
- But there may be multiple optima, and the best we can hope for is that $|A| = |\text{OPT}|$
- The proof idea (typical for optimality of greedy algorithms) is to show the greedy algorithm stays ahead of the optimal solution at all times

1.1.8. ***Proposition***

The Earliest Finish Time Algorithm is correct

Proof

Suppose $a_1, ..., a_k$ are the indices of jobs in the greedy schedule, and $b_1, ..., b_m$ are jobs in an optimal schedule OPT. We want to show $k = m$. Mnemonically, a_i's are jobs picked by our algorithm while b_i's are the optimal schedule jobs. (Note we list both in order of start times.)

- Our intuition should be that the greedy algorithm makes the resource free again as soon as possible. In particular, $f(a_1) \leq f(b_1)$, that is, the greedy algorithm stays ahead. We formalize this as follows.

1.1.9. ***Lemma:*** **Greedy Structure Lemma**

For every $i \leq k$, we have $f(a_i) \leq f(b_i)$.

Proof

We proceed by induction. Note the $i = 1$ case is trivial. By the induction hypothesis, we have $f(a_{i-1}) \leq f(b_{i-1})$. Therefore $f(a_{i-1}) \leq s(b_i)$, since clearly $f(b_{i-1}) \leq s(b_i)$. But notice $s(b_i) \leq f(b_i)$, which implies $f(a_i) \leq f(b_i)$ as desired.

□

Notice by the lemma, $f(a_k) \leq f(b_k)$, but $f(b_k) \leq f(b_m)$, so $f(a_k) \leq f(b_m)$, implying $k \leq m$. But $m \leq k$ since OPT is optimal, implying $m = k$ as desired.

□

1.1.10. Runtime Analysis

The runtime of the previous algorithm is $O(n \log n) + O(n) = O(n \log n)$ because of the sorting and then iteration through.

1.1.11. Remark

Consider the same setup with the following variant. Given a set of activities, what is the smallest number of machines needed to schedule them all?

A natural greedy algorithm to try is the following: use EFT to find the maximum number of activities that can be scheduled on one machine. Delete those and repeat the rest until no activities are left.

However, this algorithm fails because we can construct a setup where EFT uses 3 machines but the optimal solution is 2 machines.

Instead, a simpler greedy algorithm works:

```
Sort activities by start time
Put activity 1 on machine 1
for i = 2 to n:
  if i can be scheduled on any of the existing machines, add to that machine otherwise
schedule activity i on a new machine
```

1.1.12. ***Proposition***

> The previous algorithm is optimal.

Proof

Define the depth d as the maximum number of activities working at the same time. Notice that OPT >= depth, because we have to run all processes. Further, Greedy <= depth (since we only start on a new machine when there are $d - 1$ processes running). Therefore, the Greedy algorithm has to be optimal.

□