CS 341: ALGORITHMS (S18) — LECTURE 6
INTERVAL SCHEDULING

ERIC BLAIS

In the last two lectures, we explored how the divide and conquer technique is useful for designing algorithms for many different problems. Today, we start examining another technique: *greedy algorithms*.

1. Greedy algorithms

A *greedy* algorithm is one in which we:

(1) Break down a problem into a sequence of decisions that need to be made, then
(2) Make the decisions one at a time, each time choosing the option that is optimal at the moment (and not worrying about later decisions).

This is one of the simplest algorithm design techniques around,[1] and as we will see it yields efficient algorithm for many different problems. But the challenge with this technique in many instances is proving the algorithm's correctness—or, often, *determining* whether the algorithm is correct or not in the first place.

You have already seen one classic greedy algorithm in CS 240: Huffman codes are optimal prefix codes obtained by running the greedy algorithm to join trees with different frequencies together. This algorithm is both correct and efficient.

We also use a greedy algorithm in real life when we make change.

Problem 1 (Making change)**.** *Given coins with denominations $d_1 > d_2 > \cdots > d_n = 1$ and a value $v \geq 1$, determine the minimum number of coins whose denominations sum to v. (I.e., a valid solution $a_1, \ldots, a_n \in \mathbb{N}$ is one that satisfies $\sum_{i=1}^{n} a_i d_i = v$ and minimizes $\sum_{i=1}^{n} a_i$.)*

For example, we have coins worth 200, 100, 25, 10, 5, and 1 cents in current circulation in Canada. How would you make change for 1.43$ using those coins? You would start by taking a coin with maximal denomination that is at most 143 (here the loonie worth 100 cents), subtract that amount from the total (leaving 43 cents in this case), and repeat.

Algorithm 1: GreedyChange$(d_1 > d_2 > \cdots > d_n; v)$

for $i = 1, \ldots, n$ **do**
    $a_i \leftarrow \lfloor v \,/\, d_i \rfloor$;
    $v \leftarrow v \bmod d_i$;
return $(a_1, \ldots, a_n)$;

[1] As an aside: There is an even simpler technique that we will not cover in this course but for which I am especially fond: the *random choice* technique, where you don't even try to make the best decision but instead choose one at random instead.

Does this algorithm always return a valid solution to the Making Change problem? It does for every value v when the coins have denominations 200, 100, 25, 10, 5, and 1 but the proof of correctness in this case is not trivial. In general, however, there are denominations and choices of v where the algorithm does not return the minimal number of coins. Take for example the case where the denominations are 8, 7, and 1 and the value to return is 14. The greedy algorithm will return the solution $(1, 0, 6)$, for a total of 7 coins, when the solution $(0, 2, 0)$ requires only two coins.

Today, we explore another fundamental problem that can be solved using greedy algorithms.

2. Interval scheduling problem

Many real-world scheduling problems can be formulated in terms of the abstract *interval scheduling problem*.

Definition 6.1 (Interval scheduling problem)**.** An instance of the *interval scheduling problem* is a set of n pairs of start and finish times $(s_1, f_1), \ldots, (s_n, f_n)$ where each pair (s_i, f_i) satisfies $s_i < f_i$. A valid solution to an instance is a a maximum subset $I \subseteq \{1, 2, \ldots, n\}$ such that no two intervals in I overlap. I.e., for every $i, j \in I$, $s_i > f_j$ or $s_j > f_i$.

It is quite natural to try to use the greedy method to solve this problem: to do so, we just need to decide how the algorithm chooses the "best" interval to add to I given the intervals that have already been added to this set. We have many possibilities on how to define the notion of "best" interval:

Earliest starting time: Pick the interval with the earliest starting time that does not overlap any of the intervals we already added to I.

Earliest finish time: Pick the interval with the earliest finish time that does not overlap any of the intervals we already added to I.

Shortest interval: Pick the interval with the shortest length $f_i - s_i$ among those that don't overlap any of the intervals we already added to I.

Minimum conflicts: Let J be the set of intervals that don't overlap with any interval in I. Pick the interval in J that overlaps with the fewest other number of other intervals in J.

All four options sound perfectly reasonable, but only one yields an algorithm that always outputs a valid solution to the interval scheduling problem.

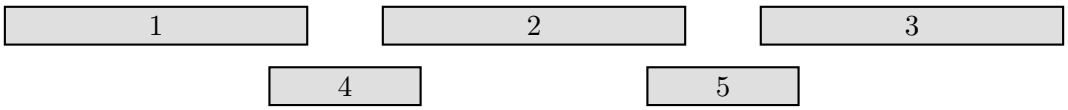
Proposition 6.2. *The greedy algorithm with earliest starting time does not always find a valid solution to the interval scheduling problem.*

Proof. Consider the set of intervals

The valid solution is $I = \{2, 3, 4\}$ but the greedy algorithm outputs $I = \{1\}$ instead. □

Proposition 6.3. *The greedy algorithm with shortest interval does not always find a valid solution to the interval scheduling problem.*

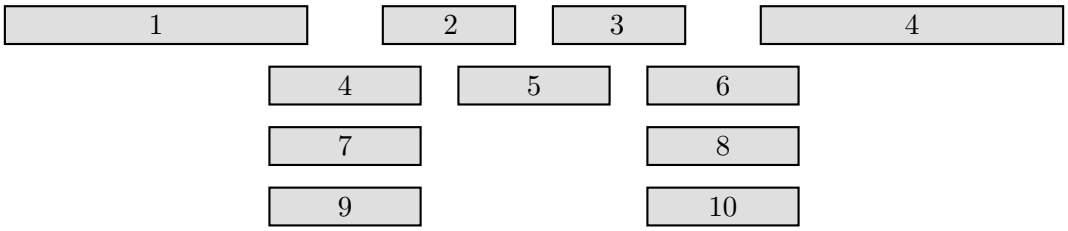
Proof. Consider the set of intervals



The valid solution is $\{1, 2, 3\}$ but the greedy algorithm outputs $I = \{4, 5\}$ instead. □

Proposition 6.4. *The greedy algorithm with minimal conflicts does not always find a valid solution to the interval scheduling problem.*

Proof. Consider the set of intervals



The valid solution is $\{1, 2, 3, 4\}$ but the greedy algorithm outputs $I = \{1, 4, 5\}$ instead. □

We are now left with a single candidate algorithm: earliest finish time. We can implement this greedy algorithm in a simple way: sort the intervals by finish time, and when we go through the list we add an interval (s_i, f_i) iff its start time is larger than the last (and therefore most recently added) finish time of any interval in I.

Algorithm 2: GreedyScheduler((s_1, f_1),...,(s_n, f_n))

```
A ← indices {1, ..., n} sorted by f_i;
I ← A[1];
f* ← f_{A[1]};
for i = 2, ..., n do
    if s_{A[i]} > f* then
        I ← I ∪ A[i];
        f* ← f_{A[i]};
return I;
```

The time complexity of the GreedyIntervalScheduler is $\Theta(n \log n)$. It remains to show that it always returns a valid solution to the interval scheduling problem. We do so via an *exchange argument*: we show that for any maximum set $I^* \subseteq [n]$ of non-interlapping intervals, we can exchange some of the intervals in I^* in a way that we end up with the output of the GreedyIntervalScheduler.

As a first step, let us show how we can always exchange the *first* interval in a maximum set I^* of non-overlapping intervals with the first interval chosen by GreedyIntervalScheduler and still obtain a valid solution to the interval scheduling problem.

Proposition 6.5. *Let $I^* = \{i_1, i_2, \ldots, i_k\} \subseteq \{1, 2, \ldots, n\}$ be the indices of a maximum set of non-overlapping intervals in some instance of the Interval Scheduling problem sorted by finish times so that $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$. Let $j \in [n]$ be the index of the first interval selected by the GreedyIntervalScheduler. Then $I^\dagger = \{j, i_2, i_3, \ldots, i_k\}$ is also a maximum set of non-overlapping intervals.*

Proof. The set $I^\dagger$ must have the same cardinality as I^* since $f_j \leq f_{i_1} < f_{i_\ell}$ for each $\ell \in \{2, 3, \ldots, k\}$ and so $j \notin \{i_2, \ldots, i_k\}$. We need to show that $I^\dagger$ is a set of non-overlapping intervals. Since I^* is a set of non-overlapping intervals, the only thing we need to show is that the interval j does not overlap with any of the intervals $i_2, \ldots, i_k$.

The fact that I^* contains non-overlapping intervals and that f_{i_1} is the smallest finish time implies that we must have $f_{i_1} < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$. And the definition of the GreedyIntervalScheduler guarantees that $f_j \leq f_{i_1}$, so we also have $f_j < s_{i_\ell}$ for each $\ell = 2, 3, \ldots, k$ and the interval j does not overlap with any of the intervals $i_2, \ldots, i_k$, as we wanted to show. □

We can now use a proof by induction to establish the correctness of the GreedyScheduler algorithm.

Theorem 6.6. *The* GreedyScheduler *solves the interval scheduler problem.*

Proof. Let $I^* = \{i_1, i_2, \ldots, i_k\}$ be a maximum set of non-overlapping intervals, and let $I^\dagger = \{j_1, j_2, \ldots, j_m\}$ be the set of non-overlapping intervals returned by the algorithm. We again sort the indices so that $f_{i_1} < f_{i_2} < \cdots < f_{i_k}$ and $f_{j_1} < \cdots < f_{j_m}$. Clearly, $m \leq k$; we want to show that in fact $m = k$.

Let us now use a proof by induction on c to show that for every c in the range $1 \leq c \leq m$, the set $I^{(c)} = \{j_1, j_2, \ldots, j_c, i_{c+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. The base case was established in Proposition 6.5.

For the induction step with $c \geq 2$, the induction hypothesis is that $I^{(c-1)}$ is a maximum set of non-overlapping intervals; we want to show the same is true of $I^{(c)}$. Note that $I^{(c)} = (I^{(c-1)} \setminus \{i_c\}) \cup \{j_c\}$. We must have $f_{j_c} \leq f_{i_c} < f_{i_{c+1}} < \cdots < f_{i_k}$ so $j_c \notin \{j_1, \ldots, j_{c-1}, i_{c+1}, \ldots, i_k\}$ and $|I^{(c)}| = |I^{(c-1)}|$. Furthermore, since $I^\dagger$ and $I^{(c-1)}$ are sets of non-overlapping intervals and since $f_{j_c} \leq f_{i_c} < s_{i_{c+1}} < \cdots < s_{i_k}$, the set $I^{(c)}$ is a maximum set of non-overlapping intervals.

The proof by induction we just completed shows that $I^{(m)} = \{j_1, j_2, \ldots, j_m, i_{m+1}, \ldots, i_k\}$ is a maximum set of non-overlapping intervals. But the greedy algorithm returns $I^\dagger = \{j_1, \ldots, j_m\}$ only if all other intervals with finish times greater than j_m intersect with some interval already in $I^\dagger$; this means that we must have $m = k$ and the algorithm returns a maximum set of non-overlapping intervals. □