CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 14: Independent Sets

Harvard SEAS - Fall 2021 *Oct. 21, 2021*

1 Announcements

Recommended Reading:

- CLRS Sec 16.1–16.2
- Salil OH today after class
- Active learning today!

2 Definitions

In the active learning exercise, you've seen the definition of independent sets, which are closely related to graph colorings:

Definition 2.1. Let $G = (V, E)$ be a graph. An *independent set* in G is a subset $S \subseteq V$ such that there are no edges entirely in S. That is, $\{u, v\} \in E$ implies that $u \notin S$ or $v \notin S$.

A proper k-coloring of a graph G is equivalent to a partition of V into k independent sets (each color class should be an independent set).

When we have a graph $G = (V, E)$ representing conflicts, instead of partitioning V into a small number of conflict-free subsets (as coloring would), it is sometimes useful to instead find a single, large conflict-free subset. This gives rise to the following computational problem:

Input : A graph $G = (V, E)$
Output : An independent set $S \subseteq V$ in G of maximum size

Computational Problem Independent Set

Example: throwing a big party where everyone will get along

Like with graph coloring, we can try a greedy algorithm for Independent Set:

`GreedyIndSet(`G`)`
Input : A graph $G = (V, E)$
Output : A "large" independent set in G
Choose an ordering $v_0, v_1, v_2, \ldots, v_{n-1}$ of V;
$S = \emptyset$;
foreach $i = 0$ *to* $n - 1$ **do**
    if $\forall j < i$ *s.t.* $\{v_i, v_j\} \in E$ *we have* $v_j \notin S$ **then** $S = S \cup \{v_i\}$;
return S

And, similarly to coloring, we can only prove fairly weak bounds on the performance of the greedy algorithm in general:

Theorem 2.2. *For every graph G with n vertices and m edges,* `GreedyIndSet(`G`)` *can be implemented in time $O(n+m)$ and outputs an independent set of size at least $n/(d_{max}+1)$, where d_{max} is the maximum vertex degree in G.*

Proof.
Omitted (and possibly covered in section). □

However, when there is more structure in the conflict graph, a careful ordering for the greedy algorithm can yield an optimal solution. An example of such structure comes from the Interval Scheduling problem we saw in the first lecture:

Input	: A collection of intervals $[a_0, b_0], \ldots, [a_{n-1}, b_{n-1}]$, where each $a_i, b_i \in \mathbb{R}$ and $a_i \leq b_i$
Output	: YES if the intervals are disjoint (for all $i \neq j$, $[a_i, b_i] \cap [a_j, b_j] = \emptyset$) NO otherwise

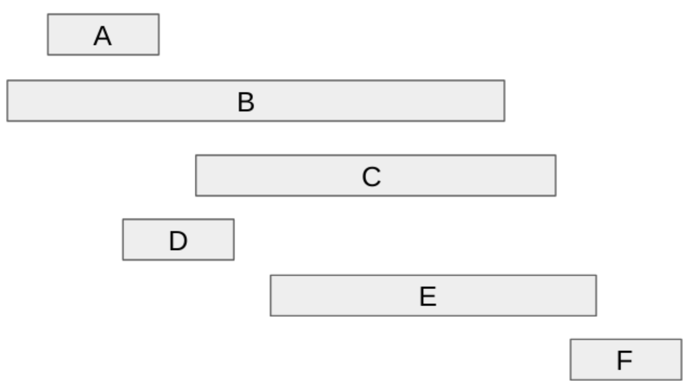
Computational Problem IntervalScheduling-Decision

We saw that we could solve this problem in time $O(n \log n)$ by reduction to Sorting. However, if the answer is NO, we might be satisfied by trying to schedule *as many* intervals *as possible*:

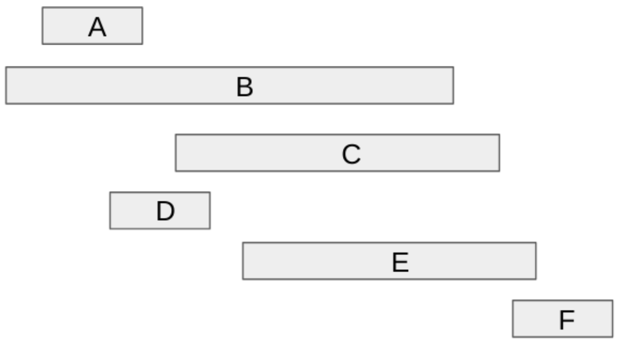
Input	: A collection of intervals $[a_0, b_0], \ldots, [a_{n-1}, b_{n-1}]$, where each $a_i, b_i \in \mathbb{Q}$ and $a_i \leq b_i$
Output	: A maximum-size subset $S \subseteq [n]$ such that $\forall i \neq j \in S$, $[a_i, b_i] \cap [a_j, b_j] = \emptyset$.

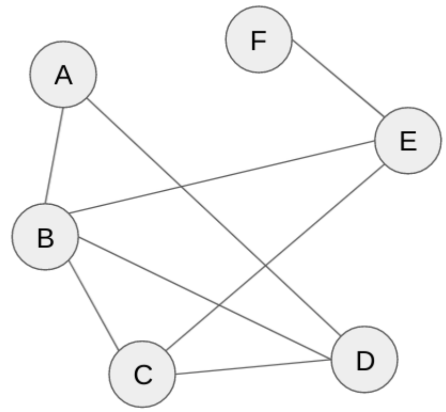
Computational Problem IntervalScheduling-Optimization

Example:



Q: How can we model IntervalScheduling-Optimization as an Independent Set problem?





A: We represent each interval as a vertex, and we place an edge between two vertices (i.e. intervals) if they conflict. Then an independent set is exactly a set of intervals which have no conflicts, so maximizing the size of this is equivalent to finding the largest set of conflict-free intervals.

With this graph-theoretic modelling, we can instantiate `GreedyIndSet()` for IntervalScheduling-Optimization:

`GreedyIntervalScheduling`(x)
Input : A list x of n intervals $[a, b]$, with $a, b \in \mathbb{Q}$
Output : A "large" subset of the input intervals that are disjoint from each other
Choose an ordering of the input intervals $[a_0, b_0], [a_1, b_1], \ldots, [a_{n-1}, b_{n-1}]$;
$S = \emptyset$;
foreach $i = 0$ *to* $n - 1$ **do**
  if $\forall j < i$ *s.t.* $j \in S$ *we have*$[a_j, b_j] \cap [a_i, b_i] = \emptyset$ **then** $S = S \cup \{i\}$;
return S

Q: What ordering of the input intervals should we use?

A: Want to first assign the intervals with the earliest *end* time.

Theorem 2.3. *If the input intervals are sorted by increasing order of end time b_i, then we have that that* `GreedyIntervalScheduling`(x) *will find an optimal solution to IntervalScheduling-Optimization, and can be implemented in time $O(n \log n)$.*

Proof.
Let $S^* = \{i_0^* \leq i_1* \leq \ldots \leq i_{k^*-1}^*\}$ be an optimal solution to Interval Scheduling. Then let $S = \{i_0 \leq i_1 \leq \ldots \leq i_{k-1}\}$ be the solution found by the greedy algorithm. Recall that b_{i_j} is the endtime of interval i_j (and above we sort both solutions on end time).

Claim 2.4 (greedy stays ahead)**.** *For all $j \in \{0, \ldots, k^* - 1\}$, we have:*

1. *$j < k$, i.e. the Greedy Algorithm schedules at least $j + 1$ intervals, and*
2. *$b_{i_j} \leq b_{i_j^*}$, i.e. the j'th interval scheduled by the Greedy algorithm ends no later than the j'th interval scheduled by the optimal solution.*

Proof. For the $j = 0$ base case, since greedy always picks the absolute first interval by end time, the claim follows. Then assuming it holds up to j, we have $b_{i_j} \leq b_{i_j^*} < a_{i_{j+1}^*}$. The second inequality follows since the next interval in the optimal solution must start after the prior interval ending. But this means that interval i^*_{j+1} is *available* to the greedy algorithm after it has picked interval i_j, and since we would only not pick it if there is an available interval ending even earlier, we establish the claim for $j+1$ and conclude. □

Then from this claim we establish that $k^* - 1 < k$ and so the Greedy Algorithm schedules $k \geq k^*$ intervals. Since k^* is the optimal (maximum) number of intervals that can be scheduled, we conclude that $k = k^*$ and the Greedy Algorithm schedules an optimal number of intervals.

For the runtime, we can order the intervals by increasing end time by sorting in time $O(n \log n)$. Next we observe that in Line 5 we only need to check that the start time a_i of the current interval is later than the end time of b_j of the most recently scheduled interval (since all others have earlier end time), so we can carry out this check in constant time. Thus the loop can be implemented in time $O(n)$, for a total runtime of $O(n \log n) + O(n) = O(n \log n)$. □