CS120: Intro. to Algorithms and their Limitations	Hesterberg & Vadhan

Lecture 12: Independent Sets

Harvard SEAS - Fall 2022 — *2022-10-13*

1 Announcements

Recommended Reading: CLRS Sec 16.1–16.2

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**
    ;
    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.

□

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 $a_i \le b_i$
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 $a_i \le b_i$
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?

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**
    ;
    return S

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

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

Proof.

□