CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 13: Independent Sets

Harvard SEAS - Fall 2025 *2025-10-13*

1 Announcements

- Midterm in class this Thursday 10/15. Closed book. Bring black pen. Arrive early. Relax and get a good night's sleep!
- Next SRE Tuesday 10/21.

Recommended Reading:

- Hesterberg–Vadhan 13.2, 14
- CLRS 16.1

2 Loose Ends: Greedy Coloring

Definition 2.1. For an undirected graph $G = (V, E)$, a *k-coloring* of G is a mapping $f : V \to [k]$ such that for all edges $\{u, v\} \in E$, we have $f(u) \neq f(v)$.

The computational problem of finding a coloring is called the Graph Coloring problem:

Input: A graph $G = (V, E)$ and a natural number k

Output: A k-coloring of G (if one exists)

Computational Problem Graph Coloring

Remark. A common variant of the Graph Coloring problem is to find a coloring using as *few* colors as possible, for a given a graph G. This problem is in some sense the opposite of another problem we recently looked at:

3 Greedy Coloring

A natural first attempt at graph coloring is to use a *greedy* strategy (in general, a *greedy* algorithm is one that makes a sequence of myopic decisions, without regard to what choices will need to be made in the future):

```
   GreedyColoring(G):
   Input      : A graph G = (V, E)
   Output     : A coloring f of G using "few" colors
0  Select an ordering v_0, v_1, v_2, ..., v_{n-1} of V;
1  foreach i = 0 to n − 1 do
2  |  f(v_i) = _____.
3  return f
```

Algorithm 3.1: A greedy coloring algorithm.

An example of this algorithm is depicted in Figure 1.

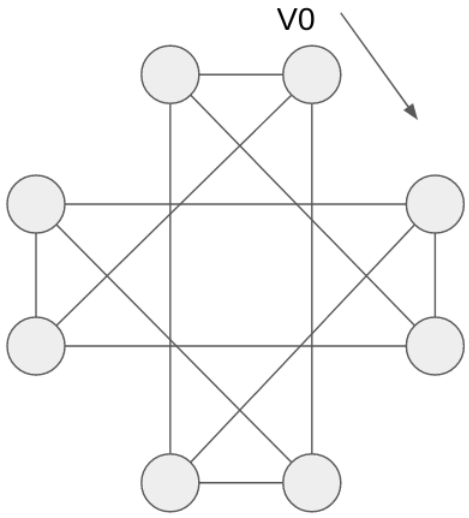


Figure 1: `GreedyColoring` algorithm with the ordering of vertices in clockwise direction starting from v_0

Assuming that we select the ordering (Line 0) in a straightforward manner (e.g. in the same order that the vertices are given in the input), `GreedyColoring`(G) can be implemented in time $O(n+m)$.

By inspection, `GreedyColoring`(G) always outputs a proper coloring of G. What can we prove about how many colors it uses?

Theorem 3.1. *When run on a graph $G = (V, E)$ with any ordering of vertices,* `GreedyColoring`(G) *will use at most* $\deg_{max}+1$ *colors, where* $\deg_{max} = \max\{\deg(v) : v \in V\}$.

Proof.

□

Remark. Note that this is an algorithmic proof of a pure graph theory fact: every graph is $(\deg_{max} + 1)$-colorable. However, this bound of $\deg_{max} + 1$ can be much larger than the number of colors actually needed to color G.

The performance of greedy algorithms can be very sensitive to the order in which decisions are made, and often we can achieve much better performance by picking a careful ordering. For example, we can process the vertices in *BFS order*, as described below.

```
   BFSColoring(G):
   Input          : A connected graph G = (V, E)
   Output         : A coloring f of G using "few" colors
0  Fix an arbitrary start vertex v_0 ∈ V;
1  Start breadth-first search from v_0 to obtain a vertex order v_1, v_2, ..., v_{n-1};
2  foreach i = 0 to n − 1 do
3  |  f(v_i) = min {c ∈ ℕ : c ≠ f(v_j)  ∀j < i s.t. {v_i, v_j} ∈ E}.
4  return f
```

Algorithm 3.2: Greedy coloring in BFS order.

Running the `BFSColoring` algorithm on the same example as above, we obtain

Now, we show that, in general, for 2-colorable graphs, `BFSColoring` finds a

2-coloring efficiently.

Theorem 3.2. *If G is a connected 2-colorable graph, then* `BFSColoring`(G) *will color G using 2 colors.*

Proof sketch.

□

Corollary 3.3. Graph 2-Coloring *can be solved in time $O(n + m)$.*

Proof.

□

4 Independent Sets: Definition

Last time, we modelled the problem of Register Allocation using a *conflict graph* and this led us to study the computational problem of Graph Coloring. Another problem to which this approach can be naturally applied is a variant of the IntervalScheduling–Decision problem introduced in earlier in the course:

Input: A collection of intervals $[a_0, b_0], \ldots, [a_{n-1}, b_{n-1}]$, where for all i, $a_i, b_i \in \mathbb{Q}$ and $a_i \leq b_i$

Output: YES if the intervals are disjoint from each other (for all $i \neq j$, $[a_i, b_i] \cap [a_j, b_j] = \emptyset$)
NO otherwise

Computational Problem IntervalScheduling–Decision

(Here we have modified the problem to have the endpoints a_i, b_i be rational numbers, since our computational models cannot represent arbitrary real numbers.)

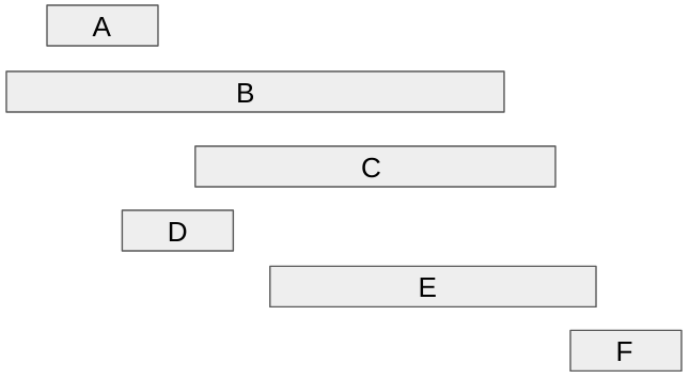
We saw that we could solve this problem in time $O(n \log n)$ by reduction to SORTING. However, if the answer is NO, we might be satisfied by trying to schedule *as many* intervals *as possible*:

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

Let's see how we can reduce this to a graph-theoretic problem. For example, consider the following set of intervals:



To model this graph-theoretically, we can

A set of intervals that do not intersect correspond to a set of vertices that do not have edges between them.

Definition 4.1. 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 $v \notin S$.

Finding the largest set of conflict-free intervals thus amounts to finding the largest independent set in the corresponding conflict graph.

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

Computational Problem INDEPENDENT SET

Another example of what is captured by the INDEPENDENT SET problem is coming up with as long an invitation list as possible for a party with the constraint that all attendees get along with each other.

5 Greedy Algorithms

Unfortunately, like GRAPH 3-COLORING, the fastest known algorithms for INDEPENDENT SET run in exponential time (though there are ones significantly better than exhaustive search, as you will see in the next SRE). For efficient algorithms, it is natural to try a greedy strategy similarly to `GreedyColoring` (Algorithm 3.1). Such an approach is described in Algorithm 5.1.

```
   GreedyIndSet(G):
   Input      : A graph G = (V, E)
   Output     : A "large" independent set in G
0  Choose an ordering v_0, v_1, v_2, ..., v_{n-1} of V;
1  S = ∅;
2  foreach i = 0 to n − 1 do
3  |                                        ;
4  |  return S
```

Algorithm 5.1: `GreedyIndSet()`

For simple vertex orderings (like the order the vertices appear in the input or BFS order), `GreedyIntervalScheduling` can be implemented in time $O(n + m)$. How large is the independent set it finds? Similarly to coloring (Theorem 3.1), we can prove only fairly weak bounds on its performance in general:

Theorem 5.1. *For every graph G with n vertices and m edges, and every vertex ordering,* `GreedyIndSet(G)` *outputs an independent set of size at least* $n/(\deg_{max}+1)$*, where* $\deg_{max}$ *is the maximum vertex degree in* G*.*

The guarantee of Theorem 5.1 can be quite weak compared to the size of the largest independent set in G. However, we can do much better for IntervalScheduling–Optimization, and find the true largest independent set. The point is that the conflict graphs that arise from collections of intervals are much more structured than arbitrary graphs.

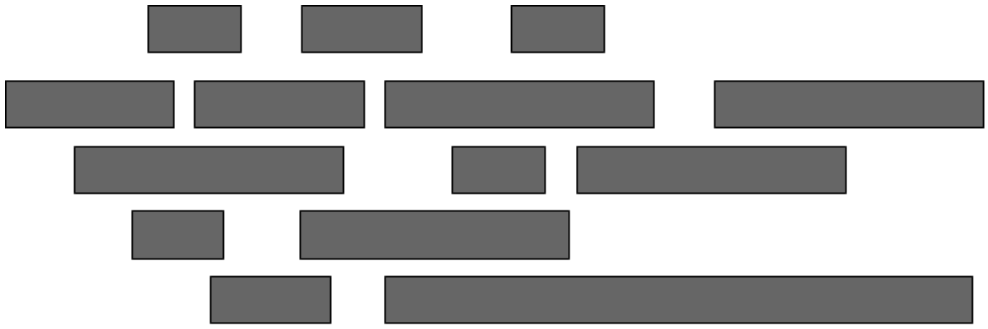
Specifically, a nearly linear-time greedy algorithm for IntervalScheduling–Optimization is given in Algorithm 5.2.

```
GreedyIntervalScheduling(x):
Input  : A list x of n intervals [a, b], with a, b ∈ ℚ
Output : A "large" subset of the input intervals that are disjoint
         from each other
0 Sort the intervals with                                   ;
1 S = ∅;
2 foreach i = 0 to n − 1 do
3 |                                                         ;
4 |  return S
```

Algorithm 5.2: `GreedyIntervalScheduling()`

Theorem 5.2. *For every input list x of intervals,* `GreedyIntervalScheduling(x)` *finds an optimal solution to* IntervalScheduling–Optimization *in time* $O(n \log n)$*.*

Proof. For correctness, we will show that for any IntervalScheduling–Optimization instance (such as the one shown in black in below) and any solution $S^* = \{i_0^* \leq i_1^* \leq \ldots \leq i_{k^*-1}^*\}$ (where $i_j^* \in S^*$ means that we include the interval $[a'_{i_j^*}, b'_{i_j^*}]$), we can modify S^* into the greedy solution $S = \{i_0, i_1, \ldots, i_{k-1}\}$ by "smushing it left."



Induction Hypothesis. Formally, we can prove the following by induction on $j = 0, 1, \ldots, k^* - 1$:

1. The greedy solution contains at least $j + 1$ intervals.
2. $b'_{i_j} \leq b'_{i^*_j}$, i.e. the j^{th} interval selected by the greedy algorithm ends no later than the j^{th} interval in the optimal solution ("greedy stays ahead").

Setting $j = k^* - 1$ in Item 1, we see that `GreedyIntervalScheduling` finds a solution containing at least k^* intervals. That is, for every independent set S^*, `GreedyIntervalScheduling` finds an independent set with at least as many intervals as S^*. Thus, `GreedyIntervalScheduling` finds an optimal solution.

Runtime.

□