CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 12: Interval Scheduling, Independent Sets and Matching

Harvard SEAS - Fall 2024 *2024-10-10*

1 Announcements

Recommended Reading: CLRS Sec 16.1–16.2

- Mid term Oct-17
- SRE 4 next Tuesday Oct-15.

2 Interval Scheduling

In Lecture 11, we modelled the conflicts arising in RAM allocation as a Coloring problem and discussed greedy algorithms for the problem. We found that the greedy - but careful - way of coloring via BFS worked well for 2-colorable graphs.

Another example where the greedy approach to algorithm design works well is Interval Scheduling. The following decision version of Interval Scheduling was introduced in Lecture 4.

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	**:** 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

A greedy algorithm for IntervalScheduling-Optimization is as follows.

```
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
Sort the intervals with increasing ordering of end time, to obtain
  [a'_0, b'_0], [a'_1, b'_1], ..., [a'_{n-1}, b'_{n-1}];
S = ∅;
foreach i = 0 to n − 1 do
    if ∀j < i s.t. j ∈ S  we have [a'_j, b'_j] ∩ [a'_i, b'_i] = ∅ then S = S ∪ {i};
return S
```

Theorem 2.1. `GreedyIntervalScheduling(`x`)` *finds an optimal solution to IntervalScheduling-Optimization, and can be implemented in time* $O(n \log n)$.

Proof.
Intuitively, for any interval scheduling problem (black in Figure 1), we can modify any solution $(i_0^*, i_1^*, \ldots, i_{\ell-1}^*)$ to it (gray boxes) into the greedy solution $(i_0, i_1, \ldots, i_{k-1})$ (white) by "smushing it left", replacing the first j intervals of our solution with the first j intervals of the greedy solution to get a valid solution $(i_0, i_1, \ldots, i_{j-1}, i_j^*, \ldots, i_{\ell-1}^*)$. Also, $k \geq \ell$: If not, then Greedy would pick one more interval, since i_ℓ^* would be valid.

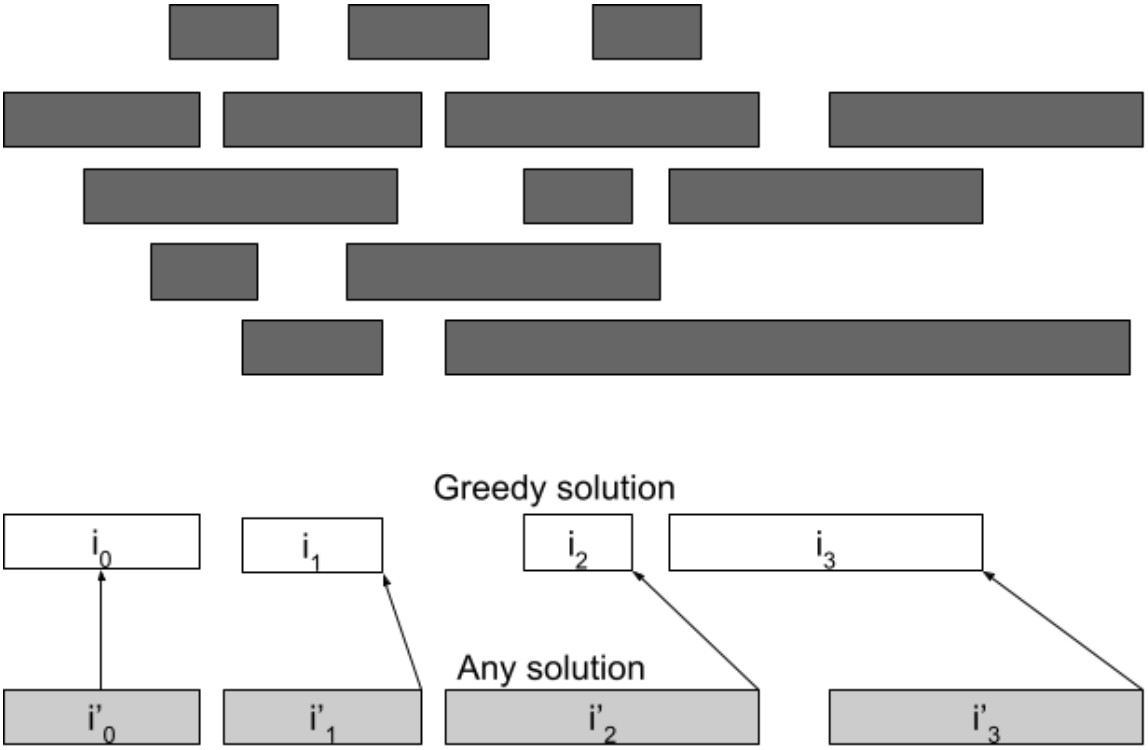


Figure 1: Transforming any interval scheduling solution into the greedy one.

Formally, let $S^* = \{i_0^* \leq i_1^* \leq \ldots \leq i_{k^*-1}^*\}$ be an optimal solution to Interval Scheduling (where we say that $i < i'$ for intervals i and i' if i ends before i' begins). 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).

We proceed via an inductive argument. Consider the following inductive hypothesis (for $j \in \{0, \ldots, k^* - 1\}$):

(Greedy stays ahead):

1. 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.

Base case: For the $j = 0$, the hypothesis holds since the greedy algorithm always picks the absolute first interval by end time.

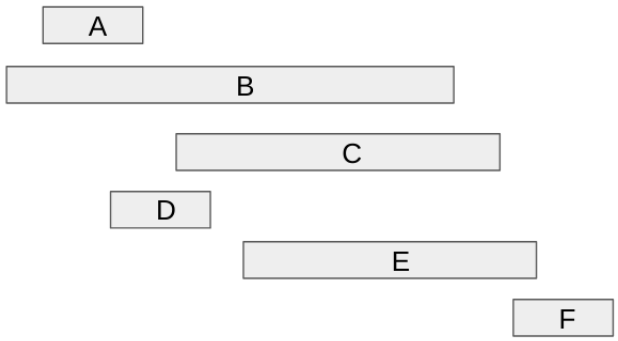
Inductive argument: Assuming the hypothesis holds up to j, we have $b'_{i_j} \leq b'_{i^*_j} < a'_{i^*_{j+1}}$. The second inequality follows since the next interval in the optimal solution must start after the prior interval ending. But this means that interval i^*_{j+1} is *available* to the greedy 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.

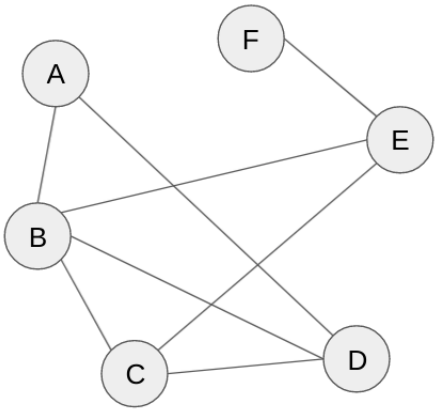
This establishes the inductive hypothesis. Using it for $j = k^* - 1$, we find that $k^* \leq k$ and so the Greedy Algorithm schedules at least k^* intervals. Since k^* is the optimal (maximum) number of intervals that can be scheduled, we conclude that $k = k^*$ and the Greedy Algorithm schedules an optimal number of intervals.

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 $O(n \log n) + O(n) = O(n \log n)$. □

3 Independent Set

In Lecture 11, we used graph theoretic modelling on RAM allocation to rephrase it as a Coloring problem. Graph theoretic modelling can also be applied to the IntervalScheduling-Optimization problem. For example, consider the following set of intervals:





We can represent each interval as a vertex, and we place an edge between two vertices (i.e. intervals) if they conflict. A set of intervals that do not conflict correspond to a set of vertices that do not have edges between them.

This is the notion of an independent set, which is defined as follows for a general graph.

Definition 3.1. 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 becomes equivalent to finding the largest number of vertices with no edges between them. For a general graph, the analogous problem is known as the Independent Set 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.
Remarks:

- **Independent Set vs IntervalScheduling-Decision:** Independent Set problem is more general than IntervalScheduling-Decision. While any interval scheduling problem can be modelled as an independent set problem on some graph (as discussed above), one cannot take an independent set problem on an arbitrary graph and model it as an interval scheduling problem (for example, a cycle).
- **Independent Set vs Coloring:** Independent Set is closely related to Coloring. In a proper k-Coloring of a graph G, the vertices of certain color do not have edge connecting them. Such a collection of vertices form an independent set. In the next Sender-Receiver exercise, you will see further implications of this connection.

There are no known efficient algorithms for the Independent Set problem (we will discuss this more in future lectures). However, a greedy algorithm along the lines of `GreedyIntervalScheduling` and `GreedyColoring` can be designed which outputs a somewhat large independent set.

```
1 GreedyIndSet(G)
  Input      : A graph G = (V, E)
  Output     : A "large" independent set in G
2 Choose an ordering v_0, v_1, v_2, ..., v_{n-1} of V;
3 S = ∅;
4 foreach i = 0 to n − 1 do
5 |  if ∀j < i s.t. {v_i, v_j} ∈ E we have v_j ∉ S then S = S ∪ {v_i};
6 return S
```

How large is the independent set in the above algorithm? Similarly to coloring, we can only prove fairly weak bounds on the performance of the greedy algorithm in general:

Theorem 3.2. *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 $n/(\deg_{max} + 1)$, where $\deg_{max}$ is the maximum vertex degree in G.*

Proof. We prove this via contradiction. Suppose the size of the independent set output by the algorithm is $K < n/(\deg_{max} + 1)$. The number of vertices either in this independent set or having an edge with a vertex in this independent set are $K(\deg_{max} + 1) < n$, thus at least one vertex - denoted v_{ell} - does not have this property. Considering the execution of Line 5 for v_ℓ, we see that it must have been added to the independent set, leading to size $K + 1$, which is a contradiction. □

4 Matching

While Independent Set is a graph theoretic notion about "non-conflicting" vertices, matching is about "non-conflicting" edges.

Definition 4.1. For a graph $G = (V, E)$, a *matching* in G is a subset $M \subseteq E$ such that every vertex $v \in V$ is incident to at most one edge in M[1].Equivalently, no two edges in M share an endpoint.

The problem of finding the largest matching in a graph is called Maximum Matching.

Input	: A graph $G = (V, E)$
Output	: A matching $M \subseteq E$ in G of maximum size

Computational Problem Maximum Matching

An example of maximum matching is depicted in Figure 2.

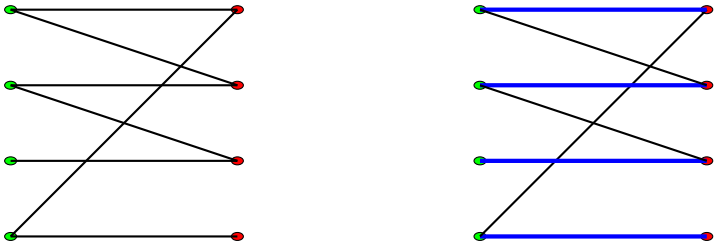


Figure 2: A graph (left) and a maximum matching (right)

We can use a greedy strategy to try finding a maximum matching. We can start with an edge and increase the size of the matching by adding edges (that do not share a vertex with edges that have already been added) till we can no longer add edges. An example is depicted in Figure 3.

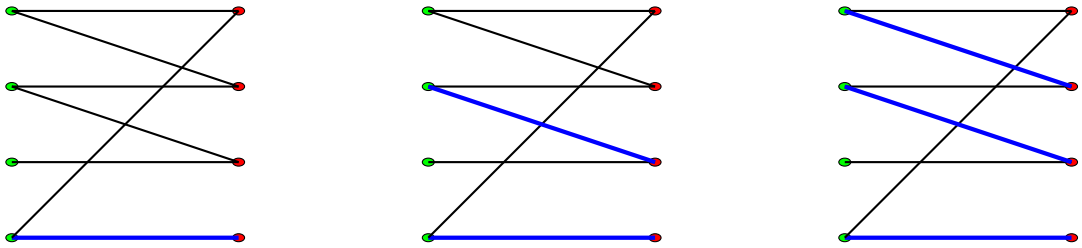


Figure 3: Greedily choose edges may lead to a matching of smaller size than the maximum matching.

The example in Figure 3 leads to a matching of size 3, which is smaller than the maximum matching of size 4 in Figure 2. Furthermore, it is not possible to add an edge to further increase the size of the matching.

Remarkably, sometimes it is possible to do more sophisticated operations than just adding an edge and still grow the matching. For this, we need the notions of alternating walk and augmenting path.

Definition 4.2. Let $G = (V, E)$ be a graph, and M be a matching in G. Then:

1. An *alternating walk* W in G with respect to M is a walk $(v_0, v_1, \ldots, v_\ell)$ in G such that for every $i = 1, \ldots, \ell - 1$, $\{v_{i-1}, v_i\} \in M \Leftrightarrow \{v_i, v_{i+1}\} \in E \setminus M$.

[1]Saying a vertex v is *incident* to an edge e is another way of saying v is an endpoint of e. It is more symmetric, in that we would also say that e is incident to v.

2. An *augmenting path* P in G with respect to M is an alternating walk in which v_0 and v_ℓ are respectively unmatched by M, and in which all of the vertices in the walk are distinct, and $\ell \geq 1$.

An example of alternating walk is depicted in Figure 4, which is also an augmenting path. We can further observe that the set of edges appearing in this augmenting path is precisely the *set difference* between the matching on the left hand side of Figure 4 and the maximum matching in Figure 2. Thus, intuitively, the augmenting path tells us which edges to 'switch' to go from the matching of size 3 to the matching of size 4.

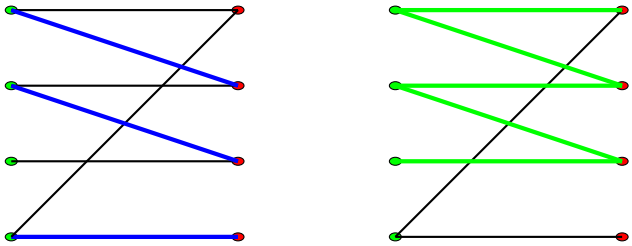


Figure 4: For the matching of size 3 on the left, an alternating walk on the right. This is also an augmenting path, since all the vertices are distinct and the first and the last vertices are not in the matching.

This suggests a natural algorithm for maximum matching: repeatedly try to find an augmenting path and use it to grow our matching. But we need to argue that augmenting paths always exist and we can find them efficiently. An important piece in this argument is the following theorem.

Theorem 4.3 (Berge's Theorem)**.** *Let $G = (V, E)$ be a graph, and $M \subseteq E$ be a matching. If (and only if) M is not a maximum-size matching, then G has an augmenting path with respect to M.*

In the next Lecture, we will see how to turn the above idea into an algorithm for the case of *bipartite* graphs.

Definition 4.4. A graph $G = (V, E)$ is bipartite if it is 2-colorable. That is, there is a partition of vertices $V = V_0 \cup V_1$ (with $V_0 \cap V_1 = \emptyset$) such that all edges in E have one endpoint in V_0 and one endpoint in V_1.

The example in Figure 2 is a bipartite graph.