Algorithms Fall 2019

Lecture : Interval Scheduling and Coloring

Imdad Ullah Khan

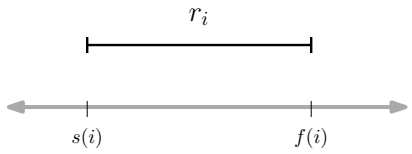
Contents

1 Interval Scheduling (Activity Selection) 1

2 Interval Coloring (Interval Partitioning): 5

1 Interval Scheduling (Activity Selection)

Suppose we are given a set $\mathcal{R}$ of n requests : $\{r_1, r_2, \ldots, r_n\}$ for scheduling on a single resource. The i^{th} request r_i is described by its start and finish time denoted by $s(i)$ and $f(i)$ respectively. Duration $d(i)$ of a request i is $f(i) - s(i)$ where $s(i) < f(i)$ is implicitly given.



This situation appears in many practical scenarios, e.g. classes/exams scheduling, NADRA servicing windows where each request takes a certain duration, and in parallel processing.

We visualize each request as an interval on the time line. Two requests r_i and r_j are said to be compatible if they do not overlap in any interval of time. Otherwise, they are said to be conflicting. Given two requests r_i and r_j we can check in constant time whether they are compatible. It is easy to see that a pair of requests r_i and r_j is compatible if and only if

$$s(i) < f(i) < s(j) < f(j) \quad \text{or} \quad s(j) < f(j) < s(i) < f(i), \tag{1}$$

i.e. either r_i comes completely before r_j or vice-versa.

Set-wise compatible: A set of requests that are all pairwise compatible.

Problem 1 (Interval Scheduling)**.** *Given a set of requests $\mathcal{R}$, find the largest subset of compatible requests.*

The selected subset is required to be compatible so that they can be scheduled on a single resource.

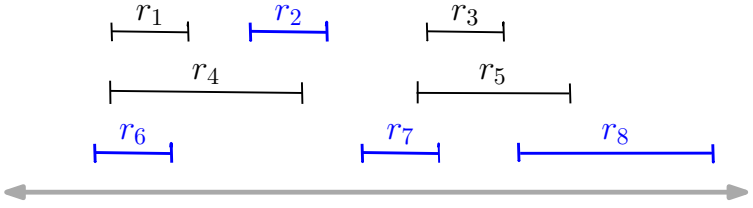


Figure 1: r_1 and r_2 are compatible, r_4 and r_8 are compatible whereas r_1 and r_4 are conflicting and r_5 and r_7 are conflicting. Set $\{r_1, r_2, r_8\}$ is compatible whereas $\{r_1, r_2, r_5, r_8\}$ is not compatible. The largest compatible subset is $\{r_6, r_2, r_7, r_8\}$.

Greedy algorithm for Interval Scheduling

A natural greedy algorithm for interval scheduling problem is to process requests in some fixed order by selecting a request r_i from $\mathcal{R}$ greedily and deleting all other requests conflicting with r_i. This process is repeated until no requests remain.

Algorithm Interval Scheduling ($\mathcal{R}$)

$A \leftarrow \emptyset$
while $\mathcal{R} \neq \phi$ **do**
  select a request r_x from $\mathcal{R}$
  remove from $\mathcal{R}$ all those requests conflicting with r_x
  $A \leftarrow A \cup \{r_x\}$
return A

Clearly, if the algorithm selects r_i, then it cannot select any request that is conflicting with r_i. Therefore, this algorithm is correct by design. The optimality of the algorithm, i.e. whether it outputs the largest compatible subset, depends on the greedy approach for selecting r_i, i.e. in what order should the requests be processed. We discuss a few selection rules below.

Selection Rule 1: Earliest Starting Time First

Select r_i that has the smallest $s(i)$. The idea is to start working as early as possible. Figure 2 provides a counterexample to prove that this greedy approach is not optimal.

Figure 2: Earliest Starting Time First: The algorithm's selection (left) is only r_1 as it starts earliest and the rest are conflicting with r_1, while the optimal solution (right) is the subset $\{r_2, \ldots, r_5\}$ of 4.

Selection Rule 2: Latest Finishing Time First

Select r_i that has the largest $f(i)$. The idea of this approach is similar to earliest starting time, to keep using resources until as late as possible. Figure 3 provides a counterexample to prove that this greedy approach is not optimal either. The problem with this approach is same as above, on the example in Figure **??** it selects the r_1, while the optimal solution again is selecting $r_2, \ldots, r_5$.

Figure 3: Latest Finishing Time First: The algorithm's selection (left) is the same as above, 1 request i.e. r_1, whereas the optimal solution is the subset $\{r_2, \ldots, r_5\}$ of 4

Selection Rule 3: Shortest Request First

Both of the above selection rules ignore duration of the requests. The next attempt is to select the request with smallest duration, i.e. a request for which $d(i) = f(i)$ - $s(i)$ is as small as possible. Despite seeming better than the previous approaches, this one too can produce a suboptimal result, as shown in Figure 4

Figure 4: Shortest Request First: The algorithm's selection (left) is just 1 request r_2 whereas the optimal solution is $\{r_1, r_3\}$

Selection Rule 4: Least Conflicting (Most Compatible) First

The above approaches did not consider the number of conflicting request during selection of request. The next attempt is to greedily select a request with a minimum number of conflicting requests. However, this approach is not optimal either, as shown in Figure 5

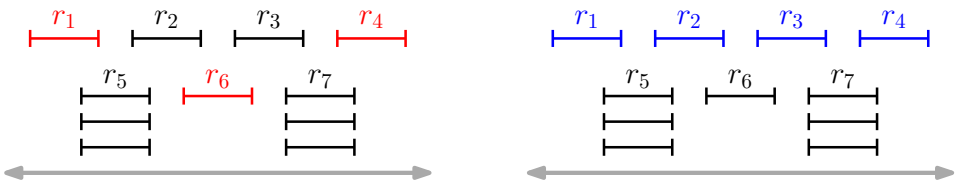


Figure 5: Least Conflicting First: The algorithm first selects r_6 as it has 2 conflicts while all other requests have at least 3 conflicts. Hence after selecting r_6, the algorithm can only select r_1 and r_4 or some other combination of two requests other than r_2 and r_3. However, the optimal solution is $\{r_1, r_2, r_3, r_4\}$.

Selection Rule 5: Earliest Finishing Time First

Select r_i with smallest $f(i)$. This approach is based on the intuition that the resourced should be freed as early as possible for other requests and produces the optimal result for all the examples we have seen so far in Figures 3, 4 and 5. However, it is not enough to show that this approach works for a few numbered examples. It must be proven formally that the following earliest finishing time first greedy interval scheduling algorithm selects the largest compatible subset for all inputs.

Algorithm Earliest Finishing Time First Interval Scheduling ($\mathcal{R}$)

$A \leftarrow \emptyset$
while $\mathcal{R} \neq \emptyset$ **do**
  Select the request r_x with smallest finishing time from $\mathcal{R}$
  Remove from $\mathcal{R}$ all those requests conflicting with r_x
  $A \leftarrow A \cup \{r_x\}$
return A

Proof of Correctness:

The algorithm is correct by design, as in each iteration, only a request compatible with the selected set A so far A is added to A. If the request r_i added to A conflicts with another previously added request r_j, then r_i would have been deleted when r_j was added to A according to the algorithm, which is a contradiction.

Proof of Optimality:

Let O be an optimal solution, should we prove that $A = O$? There may be more than one optimal solution and at best A is equal to a single one of them. Therefore, we need to just prove that $|A| = |O|$, i.e. A contains the same number of requests as O.

Let $i_1, i_2, \ldots, i_k$ be the set of requests in A in the order they were added to A and $|A| = k$. Since the request were added in order of increasing finishing time, it can be said that A is sorted by finishing time.

Similarly, let the set of requests in O be denoted by $j_1, j_2, \ldots, j_k$, in the increasing order of finishing time and $|O| = m$.

The goal is to prove that $k = m$. Note that the requests in O and A are compatible, which implies that the starting time have the same order as the finish time. See (1).

The greedy strategy guarantees that $f(i_1) \leq f(j_1)$, as all requests are available to the algorithm in the beginning and the one with the earliest finishing time is selected. We need to show that the greedy selection 'stays ahead' i.e. each of its interval finishes at least as soon as the corresponding interval in O. This follows from the intuition that the algorithm stays ahead by releasing the resource as early as possible, Thus, we now prove that for each $r \geq 1$, the r^{th} accepted request in the algorithm's schedule finishes no later than the r^{th} request in optimal schedule.

Lemma 1. *For* $1 \leq r \leq k$

$$f(i_r) \leq f(j_r).$$

Note by the optimality of O, we have that $k \leq m$. Also note that this statement is made for $1 \leq r \leq k$. It would be wrong to state that for $1 \leq r \leq m$, $f(i_r) \leq f(j_r)$ as if $k < m$ then $f(i_m)$ would be undefined.

Proof. We prove this statement by induction on r.

Basis Step: For $r = 1$ the statement is true, since the algorithm starts by selecting the request i_1 with minimum finish time, i.e: $f(i_1) \leq f(j_1)$

Inductive Hypothesis: If $f(i_{r-1}) \leq f(j_{r-1})$*, then* $f(i_r) \leq f(j_r)$.

Induction Step: It is know that $f(i_{r-1}) \leq s(j_r)$ because by induction hypothesis, $f(i_{r-1}) < f(j_{r-1})$ and by compatibility of O, $f(j_{r-1}) < s(j_r)$. Thus, the j_r interval is in the set $\mathcal{R}$ of available requests at the time when the greedy algorithm selects i_r interval. Since the algorithm selects the available request with smallest finish time, if $f(j_r) < f(i_r)$, then the algorithm would have selected $f(j_r)$. Therefore, we have that $f(i_r) \leq f(j_r)$. This completes the induction step.

□

Theorem 2.

$$m = k$$

Proof. We prove this statement by contradiction. Assume that $m > k$. By the above lemma, $f(i_k) \leq f(j_k)$. Since $m > k$, there must be a request $(j_k + 1) \in O$. By the compatibility of O, $s(j_k + 1) \geq f(j_k) \geq f(i_k)$. Hence, the request $(j_k + 1)$ is compatible with A and available for selection (it has not been selected yet), i.e. and $\mathcal{R}$ is not empty. However, the greedy algorithm stops with request (i_k), and it is only supposed to stop when $\mathcal{R}$ is empty which is a contradiction. Therefore, since it can not be the case that $m > k$, and it is already known that $k \leq m$, $k = m$. □

Implementation and Runtime:

A naive implementation of this algorithm takes $O(n)$ time in every iteration for finding request with minimum finishing time and deleting incompatible requests. Incompatible intervals can be found by traversing the remaining intervals and testing compatibility of each by (1). Hence, the overall runtime is $O(n^2)$.

Runtime can be improved by initially sorting all requests by finishing time in $O(n \log n)$ time. Imagine the input requests as an array of two values, starting and finishing time, similar to a $2 \times n$ matrix. Sort by finishing time values while dragging along the start time. Keep a marker at current boundary of $\mathcal{R}$, i.e. the first 'non-deleted' request in R. Once a request r_i is selected, traverse through the array starting from

the current marker and increment the marker as long as $s(j) \leq f(i)$, i.e. move the marker over the request incompatible with the just added r_i and consider them deleted as the next selection is a request from the marker onwards. This reduces the runtime to a linear scan in $O(n)$ with the sorting as an initialization step. Hence, the overall runtime is $O(n \log n)$.

2 Interval Coloring (Interval Partitioning):

Interval Coloring is another version of the Interval Scheduling problem in which all intervals must be scheduled while minimizing the number of resources used. In other words, the goal is to partition the set of intervals into subsets such that each subset contains only compatible intervals and the number of subsets is as small as possible. This problem has many applications, for example, to find the minimum number of classrooms to schedule all lectures or exams (given as intervals) so that no two classes occur at the same time in the same room. Other applications are in multi-processor systems, service windows etc.

Considering the classrooms example, n rooms can be used to schedule all intervals (each request is in a separate subset). This is evidently a correct solution. Hence the problem is well defined, i.e. a solution exists. However, this is a trivial solution. The problem becomes interesting if the number of classrooms used is to be minimized, i.e. find the minimum number of subsets. In order to see why this problem is called Interval Coloring problem, imagine the resources as colors and each interval must be assigned a color. Similarly, it is also called the Interval Partitioning problem as it requires that the interval set is partitioned into subsets.

Problem 2 (Interval Coloring)**.** *Given a set of requests $\mathcal{R}$, find a partition of $\mathcal{R}$ with minimum number of compatible subsets.*

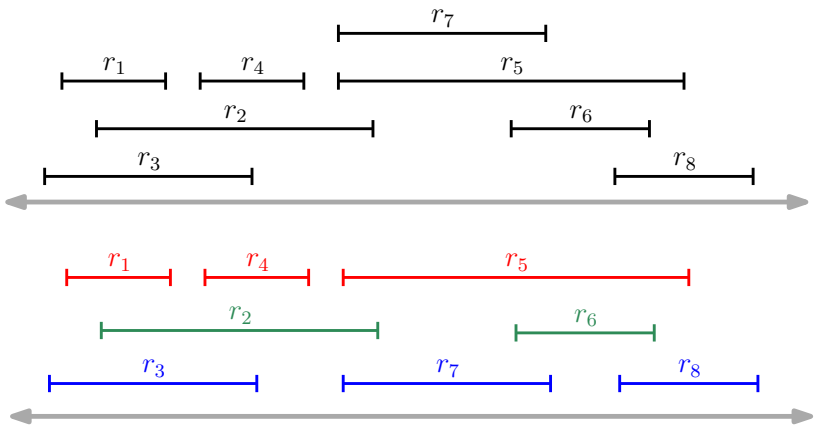


Figure 6: An instance of the interval coloring problem (top) with an input set of 8 intervals. A solution (bottom) is to partition intervals as follows: $\{r_1, r_4, r_5\}$ colored red, $\{r_2, r_6\}$ colored green and $\{r_3, r_7, r_8\}$ colored blue. Is it possible to use only two resources in this example? Clearly not. There is a point in time where at maximum 3 intervals are intersecting, therefore, at least 3 partitions are required to color all intervals. In other words, 3 is a lower bound on the number of colors needed in this example.

Definition 3 (Depth of a set of intervals:)**.** *Depth of a set of intervals is the largest number of intervals passing through a point in time.*

All intervals passing through the deepest point (depth realizing point) must be scheduled on different resources. Hence, in general, the depth d of a set $\mathcal{R}$ is a lower bound on the number of resources needed to color all intervals.

Let $\mathcal{R}_p(r_j)$ be set of intervals preceding r_j with $s(i) \leq s(j)$ that are conflicting with r_j

---

Algorithm Interval Coloring $(\mathcal{R}, d)$

$C \leftarrow \{c_1, \ldots, c_d\}$
Let $r_1, r_2, \ldots, r_n$ be $\mathcal{R}$ sorted by $s(.)$
Let $\mathcal{R}_p(r_j)$ be set of intervals with $s(i) \leq s(j)$, i.e. preceding r_j, which conflict with r_j
for $j = 1$ to n **do**
    $C' \leftarrow C \setminus \{$colors used for any $r_i \in \mathcal{R}_p(r_j)\}$
    if $C' \neq \emptyset$ **then**
        color r_j with a $c_l \in C'$
    else
        Leave r_j uncolored

Proof of Correctness:

Proof. In order to prove that each request in $\mathcal{R}$ is assigned a non-conflicting color, we first prove by contradiction that every interval does get a color, i.e. that the last line in the pseudocode is never executed, and next we prove, again by contradiction that there are no conflicting colors.

Suppose that some request r_j does not get any color. Suppose there are k other intervals before r_j that overlap with r_j. Each one of them must have start time before $s(j)$ (as they must be earlier because we are working in sorted order) and their finishing time is either before $f(j)$ or after $f(j)$. All of them have the point $s(j)$ in common, so these $k+1$ (including r_j) requests are conflicting, therefore, $k + 1 \leq d \implies k \leq d - 1$. These k requests consume at most $d - 1$ colors, hence one of the colors must be unused when r_j is considered and is available for coloring r_j, a contradiction to the statement that r_j is not colored.

Suppose r_i and r_j are two overlapping intervals that are assigned the same color. Assume without loss of generality that $i \leq j$. Then, the color assigned to r_i must have been excluded from consideration for r_j and can not be assigned the color assigned to r_i, a contradiction to the statement that both r_i and r_j are assigned the same color. □

Proof of Optimality:

Proof. The algorithm achieves the lower bound, i.e. atleast d colors are required, as it colors all intervals with at most d colors and therefore produces an optimal solution for any $\mathcal{R}$. □

Finding Depth of $\mathcal{R}$

The current algorithm assumes that the depth d of $\mathcal{R}$ is known in advance or given as an input. In fact, dinding the depth of $\mathcal{R}$ is equivalent to finding the number of colors needed to color $\mathcal{R}$. This is not very hard to see, the above proof has all the ideas to understand this concept.

A more general algorithm that does not assume knowledge of d in advance and outputs d along with an optimal coloring, is as follows. It allocates a new color on need basis, and proves that exactly d colors are allocated.

Algorithm Interval Coloring with Unknown Depth ($\mathcal{R}$)

```
d ← 1
while R ≠ ∅ do
    Choose r_i ∈ R such that s(i) is smallest
    if r_i can be colored by some color c ≤ d then
        Color r_i with color c
    else
        Allocate a new color d + 1
        d ← d + 1
        Color r_i with color d
return d
```

Implementation and Runtime:

As in interval scheduling, a naive implementation would find an unused color for each request. The overall runtime would be $O(n^2)$ as a linear scan is required per request.

A better implementation is as follows. Similar to interval scheduling, view the input intervals as a $2D$ array, each interval is a starting and finishing time. Sort intervals by starting time $s(i)$ as required by the algorithm. While considering r_j, in order to find all the requests conflicting with r_j, which are exactly those among the intervals preceding r_j whose finishing time is higher than $s(j)$, scan all of the preceding intervals. This again leads to $O(n^2)$ runtime. However, realize that finding all the intervals that conflict with r_j, even all the one preceding r_j, is not necessary. It is enough to only find an available color among the d colors allocated (this is true for both algorithms above), which can be done as follows.

Suppose for each resource c, $(1 \leq c \leq d)$ the latest finishing request that is colored with color c is maintained in an array A of length d indexed by color id such that $A[c]$ is the latest finishing request that was colored with c. Each time the color c is assigned to some interval, $A[c]$ is updated to the finishing time of the interval currently assigned c. A color c is available for r_j if the finishing time stored at $A[c]$ is smaller than $s(j)$, i.e. the latest interval using c finishes before r_j and therefore does not conflict with r_j. In this case, we assign c to r_j and update $A[c]$ to $f(r_j)$.

The runtime of this approach is $O(nd)$ time. However, since d in the worst case could be $O(n)$, the runtime, in the worst case, of this approach is also $O(n^2)$.

Improved runtime and implementation using a Min-Heap:

The problem in finding a color for r_j still lies in repeatedly finding an index in A where the finishing time is smaller than $s(j)$. A more suitable data structure can be used for this purpose instead of an array. Recall the priority queue data structure. Store the latest finishing times of each color in a min-heap H, where the items are the d colors and the corresponding key is the latest finishing time. Then, while processing r_j, usage of the color c with minimum latest finishing time returned by ExtractMin(H) differs slightly depending on whether the algorithm knows the depth d or not.

In the algorithm with known depth, the color c returned by ExtractMin(H) operation is allocated to r_j (the extracted color must be available as otherwise the lower bound d would be violated), followed by a IncreaseKey($H, c, f(j)$). In the algorithm with unknown depth, if c has finishing time less than $s(j)$ then it is assigned to r_j followed by IncreaseKey($H, c, f(j)$), else if $s(j)$ is smaller than the minimum finishing time of c, then no color is currently available for r_j and a new color c' is allocated for r_j followed by a Insert($H, c', f(j)$) operation.

The overall runtime is reduced to $O(n \log n + n \log d)$.