Interval scheduling

- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs are **compatible** if they don’t overlap.
- A **schedule** is the subset of jobs we run given in order.
- Task: Find a schedule with maximum *number* of compatible jobs.

![Diagram of job intervals on a timeline]

- a
- b
- c
- d
- e
- f
- g
- h
Interval scheduling

- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs are **compatible** if they don’t overlap.
- A **schedule** is the subset of jobs we run given in order.
- Task: Find a schedule with maximum *number* of compatible jobs.
Greedy algorithms

What makes an algorithm simple and efficient?

* run fast
* without recursive calls
* does not use too much memory
* easy to prove
* avoid unnecessary checks

Goal: find the maximum or minimum subset
→ expand the solution one step at a time
Greedy algorithms

When do we call an algorithm “greedy”?

• simple computations
• myopic, makes local decisions
  • decides based on computations on a small number of easy to obtain data.
  • doesn’t take global structure in to consideration
  • e.g. BFS/DFS picks an available but unexplored neighbor at random, interval scheduling takes the next compatible interval, etc.
• once a choice has been made, it’s never updated when more information is available

• In this class we will mostly look at greedy algorithms that find an optimal solution. Most greedy algorithms however are not optimal. Yet, we use them in real-life applications as they are pretty close to optimal and very efficient.
Greedy algorithms

Goal: Find a greedy algorithm for the interval scheduling problem

input: starting time $s_j$ and finishing time $f_j$ for each job $j$

return: a maximum compatible schedule

High level:

Consider jobs $j$ one at a time

• for each $j$ make a decision whether to include it in the schedule
• the decision is final

Input: sequence of $n$ jobs
Which sorting order will always yield a maximum schedule?

A. sort jobs by earliest start time first
B. sort jobs by earliest finish time first
C. sort jobs by shortest interval first
D. sort jobs by fewest conflicts (= number of jobs it intersects) first
Interval scheduling: greedy algorithms

Greedy template. Consider jobs in some natural order. Take each job provided it’s compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of $s_j$.
- Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each job $j$, count the number of conflicting jobs $c_j$. Schedule in ascending order of $c_j$. 
Interval scheduling: greedy algorithms - what doesn’t work

**Greedy template.** Consider jobs in some natural order. Take each job provided it’s compatible with the ones already taken.

- **counterexample for earliest start time**
- **counterexample for shortest interval**
- **counterexample for fewest conflicts**
Algorithm 1: EarliestFinishTimeFirst(j=1...n: s\textsubscript{j}, f\textsubscript{j})

/* s\textsubscript{j} and f\textsubscript{j} are the start and finish time of each job */
1 sorted\_jobs = mergeSort(f\textsubscript{1}, f\textsubscript{2}, ..., f\textsubscript{n})/* sort jobs by finish time */

2 S ← []/* schedule; list of compatible jobs */
3 for j in sorted\_jobs do
4    if is compatible(j, S) then
5        S.add(j);
6    Θ(?)
6 return S;

Θ(n log n)
Algorithm 1: EarliestFinishTimeFirst(j=1...n: \(s_j, f_j\))

/* \(s_j\) and \(f_j\) are the start and finish time of each job */
1 sorted_jobs = mergeSort(f_1, f_2, ..., f_n) /* sort jobs by finish time */

2 \(S \leftarrow [\] /* schedule; list of compatible jobs */

3 for \(j\) in sorted_jobs do
4     if is_compatible(j, S) then
5         S.add(j);
6 return S;

Question: How to implement is_compatible(j, S) in O(1) per job?

Compare the next candidate against the last job in the set \(S\).
Interval scheduling: earliest-finish-time-first algorithm

**Algorithm 1: EarlyFirstFinishTimeFirst (j=1...n: s_j, f_j)**

```plaintext
/* s_j and f_j are the start and finish time of each job */
sorted_jobs = mergeSort(f_1, f_2, ..., f_n) /* sort jobs by finish time */
S ← [] /* schedule; list of compatible jobs */
for j in sorted_jobs do
    if is_compatible(j, S) then
        S.add(j);
return S;
```

**Question:** How to implement `is_compatible(j, S)` in O(1) per job?

**Algorithm 1: is_compatible(j, S)**

```plaintext
/* j is a job id, S is the current schedule */
f_S ← finish time of S[-1] /* finish of last job in schedule */
if s_j > f_S then
    /* j starts after last job finishes */
return True;
return False
```
Interval scheduling: EFTF algorithm is correct

**Theorem.** The earliest-finish-time-first algorithm is optimal.

What does it mean that EFTF is optimal?

maximum amount of compatible jobs
Interval scheduling: EFTF algorithm is correct - TopHat

Theorem. The earliest-finish-time-first algorithm is optimal.

Which of these statements is correct?

A. There is always a unique optimal schedule. \[ \times \]

B. The number of jobs in the optimal schedule is unique. \[ \checkmark \]

C. If the EFTF schedule has at least as many jobs as the optimal schedule, then it is of max size. \[ \checkmark \]

D. The EFTF schedule cannot have more jobs than the optimal schedule. \[ \checkmark \]
Interval scheduling: EFTF algorithm is correct

Theorem. The earliest-finish-time-first algorithm is optimal.

(very) high level proof approach:

sometimes called “greedy stays ahead”

How to compare?

We swap jobs of the optimal solution with the jobs in the greedy until the optimal solution is identical to the greedy solution.
Interval scheduling: EFTF algorithm is correct

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [general template for proving optimization problems]

- “correct” here means optimal (it’s trivial that the schedule returned is compatible).
- it’s a maximization problem, hence optimal = maximum number
- there must exist an optimal solution (sometimes there are multiple)
  - (what is a brute force algorithm to find one?) \( \Theta(2^n) \)
- we compare the output of EFTF (which we don’t know whether it is optimal) with an optimal solution.
- if we can conclude in some clever way that the number of jobs in the output is equal to the number in the optimal solution, then EFTF is also optimal (maximal).
Interval scheduling: EFTF algorithm is correct

Compare the optimal solution to greedy solution.
Assume the jobs 1, 2, 3, ..., r are identical in the two schedules.
The job \( j_{r+1} \) in the optimal solution finishes later. Therefore, the job
\( i_{r+1} \) can fit in the time slot defined by \( j_{r+1} \), between \( j_0 \) and \( j_{r+2} \).
We can swap \( i_{r+1} \) and \( j_{r+1} \) without creating conflicts.
Continue comparing the schedules and repeat again once they differ.
Interval partitioning

- Lecture $j$ starts at $s_j$ and finishes at $f_j$.
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

**Ex.** This schedule uses 4 classrooms to schedule 10 lectures.
Interval partitioning

Interval partitioning.

- Lecture $j$ starts at $s_j$ and finishes at $f_j$.
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures.
Earliest Start Time First algorithm

### Time Schedule

- **9:30**
- **10:00**
- **10:30**
- **11:00**
- **11:30**
- **12:00**
- **12:30**
- **1:00**
- **1:30**
- **2:00**
- **2:30**
- **3:00**
- **3:30**
- **4:00**
- **4:30**

### Rooms
- **Room 1**
- **Room 2**
- **Room 3**
- **Room 4**
earliest-start-time-first algorithm

Algorithm 1: EarliestStartTimeFirst(\( j = 1 \ldots n : s_j, f_j \))

1. \( \text{A} \leftarrow \text{empty hash table} /\!* \text{A}[k] \text{ contains the list of courses assigned to room } k \*/ \)
2. \( \text{sorted\_class} = \text{sort}(s_1, \ldots, s_n) /\!* \text{sort by start time} \*/ \)
3. \( \text{for } c \text{ in sorted\_class do} \)
   4. \( \quad k = \text{find\_compatible\_room}(c, \text{A}, \ ) \)
   5. \( \quad \text{if } k \text{ is not None then} \)
   6. \( \quad \quad \text{A}[k].\text{add}(c); \)
   7. \( \quad \text{else} \)
   8. \( \quad \quad d \leftarrow \text{len(A)} /\!* \text{highest room id} \*/ \)
   9. \( \quad \quad \text{A}[d + 1] = [ ] /\!* \text{open new room} \*/ \)
10. \( \quad \text{A}[d + 1].\text{add}(c); \)
11. \( \text{return } \text{A} \)

Pre-processing \( \Theta(n, \log n) \)

\( \Theta(n) \)

\( \Theta(\log n) \)
earliest-start-time-first algorithm - running time

Algorithm 1: EarliestStartTimeFirst\((j = 1 \ldots n : s_j, f_j)\)

1. \(A \leftarrow \) empty hash table /* \(A[k]\) contains the list of courses assigned to room \(k\) */
2. \(\text{sorted}_\text{class} = \text{sort}(s_1, \ldots, s_n)\) /* sort by start time */
3. for \(c\) in \(\text{sorted}_\text{class}\) do
4. \(k = \text{find}_\text{compatible}_\text{room}(c, A, \Theta(n))\);
5. if \(k\) is not \(\)None\ then
6. \(\quad A[k].\text{add}(c);\)
7. else
8. \(\quad d \leftarrow \text{len}(A)\) /* highest room id */
9. \(\quad A[d + 1] = [\ ]\) /* open new room */
10. \(\quad A[d + 1].\text{add}(c);\)
11. return \(A\)

\(O(n \log n)\)

How can we implement the operations to have this runtime?
Priority Queue

Abstract data structure

Maintains set of items with priorities (= keys in queue)
  • contains <key, value> pairs
    - the meaning of the key and value are chosen specific to the application
  • we will use a Min-Queue; lower key = higher priority

Operations:
  • INSERT
  • PEEK-MIN: look at the key at the root
  • DECREASE-KEY: update the key and restore the min-queue
  • EXTRACT-MIN: find and remove item with min key

When to use: in applications where we keep track and/or update the minimum value
Implementation of the priority queue

Naive implementation of Q:
Use an unordered array.
  • we need to traverse the entire array for EXTRACT-MIN and DECREASE-KEY.
  • Each operation takes $O(n)$ time

Better:
Use a min-heap
  • (in this course we’re always using a binary tree-based heap)
  • Each operation takes $O(\log n)$ time
Min-Heap

Tree-based data structure that contains keys of items

- almost complete tree
- (here we only mention the keys stored in the heap, but there could be a pointer from each key to some attribute, e.g. key = distance, attribute = node id)

(min) Heap-property

- If node u is a parent of v, then \( key(u) \leq key(v) \)
- u may have multiple children, the heap-property doesn’t imply a specific order among them.
- It follows from the heap property that the minimum key is at the root
A complete binary tree in nature - Kenya

Hyphaene Compressa - Doum Palm

© Shlomit Pinter
Min-Heap vs. Binary Search Tree

Tree-based data structure that contains keys of item

(min) Heap-property

- If node \( u \) is a parent of \( v \), then \( key(u) \leq key(v) \)
- \( u \) may have multiple children, the heap-property doesn’t imply a specific order among them.

binary search tree

- if node \( u \) is the parent with left child \( v \) and right child \( w \), then
  \[ key(v) \leq key(u) \leq key(w) \]

![Diagram](image-url)
Priority queue — asymptotic running time of operations

In this course, when we refer to a Priority Queue, we always refer to a binary heap implementation. Given that the PQ holds n items the operations take:

- $O(1)$: peeking at the root.
- $O(\log n)$: INSERT, DECREASE-KEY, EXTRACT-MIN (a.k.a. DELETE-MIN)

The root of the tree has the information about the room that is available a.s.a.p.
Priority Queues for interval partitioning

How should we implement \texttt{find_compatible_room(c,A)} using PQs?

• we use PQs to efficiently keep track of the \underline{current minimum key}
• what are the keys in this PQ?
• what are the values corresponding to each key?

Use priority queue to keep track of the finish times in the rooms
Priority Queues for interval partitioning — TopHat

How should we implement `find_compatible_room(c, A)` using PQs?

- we use PQs to efficiently keep track of the current minimum key
- what are the keys in this PQ?
- what are the values corresponding to each key?

Question. PQs contain <key, value> pairs. What are the tuples to use for `find_compatible_room`?

A. `key = room id`, `value = last lecture id`
B. `key = room id`, `value = room finish time`
C. `key = room finish time`, `value = last lecture id`
D. `key = room finish`, `value = room id`

*best practice: key = quantity to minimize*
Algorithm 1: find_compatible_room(c, A, Q)

1 /* c: class id, A: current schedule of room assignments */
2 /* Q: priority queue with room finish times */
3 \( <f_k,k> = \text{PEEK\_MIN}(Q) /* \text{shows lowest} \ <\text{key}, \text{value}> \text{pair, } O(1) */ \)
4 if \( s_c > f_k \) then \( \text{finish time in room } k \)
5 \quad return \( k /* c \text{ is compatible with room } k */ \)
6 else
7 \quad return None;

We use the PQ to keep track of the finish time in each room

Trick:

• whatever value you want to keep track of the minimum should be set as the ‘key’
• in this case we use the ‘value’ to identify the object

example: key = finish time in room k, value = k (room ID)
**earliest-start-time-first algorithm - with priority queue**

**Algorithm 1: EarliestStartTimeFirst\((j = 1 \ldots n : s_j, f_j)\)**

```plaintext
/* s_j, f_j start and finish time of classes */
1 A ← empty hash table /* A[k] contains the list of courses assigned to room k */
2 Q ← empty priority queue /* contains <finishTime, roomId> */
3 sorted_class = sort\((s_1, \ldots, s_n)\)/* sort by start time */
4 for c in sorted_class do
5    k = find-compatible-room\((c, A, Q)\); \(\Theta(1)\)
6    if k is not None then
7        A[k].add\((c)\);
8        Q.DECREASE-KEY\((<f_k, k>, <f_c, k>)\)/* update finish time of room k */
9    else
10       d ← len\((A)\)/* highest room id */
11       A\([d + 1]\) = []/* open new room */
12       A\([d + 1]\).add\((c)\);
13       Q.INSERT\((<f_c, d + 1>)\); \(\Theta(\log n)\)
14 return A
```

**Running time?**

\(\Theta(n \log n) + \Theta(n) \cdot \Theta(\log n) = \Theta(n^2 \log n)\)
Interval partitioning: earliest-start-time-first algorithm

Proposition. The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

Pf. Store classrooms in a priority queue (key = finish time of its last lecture).

- To determine whether lecture $j$ is compatible with some classroom, compare $s_j$ to key of min classroom $k$ in priority queue.
- To add lecture $j$ to classroom $k$, increase key of classroom $k$ to $f_j$.
- Total number of priority queue operations is $O(n)$.
- updating key takes $O(\log n)$.
- Sorting by start time takes $O(n \log n)$ time.

Remark. This implementation chooses a classroom $k$ whose finish time of its last lecture is the earliest.
Interval partitioning — lower bound

This schedule is using three rooms. Can we use fewer?

The depth of an instance of interval partitioning is the max number of intervals overlapping at the same time.

What is the relationship between the depth and the required number of rooms?

We cannot use less than “depth” rooms.
Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest-start-time-first algorithm is optimal. (e.g. it uses the minimum possible number of rooms)

Trick that this proof depends on:
we want to find the min of something
  • e.g. number of classrooms

suppose that d is a lower bound to the min
  • e.g. depth is a lower bound on the number of rooms

Then for some value x, if we can show that x == d, then x must be minimum too.
Interval partitioning: correctness of earliest-start-time-first algorithm

Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest-start-time-first algorithm is optimal. (e.g. it uses the minimum possible number of rooms)

Pf. Suppose that the greedy approach uses $d$ rooms. (depth = $d$)

$\rightarrow$ Suppose that $j$ is the first class schedule in room $d$.

$\rightarrow$ By design of the algorithm, we only use a new room.

$\rightarrow$ Because all rooms are occupied at time $s_j$

$\rightarrow$ The other $d-1$ rooms are occupied with ongoing lectures.

$\rightarrow$ Therefore, the ESTF algorithm finds the optimal solution.

Because we don't need more than $d$ rooms.
Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest-start-time-first algorithm is optimal.

Pf.

• Let $d = \text{number of classrooms that the algorithm allocates.}$
• Classroom $d$ is opened because we needed to schedule a lecture, say $j$, that is incompatible with all $d - 1$ other classrooms.
• These $d$ lectures each end after $s_j$.
• Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than $s_j$.
• Thus, we have $d$ lectures overlapping at time $s_j + \varepsilon$.
• Key observation $\Rightarrow$ all schedules use $\geq d$ classrooms.