CS 603: Scheduling to Minimize Lateness

Ellen Veomett

University of San Francisco

Outline

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.
- We must prove that an algorithm does result in the desired solution. Typical techniques:

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.
- We must prove that an algorithm does result in the desired solution. Typical techniques:
    - Greedy algorithm stays ahead of any other optimal solution. (We already used this!)

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.
- We must prove that an algorithm does result in the desired solution. Typical techniques:
  - Greedy algorithm stays ahead of any other optimal solution. (We already used this!)
  - Any optimal solution can be *exchanged* step-by-step into a greedy solution. (This is what we will use today!)

Outline

1. Recap of Greedy Algorithms

2. Another Scheduling Problem: Minimizing Lateness

Scheduling Problem Description

- There is a single resource (say, a printer) that must be used to complete n tasks, each with different time lengths $t_1, t_2, \ldots, t_n$
- This resource is available to all tasks at a single start time, but may be used for only one task at a time.
- The resource must be used continuously on a task until that task is completed.
- Each task has a deadline $d_1, d_2, \ldots, d_n$.
- How do we schedule the task to minimize the maximum lateness?

Notation

t_i	the times it takes to complete item i
d_i	the deadlines for item i
$s(i)$	the assigned start time for item i
$f(i) = s(i) + t_i$	the corresponding finish time for item i
$\ell_i = f(i) - d_i$	the corresponding lateness for item i
$L = \max_{i=1,\ldots,n} \{\ell_i\}$	the maximum lateness

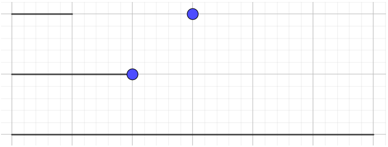
Since we want to minimize L, we can assume there is no idle time (that is, that the finish time for one item is the start time for the following item).

What could we choose as our "greedy" step?

One *incorrect* greedy option: Choose shortest jobs first: order by t_i.

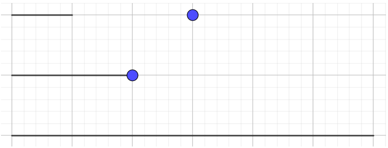
What could we choose as our "greedy" step?

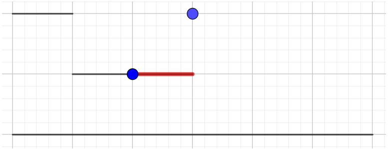
One *incorrect* greedy option: Choose shortest jobs first: order by t_i.
Problem: $t_1 = 1, d_1 = 3$ $t_2 = 2, d_2 = 2$



What could we choose as our "greedy" step?

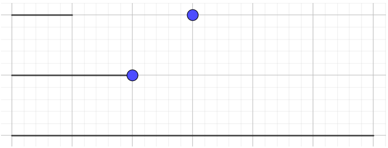
One *incorrect* greedy option: Choose shortest jobs first: order by t_i.
Problem: $t_1 = 1, d_1 = 3$ $t_2 = 2, d_2 = 2$

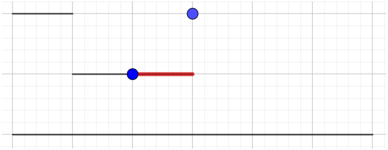




What could we choose as our “greedy” step?

One *incorrect* greedy option: Choose shortest jobs first: order by t_i.
Problem: $t_1 = 1, d_1 = 3$ $t_2 = 2, d_2 = 2$

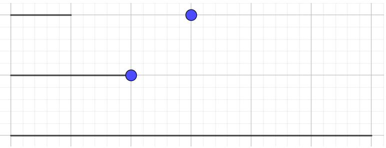


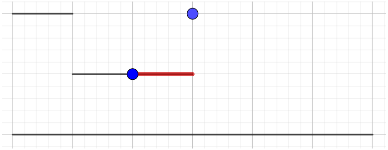


Another *incorrect* greedy option: Choose jobs with shortest slack time first: order by $d_i - t_i$

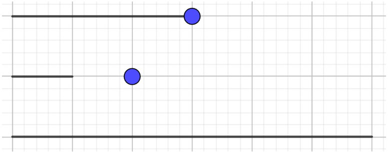
What could we choose as our “greedy” step?

One *incorrect* greedy option: Choose shortest jobs first: order by t_i.
Problem: $t_1 = 1, d_1 = 3$ $t_2 = 2, d_2 = 2$



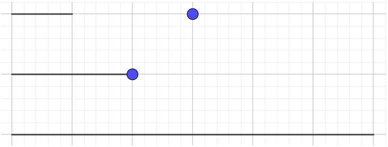


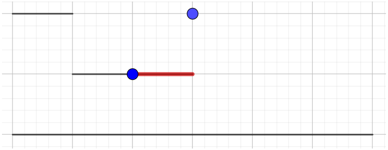
Another *incorrect* greedy option: Choose jobs with shortest slack time first: order by $d_i - t_i$
Problem: $t_1 = 3, d_1 = 3$ $t_2 = 1, d_2 = 2$



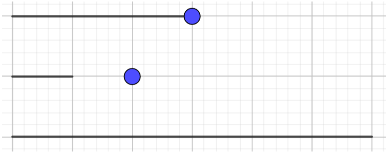
What could we choose as our “greedy” step?

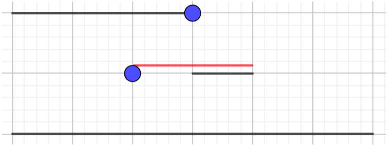
One *incorrect* greedy option: Choose shortest jobs first: order by t_i.
Problem: $t_1 = 1, d_1 = 3$ $t_2 = 2, d_2 = 2$





Another *incorrect* greedy option: Choose jobs with shortest slack time first: order by $d_i - t_i$
Problem: $t_1 = 3, d_1 = 3$ $t_2 = 1, d_2 = 2$





An aside question

Question: Why do I keep showing you greedy options that *don't* work?

An aside question

Question: Why do I keep showing you greedy options that *don't* work?

Answer:

- To show you that finding a greedy option that works may not be obvious or easy!

An aside question

Question: Why do I keep showing you greedy options that *don't* work?

Answer:

- To show you that finding a greedy option that works may not be obvious or easy!
- To keep you skeptical when approaching greedy algorithms, and remind you that you need to *prove* a greedy algorithm works.

Greedy *solution* to Scheduling to Minimize Lateness

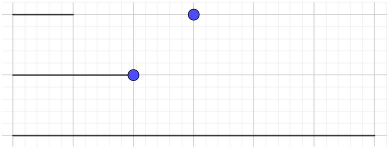
Pick the job with the earliest deadline first. That is, order by d_i.

Greedy *solution* to Scheduling to Minimize Lateness

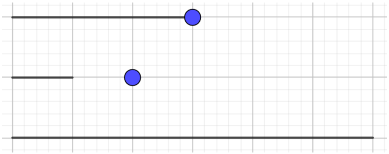
Pick the job with the earliest deadline first. That is, order by d_i.

Example

$t_1 = 1, d_1 = 3 \qquad t_2 = 2, d_2 = 2$



$t_1 = 3, d_1 = 3 \qquad t_2 = 1, d_2 = 2$



Proof that this greedy algorithm works

Re-number the indices of each interval so that they are ordered by deadline (ties broken arbitrarily):

$$d_1 \leq d_2 \leq \cdots d_n$$

First suppose that we have another different ordering that breaks the ties differently. That is, the ordering is still increasing by deadline, but elements with the same deadline may be swapped.

Proof that this greedy algorithm works

Re-number the indices of each interval so that they are ordered by deadline (ties broken arbitrarily):

$$d_1 \leq d_2 \leq \cdots d_n$$

First suppose that we have another different ordering that breaks the ties differently. That is, the ordering is still increasing by deadline, but elements with the same deadline may be swapped.

Note that, among all elements with the same deadline, the last one is the latest. Re-ordering those elements doesn't change the lateness. Thus, re-ordering elements with the same deadline among themselves produces an ordering with the same maximum lateness.

Suppose that we have an optimal solution which corresponds to the ordering

$$i_1, i_2, i_3, \ldots, i_n$$

Suppose that we have an optimal solution which corresponds to the ordering

$$i_1, i_2, i_3, \ldots, i_n$$

Suppose that the ordering in our optimal solution is not a greedy ordering.

Suppose that we have an optimal solution which corresponds to the ordering

$$i_1, i_2, i_3, \ldots, i_n$$

Suppose that the ordering in our optimal solution is not a greedy ordering.

We shall use the idea of an *inversion*. An inversion occurs when $j < k$ (so that i_j occurs before i_k in the ordering) but yet $d_{i_j} > d_{i_k}$. If this optimal solution is not a greedy ordering, we must have some positive number of inversions I.

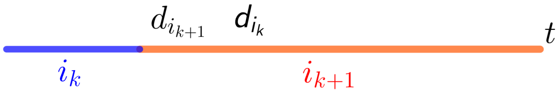
Suppose that we have an optimal solution which corresponds to the ordering

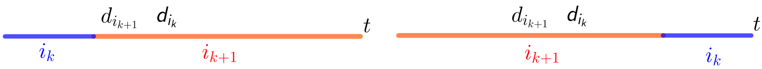
$$i_1, i_2, i_3, \ldots, i_n$$

Suppose that the ordering in our optimal solution is not a greedy ordering.

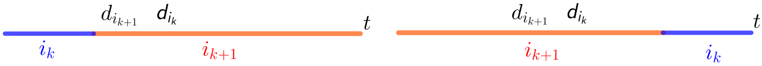
We shall use the idea of an *inversion*. An inversion occurs when $j < k$ (so that i_j occurs before i_k in the ordering) but yet $d_{i_j} > d_{i_k}$. If this optimal solution is not a greedy ordering, we must have some positive number of inversions I.

We must also have an *adjacent inversion*: a case where $d_{i_k} > d_{i_{k+1}}$. Suppose we swap those intervals.



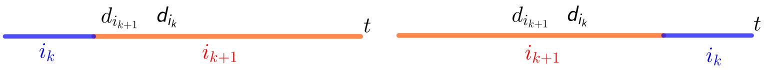


After swapping, item i_{k+1} now ends earlier, so its lateness cannot increase.



After swapping, item i_{k+1} now ends earlier, so its lateness cannot increase. Item i_k now ends at item i_{k+1}'s old finishing time. Let's call that time t. Thus, item i_k's new lateness is

$$t - d_{i_k}$$



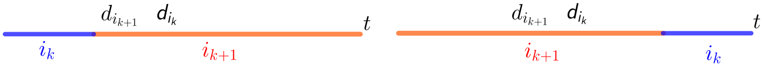
After swapping, item i_{k+1} now ends earlier, so its lateness cannot increase. Item i_k now ends at item i_{k+1}'s old finishing time. Let's call that time t. Thus, item i_k's new lateness is

$$t - d_{i_k}$$

But sinde $d_{i_k} > d_{i_{k+1}}$, we have

$$t - d_{i_k} < t - d_{i_{k+1}}$$

Thus, after swapping, item i_k's new lateness time is no more than item i_{k+1}'s old lateness time. Thus, the maximum lateness cannot have gone up!



After swapping, item i_{k+1} now ends earlier, so its lateness cannot increase. Item i_k now ends at item i_{k+1}'s old finishing time. Let's call that time t. Thus, item i_k's new lateness is

$$t - d_{i_k}$$

But sinde $d_{i_k} > d_{i_{k+1}}$, we have

$$t - d_{i_k} < t - d_{i_{k+1}}$$

Thus, after swapping, item i_k's new lateness time is no more than item i_{k+1}'s old lateness time. Thus, the maximum lateness cannot have gone up!

Hence, we can swap any adjacent inversions in an optimal solution without increasing maximum lateness, decreasing the total number of inversions I by 1. Thus, any optimal solution can be eventually exchanged (by swapping adjacent inversions) into a greedy solution without increasing maximum lateness.

One more recap

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.

One more recap

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.

One more recap

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.
- We must prove that an algorithm does result in the desired solution. Typical techniques:

One more recap

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.
- We must prove that an algorithm does result in the desired solution. Typical techniques:
    - Greedy algorithm stays ahead of any other optimal solution. (This is what we used yesterday!)

One more recap

- Greedy algorithms use a rule to choose a small locally optimal step *that results in a globally optimal solution*.
- Choosing that locally optimal step can be tricky; many ideas will not lead to a globally optimal solution.
- We must prove that an algorithm does result in the desired solution. Typical techniques:
    - Greedy algorithm stays ahead of any other optimal solution. (This is what we used yesterday!)
    - Any optimal solution can be *exchanged* step-by-step into a greedy solution. (This is what we used today!)