Lecture 5: Greedy I

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

September 21, 2023

Overview

- Greedy Algorithms
  - Greedy approach
  - Interval Scheduling
  - Interval Coloring
  - Minimizing Completion Time

- Acknowledgements

Going greedy

Greedy Approach

- Greedy strategy based on following principles:
  1. choose a "progress measure"
  2. preprocess input accordingly
  3. make next decision based on what is *best* given *current* partial solution
  4. **Main idea:** must show that the greedy solution is always *no worse* than any other optimal solution!

     Usually can prove this by begin able to "transform" any optimal solution into the greedy one without losing anything.
  5. *Optimal Substructure*: a problem has optimal substructure if any optimal solution contains optimal solutions to subproblems.

- Greedy Algorithms
  - Greedy approach
  - Interval Scheduling
  - Interval Coloring
  - Minimizing Completion Time
- Acknowledgements

Interval Scheduling

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *maximum* set of disjoint intervals
- **Model:** word RAM model

Interval Scheduling

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *maximum* set of disjoint intervals

- How to go greedy?
  1. pick interval with *earliest* starting time $(\min_i s_i)$
  2. pick interval with *earliest* finishing time $(\min_i f_i)$
  3. pick *shortest* interval $(\min_i f_i - s_i)$
  4. pick interval with *minimum* number of conflicts

Interval Scheduling

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *maximum* set of disjoint intervals

- How to go greedy?
  1. pick interval with *earliest* starting time $(\min_i s_i)$
  2. pick interval with *earliest* finishing time $(\min_i f_i)$
  3. pick *shortest* interval $(\min_i f_i - s_i)$
  4. pick interval with *minimum* number of conflicts
- Approach 1 not good: earliest starting time can be very long

Interval Scheduling

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *maximum* set of disjoint intervals

- How to go greedy?
  1. pick interval with *earliest* starting time $(\min_i s_i)$
  2. pick interval with *earliest* finishing time $(\min_i f_i)$
  3. pick *shortest* interval $(\min_i f_i - s_i)$
  4. pick interval with *minimum* number of conflicts
- Approach 3 not good: could have few short intervals

Interval Scheduling

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *maximum* set of disjoint intervals

- How to go greedy?
  1. pick interval with *earliest* starting time $(\min_i s_i)$
  2. pick interval with *earliest* finishing time $(\min_i f_i)$
  3. pick *shortest* interval $(\min_i f_i - s_i)$
  4. pick interval with *minimum* number of conflicts
- Approach 4 not good: picking minimum number of conflicts can block many good intervals

Interval Scheduling

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *maximum* set of disjoint intervals

- How to go greedy?
  1. pick interval with *earliest* starting time $(\min_i s_i)$
  2. pick interval with *earliest* finishing time $(\min_i f_i)$
  3. pick *shortest* interval $(\min_i f_i - s_i)$
  4. pick interval with *minimum* number of conflicts
- What about strategy 2?

Seems like this is good. How can we show this works?

Earliest Finishing Time

- Algorithm:
  1. Sort intervals by finishing time, so we can assume $f_1 \leq f_2 \cdots \leq f_n$
  2. Initial solution $S = \emptyset$, $k = 0$ and we set $f_0 = -\infty$
  3. For $i \in [n]$:
     - If $s_i \geq f_k$, then set $k \leftarrow i$ and add $[s_i, f_i]$ to S
  4. Return S

Earliest Finishing Time

- Algorithm:
  1. Sort intervals by finishing time, so we can assume $f_1 \leq f_2 \cdots \leq f_n$
  2. Initial solution $S = \emptyset$, $k = 0$ and we set $f_0 = -\infty$
  3. For $i \in [n]$:
     - If $s_i \geq f_k$, then set $k \leftarrow i$ and add $[s_i, f_i]$ to S
  4. Return S
- **Correctness:**

  Idea: show that any (optimal) solution would do no worse by picking interval with earliest finishing time.

  Then induct!

Earliest Finishing Time

- Algorithm:
  1. Sort intervals by finishing time, so we can assume $f_1 \leq f_2 \cdots \leq f_n$
  2. Initial solution $S = \emptyset$, $k = 0$ and we set $f_0 = -\infty$
  3. For $i \in [n]$:
     - If $s_i \geq f_k$, then set $k \leftarrow i$ and add $[s_i, f_i]$ to S
  4. Return S
- **Correctness:**
  - **Claim 1:** there is optimal solution with $[s_1, f_1]$
    - Let $[s_{j_1}, f_{j_1}], \ldots, [s_{j_\ell}, f_{j_\ell}]$ be optimal solution, with $f_{j_1} \leq f_{j_2} \leq \cdots \leq f_{j_\ell}$
    - since $f_1 \leq f_{j_1} < s_{j_2}$, we have that $[s_1, f_1], \ldots, [s_{j_\ell}, f_{j_\ell}]$ also optimal

Earliest Finishing Time

- Algorithm:
  1. Sort intervals by finishing time, so we can assume $f_1 \leq f_2 \cdots \leq f_n$
  2. Initial solution $S = \emptyset$, $k = 0$ and we set $f_0 = -\infty$
  3. For $i \in [n]$:
     - If $s_i \geq f_k$, then set $k \leftarrow i$ and add $[s_i, f_i]$ to S
  4. Return S
- **Correctness:**
  - **Claim 1:** there is optimal solution with $[s_1, f_1]$
    - Let $[s_{j_1}, f_{j_1}], \ldots, [s_{j_\ell}, f_{j_\ell}]$ be optimal solution, with $f_{j_1} \leq f_{j_2} \leq \cdots \leq f_{j_\ell}$
    - since $f_1 \leq f_{j_1} < s_{j_2}$, we have that $[s_1, f_1], \ldots, [s_{j_\ell}, f_{j_\ell}]$ also optimal
  - **Claim 2:** optimal solution for input $\{[s_i, f_i] \ : \ s_i > f_1\}$ together with $[s_1, f_1]$ is optimal solution to our problem
    - Same proof as the one above

Note that greedy always "stays ahead"

Earliest Finishing Time

- Algorithm:
  1. Sort intervals by finishing time, so we can assume $f_1 \leq f_2 \cdots \leq f_n$
  2. Initial solution $S = \emptyset$, $k = 0$ and we set $f_0 = -\infty$
  3. For $i \in [n]$:
     - If $s_i \geq f_k$, then set $k \leftarrow i$ and add $[s_i, f_i]$ to S
  4. Return S
- **Correctness:**
  - **Claim 1:** there is optimal solution with $[s_1, f_1]$
    - Let $[s_{j_1}, f_{j_1}], \ldots, [s_{j_\ell}, f_{j_\ell}]$ be optimal solution, with $f_{j_1} \leq f_{j_2} \leq \cdots \leq f_{j_\ell}$
    - since $f_1 \leq f_{j_1} < s_{j_2}$, we have that $[s_1, f_1], \ldots, [s_{j_\ell}, f_{j_\ell}]$ also optimal
  - **Claim 2:** optimal solution for input $\{[s_i, f_i] \ : \ s_i > f_1\}$ together with $[s_1, f_1]$ is optimal solution to our problem
    - Same proof as the one above
  - **Induction:** if our greedy is optimal for sets of size $\leq n - 1$, then it is optimal for any input of size n (proved in claim 2)

Earliest Finishing Time

- Algorithm:
  1. Sort intervals by finishing time, so we can assume $f_1 \leq f_2 \cdots \leq f_n$
  2. Initial solution $S = \emptyset$, $k = 0$ and we set $f_0 = -\infty$
  3. For $i \in [n]$:
     - If $s_i \geq f_k$, then set $k \leftarrow i$ and add $[s_i, f_i]$ to S
  4. Return S
- **Correctness:**
  - **Claim 1:** there is optimal solution with $[s_1, f_1]$
    - Let $[s_{j_1}, f_{j_1}], \ldots, [s_{j_\ell}, f_{j_\ell}]$ be optimal solution, with $f_{j_1} \leq f_{j_2} \leq \cdots \leq f_{j_\ell}$
    - since $f_1 \leq f_{j_1} < s_{j_2}$, we have that $[s_1, f_1], \ldots, [s_{j_\ell}, f_{j_\ell}]$ also optimal
  - **Claim 2:** optimal solution for input $\{[s_i, f_i] \ : \ s_i > f_1\}$ together with $[s_1, f_1]$ is optimal solution to our problem
    - Same proof as the one above
  - **Induction:** if our greedy is optimal for sets of size $\leq n - 1$, then it is optimal for any input of size n (proved in claim 2)
- **Running time:** sorting then linear scan $\Rightarrow O(n \log n)$

- Greedy Algorithms
  - Greedy approach
  - Interval Scheduling
  - Interval Coloring
  - Minimizing Completion Time

- Acknowledgements

Interval colouring

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *minimum* number of colours such that each interval gets one colour and we always colour *overlapping intervals* with *distinct* colours
- **Model:** word RAM model

Interval colouring

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *minimum* number of colours such that each interval gets one colour and we always colour *overlapping intervals* with *distinct* colours

- one approach:
  1. use previous problem to find maximum set of non-overlapping intervals
  2. assign a colour to this set
  3. recurse on the remaining intervals

Interval colouring

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *minimum* number of colours such that each interval gets one colour and we always colour *overlapping intervals* with *distinct* colours

- one approach:
  1. use previous problem to find maximum set of non-overlapping intervals
  2. assign a colour to this set
  3. recurse on the remaining intervals
- **Exercise:** show this won't work...

Interval colouring

- **Input:** n intervals (with integral endpoints) $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$, where $s_i < f_i$
- **Output:** a *minimum* number of colours such that each interval gets one colour and we always colour *overlapping intervals* with *distinct* colours

- one approach:
  1. use previous problem to find maximum set of non-overlapping intervals
  2. assign a colour to this set
  3. recurse on the remaining intervals
- **Exercise:** show this won't work...
- **Observation:** if there is a time t where k intervals overlap, then the minimum number of colours is $\geq k$

Is this only obstacle?

Interval Colouring

- We will associate to each colour a natural number
- Algorithm:
  1. Sort intervals by start time, so that $s_1 \leq s_2 \leq \cdots \leq s_n$
  2. Let A be a set of active intervals (i.e., whose finishing time "has not passed" yet). Initialize $A \leftarrow \{\}$.
  3. For $i \in [n]$
     - Update A by removing any interval $[s_j, f_j]$ with $f_j < s_i$
     - use minimum available colour to colour interval i

       I.e., use minimum colour that was not assigned to an active interval.
  4. output colouring and number of colours used

Interval Colouring

- We will associate to each colour a natural number
- Algorithm:
  1. Sort intervals by start time, so that $s_1 \leq s_2 \leq \cdots \leq s_n$
  2. Let A be a set of active intervals (i.e., whose finishing time "has not passed" yet). Initialize $A \leftarrow \{\}$.
  3. For $i \in [n]$
     - Update A by removing any interval $[s_j, f_j]$ with $f_j < s_i$
     - use minimum available colour to colour interval i

       I.e., use minimum colour that was not assigned to an active interval.
  4. output colouring and number of colours used
- **Correctness:** must show that cannot use $k-1$ colours. This follows from observation in previous slide, as greedy uses k colours $\Rightarrow$ there is $i \in [n]$ such that $length(A)$ after cleaning up (at the i^{th} step) is $k-1$, thus we must have k overlapping intervals.

Interval Colouring

- We will associate to each colour a natural number
- Algorithm:
  1. Sort intervals by start time, so that $s_1 \leq s_2 \leq \cdots \leq s_n$
  2. Let A be a set of active intervals (i.e., whose finishing time "has not passed" yet). Initialize $A \leftarrow \{\}$.
  3. For $i \in [n]$
     - Update A by removing any interval $[s_j, f_j]$ with $f_j < s_i$
     - use minimum available colour to colour interval i

       I.e., use minimum colour that was not assigned to an active interval.
  4. output colouring and number of colours used
- **Correctness:** must show that cannot use $k - 1$ colours. This follows from observation in previous slide, as greedy uses k colours $\Rightarrow$ there is $i \in [n]$ such that $length(A)$ after cleaning up (at the i^{th} step) is $k - 1$, thus we must have k overlapping intervals.
- **Running time:** if we output k colours, then length of A is upper bounded by $k - 1$, so running time $O(n \cdot k) = O(n^2)$

- Greedy Algorithms
  - Greedy approach
  - Interval Scheduling
  - Interval Coloring
  - Minimizing Completion Time

- Acknowledgements

Minimizing Completion Time

- **Input:** n tasks, with processing times $p_1, \ldots, p_n \in [n^{100}]$
- **Output:** an ordering of the tasks that minimizes total completion time
- **Model:** word RAM
- **Example:** given tasks with processing times $2, 3, 5, 11$, if we schedule them in this order we get completion times: $2, 5, 10, 21$, so total completion time is 38

Minimizing Completion Time

- **Input:** n tasks, with processing times $p_1, \ldots, p_n \in [n^{100}]$
- **Output:** an ordering of the tasks that minimizes total completion time
- **Intuition:** makes sense to schedule "faster/easier" tasks earlier

Minimizing Completion Time

- **Input:** n tasks, with processing times $p_1, \ldots, p_n \in [n^{100}]$
- **Output:** an ordering of the tasks that minimizes total completion time
- **Intuition:** makes sense to schedule "faster/easier" tasks earlier
- Turns out this greedy approach works!

Minimizing Completion Time

- **Input:** n tasks, with processing times $p_1, \ldots, p_n \in [n^{100}]$
- **Output:** an ordering of the tasks that minimizes total completion time
- **Intuition:** makes sense to schedule "faster/easier" tasks earlier
- Turns out this greedy approach works!
- Algorithm:
  1. Sort tasks by processing times, so can assume $p_1 \leq \cdots \leq p_n$
  2. Output the set $[n]$ (after the relabeling)

Minimizing Completion Time

- **Input:** n tasks, with processing times $p_1, \ldots, p_n \in [n^{100}]$
- **Output:** an ordering of the tasks that minimizes total completion time
- **Intuition:** makes sense to schedule "faster/easier" tasks earlier
- Turns out this greedy approach works!
- Algorithm:
  1. Sort tasks by processing times, so can assume $p_1 \leq \cdots \leq p_n$
  2. Output the set $[n]$ (after the relabeling)
- **Correctness:** if we output any other order $p_{i_1}, \ldots, p_{i_n}$, there is index $t \in [n-1]$ such that $i_t > i_{t+1}$ and thus $p_{i_t} \geq p_{i_{t+1}}$, so swapping these two tasks changes the total completion time by $p_{i_{t+1}} - p_{i_t} \leq 0$, so we are improving.

Minimizing Completion Time

- **Input:** n tasks, with processing times $p_1, \ldots, p_n \in [n^{100}]$
- **Output:** an ordering of the tasks that minimizes total completion time
- **Intuition:** makes sense to schedule "faster/easier" tasks earlier
- Turns out this greedy approach works!
- Algorithm:
  1. Sort tasks by processing times, so can assume $p_1 \leq \cdots \leq p_n$
  2. Output the set $[n]$ (after the relabeling)
- **Correctness:** if we output any other order $p_{i_1}, \ldots, p_{i_n}$, there is index $t \in [n-1]$ such that $i_t > i_{t+1}$ and thus $p_{i_t} \geq p_{i_{t+1}}$, so swapping these two tasks changes the total completion time by $p_{i_{t+1}} - p_{i_t} \leq 0$, so we are improving.
- **Running time:** only sorted and output the reindexed set $\Rightarrow O(n \log n)$

Fancier Completion Time

- **Input:** n tasks, with processing times and release times $(p_1, r_1), \ldots, (p_n, r_n) \in [n^{100}]^2$
- **Output:** an ordering of the tasks that minimizes total completion time.
- **Constraints & capabilities:** Now, task i can only be scheduled from time r_i onwards, and we also allow *preemption*, that is, we can suspend a task and resume it at a later given time.
- **Model:** word RAM

Acknowledgement

- Based on Prof Lau's Lecture 8

  `https://cs.uwaterloo.ca/~lapchi/cs341/notes/L08.pdf`
- Also on [CLRS 2009, Chapter 16]

References I

Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)

Introduction to Algorithms, third edition.

MIT Press