SOLVING OPTIMIZATION PROBLEMS

Lots of techniques
- We will study greedy approaches first
- Later, dynamic programming
  - Sort of like divide and conquer
  - but can sometimes be much more efficient than D&C
- Greedy algorithms are usually
  - Very fast, but hard to prove optimality for
  - Structured as follows...

**The Greedy Method**

**Partial solutions**
Given a problem instance \( I \), it should be possible to write a feasible solution \( X \) as a tuple \([x_1, x_2, \ldots, x_i]\) for some integer \( n \), where \( x_i \in X \) for all \( i \). A tuple \([x_1, x_2, \ldots, x_i]\) where \( i < n \) is a **partial solution** if no constraints are violated.

**Choice set**
For a partial solution \( X = [x_1, \ldots, x_i] \) where \( i < n \), we define the choice set
\[
\text{choice}(X) = \{ y \in X : [x_1, \ldots, x_i, y] \text{ is a partial solution} \}.
\]

**Local evaluation criterion**
For any \( y \in X \), \( f(y) \) is a local evaluation criterion that measures the cost or profit of including \( y \) in a (partial) solution.

**Extension**
Given a partial solution \( X = [x_1, \ldots, x_i] \) where \( i < n \), choose \( y \in \text{choice}(X) \) so that \( f(y) \) is as small (or large) as possible. Update \( X \) to be the \((i+1)\)-tuple \([x_1, \ldots, x_i, y]\).

**Greedy algorithm**
Starting with the “empty” partial solution, repeatedly extend it until a feasible solution \( X \) is constructed. This feasible solution may or may not be optimal.
Core Characteristics of Greedy Algorithms

- Greedy algorithms do not look ahead and no backtracking.
- Greedy algorithms can usually be implemented efficiently. Often they consist of a preprocessing step based on the function \( g \), followed by a single pass through the data.
- In a greedy algorithm, only one feasible solution is constructed.
- The execution of a greedy algorithm is based on local criteria (i.e., the values of the function \( g \)).

Correctness: For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!

**Problem:**

**Interval Selection**

- Input: a set \( A = \{A_1, \ldots, A_n\} \) of time intervals
- Each interval \( A_i \) has a start time \( s_i \) and a finish time \( f_i \)
- Feasible solution: a subset \( X \) of \( A \) containing pairwise disjoint intervals
- Output: a feasible solution of maximum size
  - i.e., one that maximizes \( |X| \)

Possible Greedy Strategies

- Partial solutions
  - \( X = A \) (i.e., all intervals)
  - Choice: \( X = \{y \in X : \{x_1, \ldots, x_i, y\} \text{ respects all constraints} \} \)
  - i.e., \( y \in X \) and \( \forall x \in X \) disjoint \( (y, x) \)

Local evaluation function
  - \( g(y) = s_j \) where \( y = A[j] \)
  - (i.e., \( g(y) = \text{start time of interval } y \))

Possible Greedy Strategies for Interval Selection

1. Sort the intervals in increasing order of starting times. At any stage, choose the earliest starting interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is \( s_i \)).
2. Sort the intervals in increasing order of duration. At any stage, choose the interval of minimum duration that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is \( f_i - s_i \)).
3. Sort the intervals in increasing order of finishing times. At any stage, choose the earliest finishing interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is \( f_i \)).

Does one of these strategies yield a correct greedy algorithm?

Strategy 1: Proving Incorrectness

- Idea: find one input for which the algorithm gives a non-optimal solution or an infeasible solution

Consider input:

\( (0, 10), (1, 3), (5, 7) \)
HOW ABOUT STRATEGY 2?

Consider input: \( (6, 5), [6, 10], [4, 7] \).

We will show that Strategy 3 (sort in increasing order of finishing times) always yields the optimal solution.

**Strategy 2**

Sort the intervals in increasing order of duration. At any stage, choose the interval of minimum duration that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is \( f_i - x_i \)).

**Strategy 3**

Time complexity: Sort + one pass \( \in \Theta(n \log n) \)

How to prove this is correct? (I.e., how can we show the returned solution is both feasible and optimal?)

Feasibility? Easy! We always choose an interval that starts after all other chosen intervals end.

Optimality? Harder...

**GREEDY CORRECTNESS PROOFS**

Want to prove: greedy solution \( X \) is correct (feasible & optimal)

Usually show feasibility directly and optimality by contradiction:

- Suppose solution \( O \) is better than \( X \)
- Show this necessarily leads to a contradiction

Two broad strategies for deriving this contradiction:

1. **Greedy stays ahead**: show every choice in \( X \) is "at least as good" as the corresponding choice in \( O \)
2. **Exchange**: show \( O \) can be improved by replacing some choice in \( O \) with a choice in \( X \)

Let's demonstrate approach #1

We give an induction proof.

Let \( X \) be the greedy solution,

\[ X = (A_{i_1}, \ldots, A_{i_k}) \]

where \( i_1 < \cdots < i_k \).

Let \( O \) be any optimal solution,

\[ O = (A_{j_1}, \ldots, A_{j_k}) \]

where \( j_1 < \cdots < j_k \).

CRUCIAL: We are NOT assuming the optimal algorithm uses the same sort order!

We are merely imagining reordering the intervals chosen by the optimal algorithm so we can easily compare their finish times to intervals in \( X \)
Correctness Proof (cont.)

Recall: Greedy solution is \( X = (A_1, ..., A_k) \).
Optimal solution is \( O = (A_{j_1}, ..., A_{j_{\ell}}) \).

Now we complete the proof.

From the Lemma, we have \( f_k \leq f_k \).
Suppose that \( k > k \).

No obtain a contradiction.

Greedy solution is \( X = (A_1, ..., A_k) \).
Optimal solution is \( O = (A_{j_1}, ..., A_{j_{\ell}}) \).

Recall:
1. \( A_{j_k} + 1 \) starts after \( A_{j_k} \) finishes (by disjointness)
2. \( A_i \) finishes before \( A_{j_k} \) (by lemma)
3. \( A_{j_k} \) finishes after \( A_i \) finishes

This completes the proof.

A DIFFERENT PROOF

"Slick" ad-hoc approaches are sometimes possible...

Let \( F = \{ f_1, ..., f_k \} \) be the finishing times of the intervals in \( X \).

No interval finish strictly to the left
No interval starts strictly to the right
No interval in \( F \) is strictly between these points!

So, in addition to the intervals in \( X \), only the following types of intervals are possible:

Contains \( f_i \)
Contains \( f_j \)
Contains \( f_i \) and \( f_j \)

Thus, every interval contains some finishing time in \( F \).
And, two intervals in \( O \) cannot contain the same element of \( F \).

So, there must be as many finishing times in \( F \) as there are intervals in \( O \). QED

PROBLEM: INTERVAL COLOURING

Instance: A set \( A = \{ A_1, ..., A_n \} \) of intervals.
For \( 1 \leq i \leq n \), \( A_i = (a_i, b_i) \), where \( a_i \) is the start time of interval \( A_i \) and \( b_i \) is the finish time of \( A_i \).

Feasible solution: A \( c \)-colouring is a mapping \( c : A \rightarrow \{ 1, ..., c \} \) that assigns each interval a colour such that no two intervals receiving the same colour are disjoint.

Find: A \( c \)-colouring of \( A \) with the minimum number of colours.

Example: 7 intervals, \( 7 \) colours. Feasible, but not optimal.
MORE EXAMPLES

Example

Not feasible!

Example

Feasible, but not optimal

Example

Optimal

Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time.

At a given point in time, suppose we have coloured the first \( i < n \) intervals

using \( d \) colours.

We will colour the \((i + 1)\)th interval with any permissible colour. If it

cannot be coloured using any of the existing \( d \) colours, then we introduce

a new colour and \( d \) is increased by 1.

Question: In what order should we consider the intervals?

We will colour the \((i + 1)\)th interval with any permissible colour. If it

cannot be coloured using any of the existing \( d \) colours, then we introduce

a new colour and \( d \) is increased by 1.
EXAMPLE: ORDER MATTERS!
EXAMPLE: ORDER MATTERS!

Used 4 colours

Can we do better?

EXAMPLE: ORDER MATTERS!

Pre-sort intervals by increasing start time!
EXAMPLE: ORDER MATTERS!
Example:
Order Matters!

Used 3 colours
Can we do better?

Example:
Running Greedy

Initial state

Example:
Running Greedy

Code before the loop: just assign colour 1

\(d = 1\)
\(\text{finish}[1] = \) finish time of last interval to receive colour 1

\(i = 1\)
\(d = 2\)
\(\text{finish}[1] = \) finish time of last interval to receive colour 1

Check if we can reuse any colour in 1..d

Is \(\text{finish}[1] \leq s_2\) ?
No. We cannot reuse colour 1.

While loop over c. Check if we can reuse any colour. Create a new one!
EXAMPLE: RUNNING GREEDY

While loop over c:
Check if we can reuse a color in 1..d.


Is \( f_i \leq s \) ?
No. We cannot reuse colour 1.

i = 3

While loop over c.
Check if we can reuse a colour in 1..d.

d = 2

Cannot reuse any colour. Create new one.

Is \( f_i \leq s \) ?
Yes. We can reuse colour 1.

i = 4

While loop over c.
Check if we can reuse a colour in 1..d.


Is \( f_i \leq s \) ?
No. We cannot reuse colour 1.

Is \( f_i \leq s \) ?
No. We cannot reuse colour 2.

Cannot reuse any colour. Create new one.

Is \( f_i \leq s \) ?
Yes. We can reuse colour 3.

i = 5

While loop over c.
Check if we can reuse a colour in 1..d.


Is \( f_i \leq s \) ?
No. We cannot reuse colour 1.

Is \( f_i \leq s \) ?
Yes. We can reuse colour 1.

i = 6

While loop over c.
Check if we can reuse a colour in 1..d.


Is \( f_i \leq s \) ?
No. We cannot reuse colour 2.

Is \( f_i \leq s \) ?
No. We cannot reuse colour 3.

No, we cannot reuse any colour. Create new one.

\( x \)-axis

0 2 4 6 8 10 12 14 16 18 20
Correctness of the Algorithm

The correctness of this greedy algorithm can be proven inductively as well as by a "slick" method—we give the "slick" proof.

Let $D$ denote the number of colours used by the algorithm.

Let $F_0$ be the first interval that has the last colour.

Let $I_x$ be the last interval that has colour $x$ and starts before $F_0$ ends.

We prove $F_0$ overlaps every interval $I_x$ for all $x < D$.

If $D = 1$ the proof is immediate, so suppose $D \geq 2$.

TIME COMPLEXITY?

Total $O(n \log n + nd)$

Could be $O(n \log n)$ if only a constant number of colours are needed (or even less colours!)

Could be $n^2$ if $n$ colours are needed!

Most accurate complexity statement is $O(n \log n + nd)$ where $d$ is if colour used.

What inefficiencies exist in this algorithm?
Could we make it faster with clever data structure usage?

IMPROVING THIS ALGORITHM

Current greedy algorithm:
- For each interval $A_i$, compare its start time $s_i$ with the $\text{finish}[i]$ times of all colours introduced so far.
- Why? Looking for some $\text{finish}[i]$ that is earlier than $s_i$.
- We are doing linear search... Can we do better?
- Use a priority queue to keep track of the earliest $\text{finish}[i]$ at all times in the algorithm.
- Then we only need to look at minimum element...
EXAMPLE: HEAP-BASED ALGORITHM

Initial state

Min element NULL
Heap

EXAMPLE: HEAP-BASED ALGORITHM

Iteration i = 1

Finish at time 3

No. New colour!

Finish at time 7
EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 3
Heap: Min element: finish at time 3

Check heap minimum: check if finish time 3 is before \( s_3 \)

Iteration \( i = 3 \)
Finish at time 3
No. New colour!

Finish at time 7
Finish at time 5

EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 3
Heap: Min element: finish at time 3

Check heap minimum: check if finish time 3 is before \( s_4 \)

Iteration \( i = 4 \)
Finish at time 3
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7
Finish at time 9

EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5
Heap: Min element: finish at time 5

Check heap minimum: check if finish time 5 is before \( s_5 \)

Iteration \( i = 5 \)
Finish at time 7
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7
Finish at time 9

EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5
Heap: Min element: finish at time 5

Check heap minimum: check if finish time 5 is before \( s_6 \)

Iteration \( i = 6 \)
Finish at time 7
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7
Finish at time 9

EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5
Heap: Min element: finish at time 5

Check heap minimum: check if finish time 5 is before \( s_7 \)

Iteration \( i = 7 \)
Finish at time 7
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7
Finish at time 9

EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5
Heap: Min element: finish at time 5

Check heap minimum: check if finish time 5 is before \( s_8 \)

Iteration \( i = 8 \)
Finish at time 7
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7
Finish at time 9

EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5
Heap: Min element: finish at time 5

Check heap minimum: check if finish time 5 is before \( s_9 \)

Iteration \( i = 9 \)
Finish at time 7
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7
Finish at time 9

EXAMPLE: HEAP-BASED ALGORITHM

Min element: finish at time 5
Heap: Min element: finish at time 5

Check heap minimum: check if finish time 5 is before \( s_{10} \)

Iteration \( i = 10 \)
Finish at time 7
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7
Finish at time 9
EXAMPLE: HEAP-BASED ALGORITHM

Min element

Heap

x-axis

0 2 4 6 8 10 12 14 16 18 20

Iteration i

Check heap minimum

Check if finish time

Yes, reuse colour, deleteMin and insert new finish time into heap!

Finish at

Time 7

Time 9

Time 11

And so on, and so forth...