Optimization Problems

Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.
Problem Instance: Input for the specified problem.
Problem Constraints: Requirements that must be satisfied by any feasible solution.
Feasible Solution: For any problem instance \( I \), \( \text{feasible}(I) \) is the set of all outputs (i.e., solutions) for the instance \( I \) that satisfy the given constraints.
Objective Function: A function \( f: \text{feasible}(I) \to \mathbb{R}^+ \cup \{0\} \). We often think of \( f \) as being a profit or a cost function.
Optimal Solution: A feasible solution \( X \in \text{feasible}(I) \) such that the profit \( f(X) \) is maximized (or the cost \( f(X) \) is minimized).

The Greedy Method

Given a problem instance \( I \), it should be possible to write a feasible solution \( X \) as a tuple \( [x_1, x_2, \ldots, x_n] \) for some integer \( n \), where \( x_i \in X \) for all \( i \). A tuple \( [x_1, \ldots, x_i] \) where \( i < n \) is a partial solution if no constraints are violated. Note: it may be the case that a partial solution cannot be extended to a feasible solution.

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

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 (cont.)

For any \( y \in X \), \( g(y) \) 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 \( g(y) \) 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 no looking 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

1. Partial solutions
2. Choices
3. Local evaluation function

POSSIBLE GREEDY STRATEGIES FOR INTERVAL SELECTION

1. Sort the intervals in increasing order of starting time. 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 time. 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

Strategy 1

Consider input $\{(0,10),(1,3),(5,7)\}$
HOW ABOUT STRATEGY 2?

Consider input: $[0, 5), [6, 10], [4, 7]$.

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 $f_i - s_i$).

**Strategy 3**

Time complexity: $\Theta(n \log n)$

We give an induction proof.

Let $X$ be the greedy solution, $X = (A_1, \ldots, A_k)$, where $i_1 < \cdots < i_k$.

Let $\mathcal{O}$ be any optimal solution, $\mathcal{O} = (A_{j_1}, \ldots, A_{j_l})$, where $j_1 < \cdots < j_l$.

**GREEDY CORRECTNESS PROOFS**

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

Usually show feasibility directly and optimality by contradiction:

- Suppose solution $\mathcal{O}$ 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 $\mathcal{O}$
2. **Exchange**: show $\mathcal{O}$ can be improved by replacing some choice in $\mathcal{O}$ with a choice in $X$

Let’s demonstrate approach #1

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...

Where is our local evaluation function $g$ in this code?

We are merely imagining reordering the intervals chosen by the optimal algorithm so we can easily compare their finish times to intervals in $X$.
Lemma 4.2 (Greedy stays ahead)

Greedy’s m-th interval has finishing time ≤ Optimal’s m-th interval

Proof

Initial case m = 1: We have $f_0 = f_0$, since the greedy algorithm begins by choosing $i = 1$. $(A_1$ has the earliest finishing time.)

Induction assumption $f_m \leq f_m$. Consider $A_m$ and $A_m$. We have $f_m \leq f_m$. (By induction)

Therefore, $f_m \leq f_m$.

Greedy’s $(m-1)$-th interval has earliest finishing time of any interval that starts after $f_{m-1}$ finishes. Therefore $f_m \leq f_m$.

Correctness Proof (cont.)

Recall Greedy solution is $X = (A_1, \ldots, A_k)$.

Optimal solution is $O = (A_1, \ldots, A_k)$.

Now we complete the proof.

From the Lemma, we have $f_m \leq f_m$.

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

No interval finishes strictly to the left

No interval starts strictly to the right

No interval in $X$ is strictly between these points!

Thus, every interval contains some finishing time in $F$.

And, two intervals in $F$ cannot contain the same element of $F$.

So, in addition to the intervals in $F$, only the following types of intervals are possible:

- Contains $f_i$
- Contains $f_j$ and $f_k$

Thus, every interval contains some finishing time in $F$.

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

Problem: Interval Colouring

Instance: A set $A = \{A_1, \ldots, A_k\}$ of intervals.

For $1 \leq i < n$, $A_i = (f_i, \ell_i)$, where $f_i$ is the start time of interval $A_i$ and $\ell_i$ is the finish time of $A_i$.

Feasible solution: A $c$-colouring is a mapping $c: A \rightarrow \{1, \ldots, c\}$ that assigns each interval a colour such that two intervals receiving the same colour are always disjoint.

Find: A $c$-colouring of $A$ with minimum number of colours.

Example

| 1 | 2 | 3 | 4 | 5 |

7 intervals, 5 colours.

Feasible, but not optimal.
MORE EXAMPLES

<table>
<thead>
<tr>
<th>Example</th>
<th>Not feasible!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>7 intervals, 6 colours. Feasible, but not optimal</td>
</tr>
<tr>
<td>Example</td>
<td>7 intervals, 2 colours. Optimal</td>
</tr>
</tbody>
</table>

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 \( i < 11 \) intervals using \( d \) colours.

We will colour the \((i + 1)\)st 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 \((i + 1)\)st 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!

Consider intervals in the order they are given in the input: \( A_1, A_2, \ldots, A_{10} \)

EXAMPLE: ORDER MATTERS!

EXAMPLE: ORDER MATTERS!
EXAMPLE: ORDER MATTERS!

EXAMPLE: ORDER MATTERS!

EXAMPLE: ORDER MATTERS!

EXAMPLE: ORDER MATTERS!

EXAMPLE: ORDER MATTERS!

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!

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

Used 3 colours

Can we do better?

Example:

ORDER MATTERS!

Used 3 colours

Can we do better?

Example:

RUNNING GREEDY

Initial state

Example:

RUNNING GREEDY

i=1

d=1

finish[i]=

Example:

RUNNING GREEDY

i=2

d=2

finish[]+

Example:

RUNNING GREEDY

i=2

d=2

finish[]+

Example:

RUNNING GREEDY

i=2

d=2

finish[]+

Example:

RUNNING GREEDY

i=2

d=2

finish[]+
While loop over: Check if we can reuse a color in $1..d$

GREEDY

Check if we can reuse colour 1.

Is $\text{finish}[1] \leq i$?
No, we cannot reuse colour 1.

Is $\text{finish}[2] \leq i$?
No, we cannot reuse colour 2.

Cannot reuse any colour. Create new one.

Example: Running Greedy

- $i=3$, $d=2$
- $\text{finish}[1]=6$
- $\text{finish}[2]=18$

Example: Running Greedy

- $i=3$, $d=3$
- $\text{finish}[1]=6$
- $\text{finish}[2]=18$

Example: Running Greedy

- $i=4$, $d=3$
- $\text{finish}[1]=3$
- $\text{finish}[2]=18$

Example: Running Greedy

- $i=5$, $d=3$
- $\text{finish}[1]=3$
- $\text{finish}[2]=18$

Example: Running Greedy

- $i=3$, $d=3$
- $\text{finish}[1]=6$
- $\text{finish}[2]=18$

Example: Running Greedy

- $i=3$, $d=3$
- $\text{finish}[1]=6$
- $\text{finish}[2]=18$
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 $P_0$ be the first interval that has the last colour $D$.

Let $s_k$ be the last interval that has colour $c$ and starts before $P_0$ ends.

We prove $P_0$ overlaps every interval $s_k$ for all $k < D$.

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

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 $O(n^2)$ if $n$ colours are needed

IMPROVING THIS ALGORITHM

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

Min element: NULL

Heap:

Initial state

- **Iteration** $i=1$
  - Check heap minimum
  - Empty, so a new colour is needed

Finish at time 3

- **Iteration** $i=2$
  - Check heap minimum
  - Check if finish time 3 is before $s_2$
  - No. New colour!

Finish at time 7

- **Iteration** $i=3$
  - Check heap minimum
  - Check if finish time 3 is before $s_3$
  - No. New colour!

Finish at time 7
EXAMPLE: HEAP-BASED ALGORITHM

Iteration 3
Check heap minimum
Check if finish time 3 is before time 4
No. New colour!

Min element: time 3
Heap: time 6
Finish time: time 5

Check heap minimum
Heap: Min element: NULL

Check if finish time 3 is before time 4
Yes.
Reuse colour, deleteMin and insert new finish time into heap!

Iteration 4
Check heap minimum
Check if finish time 3 is before time 4
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Min element: time 5
Heap: time 7
Finish time: time 5

Iteration 5
Check heap minimum
Check if finish time 5 is before time 4
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Min element: time 5
Heap: time 7
Finish time: time 5

Iteration 6
Check heap minimum
Check if finish time 5 is before time 4
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Min element: time 5
Heap: time 7
Finish time: time 5

Iteration 7
Check heap minimum
Check if finish time 5 is before time 4
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Min element: time 5
Heap: time 7
Finish time: time 5

Iteration 8
Check heap minimum
Check if finish time 5 is before time 4
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Min element: time 5
Heap: time 7
Finish time: time 5

Iteration 9
Check heap minimum
Check if finish time 5 is before time 4
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Min element: time 5
Heap: time 7
Finish time: time 5

Iteration 10
Check heap minimum
Check if finish time 5 is before time 4
Yes. Reuse colour, deleteMin and insert new finish time into heap!

Min element: time 5
Heap: time 7
Finish time: time 5
**EXAMPLE: HEAP-BASED ALGORITHM**

Initialization:
- Min elements: A1, A2, A3, A4, A5, A6, A7, A8, A9, A10
- Heap: A1, A2, A3, A4, A5, A6, A7, A8, A9, A10

Iteration 1:
- Check heap minimum
- Yes, reuse colour, deleteMin and insert new finish time into heap!

Iteration 2:
- Check if finish time 5 is before $s_5$
- Iteration $i = 5$
- Finish at time 7

Iteration 3:
- Check heap minimum
- Yes, reuse colour, deleteMin and insert new finish time into heap!

Iteration 4:
- Check if finish time 7 is before $s_6$
- Iteration $i = 6$
- Finish at time 9

Iteration 5:
- Check heap minimum
- Yes, reuse colour, deleteMin and insert new finish time into heap!

Iteration 6:
- Check if finish time 7 is before $s_6$
- Iteration $i = 6$
- Finish at time 11

And so on, and so forth...

**Preprocess($T[1..n]$):**
- sort $T$ by increasing start time
- let $s[1..n]$ be the finish times in $T$
- return GreedyIntervalColours($T[1..n], f[1..n]$)

**GreedyIntervalColours($T[1..n], f[1..n]$):**
- $d[1] = 1$
- $h[1] = new node$
- for $i = 2$ to $n$
  - $(f[i], h[1].colour[i]) = h[1].colour[i - 1]$
  - if $(f[i], h[1].colour[i])$
    - $h[1].colour[i] = c$
  - else
    - $h[1].colour[i] = d$
- return $d$

**Complexity:**
- $O(lg(S))$ where $S = \text{size(priority queue)}$
- Total $O(n \cdot lg(n)) + O(n \cdot lg(D))$
- Since $n \geq D$, $O(n \cdot lg(S))$