

# CS 577 - Greedy

Marc Renault

Department of Computer Sciences  
University of Wisconsin – Madison

Spring 2021

TopHat Join Code: 524741



# GREEDY

# GREEDY ALGORITHMS

## What is a Greedy Algorithm (GREEDY)?

- Typically, thought of as a *heuristic* that is locally optimal.

# GREEDY ALGORITHMS

## What is a Greedy Algorithm (GREEDY)?

- Typically, thought of as a *heuristic* that is locally optimal.
- Is GREEDY always the best?

# GREEDY ALGORITHMS

## What is a Greedy Algorithm (GREEDY)?

- Typically, thought of as a *heuristic* that is locally optimal.
- Is GREEDY always the best? No, but a good place to start.

# GREEDY ALGORITHMS

## What is a Greedy Algorithm (GREEDY)?

- Typically, thought of as a *heuristic* that is locally optimal.
- Is GREEDY always the best? No, but a good place to start.
- This notion has yet to be fully formalized, and it often problem specific.

# GREEDY ALGORITHMS

## What is a Greedy Algorithm (GREEDY)?

- Typically, thought of as a *heuristic* that is locally optimal.
- Is GREEDY always the best? No, but a good place to start.
- This notion has yet to be fully formalized, and it often problem specific.

## Definition from Priority Algorithms

A greedy algorithm is an algorithm that processes the input in a specified order. For each request in the input, the greedy algorithm processes it so as to minimize (resp. maximize) the objective, assuming that the request is the last request.

# GREEDY ALGORITHMS

## What is a Greedy Algorithm (GREEDY)?

- Typically, thought of as a *heuristic* that is locally optimal.
- Is GREEDY always the best? No, but a good place to start.
- This notion has yet to be fully formalized, and it often problem specific.

## Definition from Priority Algorithms

A greedy algorithm is an algorithm that processes the input in a specified order. For each request in the input, the greedy algorithm processes it so as to minimize (resp. maximize) the objective, assuming that the request is the last request.

For a given problem, there may be many greedy algorithms.



# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$
- FFI:

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$
- FFI: 3 bins

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$
- FFI: 3 bins
- OPT:



# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$
- FFI: 3 bins
- OPT: 2 bins

# IS GREEDY OPTIMAL?

## Not always: Bin Packing Problem

- Bins of size 1, and requests of size  $(0, 1]$ .
- Objective: Pack the items in the minimum number of bins.
- Greedy heuristic: FIRST FIT INCREASING (FFI)

Non-optimal example:

- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$
- FFI: 3 bins
- OPT: 2 bins

Techniques for showing that GREEDY is optimal:

- Always stays ahead
- Exchange argument

# SHORTEST PATH

# FINDING THE SHORTEST PATH

## Problem Definition

We have a directed graph  $G = (V, E)$ , where  $|V| = n$  and  $|E| = m$  and a node  $s$  that has a path to every other node in  $V$ . For each edge  $e$ ,  $\ell_e \geq 0$  is the length of the edge.

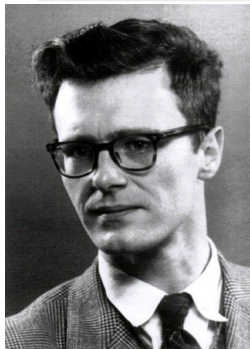
- What is the shortest path from  $s$  to each other node?

# FINDING THE SHORTEST PATH

## Problem Definition

We have a directed graph  $G = (V, E)$ , where  $|V| = n$  and  $|E| = m$  and a node  $s$  that has a path to every other node in  $V$ . For each edge  $e$ ,  $\ell_e \geq 0$  is the length of the edge.

- What is the shortest path from  $s$  to each other node?



Edsger Dijkstra, 1956  
Dijkstra's shortest path fame

# DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.

For each  $u \in S$ , we store a distance value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one incoming edge  
    originating from a node in  $S$  with the smallest

$$d'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

    Append  $v$  to  $S$  and define  $d(v) = d'(v)$ .

**end**

---

# DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.

For each  $u \in S$ , we store a distance value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one incoming edge  
    originating from a node in  $S$  with the smallest

$$d'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

    Append  $v$  to  $S$  and define  $d(v) = d'(v)$ .

**end**

---

How is it greedy?

# DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.

For each  $u \in S$ , we store a distance value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one incoming edge  
    originating from a node in  $S$  with the smallest

$$d'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

    Append  $v$  to  $S$  and define  $d(v) = d'(v)$ .

**end**

---

How is it greedy?

TopHat 3: Which technique to prove optimality?



# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

## Proof.



# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

## Proof.

By induction on the size of  $S$ .



# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

## Proof.

By induction on the size of  $S$ .

- For  $|S| = 1$ , the claim follows trivially as  $S = \{s\}$ .

# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

## Proof.

By induction on the size of  $S$ .

- For  $|S| = 1$ , the claim follows trivially as  $S = \{s\}$ .
- By the induction hypothesis, for  $|S| = k$ ,  $P_u$  is the shortest  $s - u$  path for all  $u \in S$ .

# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

## Proof.

By induction on the size of  $S$ .

- In step  $k + 1$ , we add  $v$ .
  - By definition,  $P_v$  is shortest path connected to  $S$  by one edge.



# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

## Proof.

By induction on the size of  $S$ .

- In step  $k + 1$ , we add  $v$ .
  - By definition,  $P_v$  is shortest path connected to  $S$  by one edge.
  - Since  $P_u$  is a shortest path to  $u$ ,  $P_v$  is the shortest path to  $v$  when considering only the nodes of  $S$ .



# CORRECTNESS OF DIJKSTRA'S

## Theorem 1

*Consider the  $S$  at any point in the execution of Dijkstra's. For each  $u \in S$ , the path  $P_u$  is a shortest  $s - u$  path.*

## Proof.

By induction on the size of  $S$ .

- In step  $k + 1$ , we add  $v$ .
  - By definition,  $P_v$  is shortest path connected to  $S$  by one edge.
  - Since  $P_u$  is a shortest path to  $u$ ,  $P_v$  is the shortest path to  $v$  when considering only the nodes of  $S$ .
  - Moreover, there cannot be a shorter path to  $v$  passing through another node  $y \notin S$  else  $y$  that would be added at  $k + 1$ .





# DIJKSTRA'S OBSERVATIONS

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- Negative edge weights, where does it fail?

# DIJKSTRA'S OBSERVATIONS

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- Negative edge weights, where does it fail?
- TopHat 4: It is graph exploration, what kind of exploration?

# DIJKSTRA'S OBSERVATIONS

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- Negative edge weights, where does it fail?
- TopHat 4: It is graph exploration, what kind of exploration?
  - Weighted (continuous) BFS

# IMPLEMENTATION AND RUN TIME OF DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- TopHat 5:  
Number of  
iterations of  
the loop?

# IMPLEMENTATION AND RUN TIME OF DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- TopHat 5:  
Number of  
iterations of  
the loop?  
 $n - 1$

# IMPLEMENTATION AND RUN TIME OF DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- TopHat 5:  
Number of  
iterations of  
the loop?  
 $n - 1$
- Key  
Operations:

# IMPLEMENTATION AND RUN TIME OF DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- TopHat 5:  
Number of  
iterations of  
the loop?  
 $n - 1$
- Key  
Operations:
  - Finding  
the min:  
Easy in  
 $O(m)$

# IMPLEMENTATION AND RUN TIME OF DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- TopHat 5:  
Number of  
iterations of  
the loop?  
 $n - 1$
- Key  
Operations:
  - Finding  
the min:  
Easy in  
 $O(m)$
- Overall:  
 $O(mn)$



# IMPLEMENTATION AND RUN TIME OF DIJKSTRA'S

---

**Algorithm:** *Dijkstra's*

---

Let  $S$  be the set of explored nodes.  
For each  $u \in S$ , we store a distance  
value  $d(u)$ .

Initialize  $S = \{s\}$  and  $d(s) = 0$

**while**  $S \neq V$  **do**

    Choose  $v \notin S$  with at least one  
    incoming edge originating  
    from a node in  $S$  with the  
    smallest  $d'(v) =$

$\min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$

    Append  $v$  to  $S$  and define  
     $d(v) = d'(v)$ .

**end**

---

- TopHat 5:  
Number of  
iterations of  
the loop?  
 $n - 1$
- Key  
Operations:
  - Finding  
the min:  
Easy in  
 $O(m)$
- Overall:  
 $O(mn)$
- How can we  
get  
 $O(m \log n)$ ?

# MST

# MINIMUM SPANNING TREE PROBLEM

## MST Problem

Let  $G = (V, E)$  be a connected graph, where  $|V| = n$  and  $|E| = m$ . For each edge  $e$ ,  $c_e > 0$  is the cost of the edge.

- Find an edge set  $F \subseteq E$  with minimum cost that keeps the graph connected. That is,  $F$  should minimize  $\sum_{e \in F} c_e$ .

# MINIMUM SPANNING TREE PROBLEM

## MST Problem

Let  $G = (V, E)$  be a connected graph, where  $|V| = n$  and  $|E| = m$ . For each edge  $e$ ,  $c_e > 0$  is the cost of the edge.

- Find an edge set  $F \subseteq E$  with minimum cost that keeps the graph connected. That is,  $F$  should minimize  $\sum_{e \in F} c_e$ .

## Observation 1

*Let  $T = (V, F)$  be a minimum-cost solution to the problem described above. Then,  $T$  is a tree.*

# MINIMUM SPANNING TREE PROBLEM

## MST Problem

Let  $G = (V, E)$  be a connected graph, where  $|V| = n$  and  $|E| = m$ . For each edge  $e$ ,  $c_e > 0$  is the cost of the edge.

- Find an edge set  $F \subseteq E$  with minimum cost that keeps the graph connected. That is,  $F$  should minimize  $\sum_{e \in F} c_e$ .

## Observation 1

*Let  $T = (V, F)$  be a minimum-cost solution to the problem described above. Then,  $T$  is a tree.*

## Proof.

# MINIMUM SPANNING TREE PROBLEM

## MST Problem

Let  $G = (V, E)$  be a connected graph, where  $|V| = n$  and  $|E| = m$ . For each edge  $e$ ,  $c_e > 0$  is the cost of the edge.

- Find an edge set  $F \subseteq E$  with minimum cost that keeps the graph connected. That is,  $F$  should minimize  $\sum_{e \in F} c_e$ .

## Observation 1

*Let  $T = (V, F)$  be a minimum-cost solution to the problem described above. Then,  $T$  is a tree.*

## Proof.

- By the definition of the problem,  $T$  must be connected.
- By way of contradiction, assume that  $T$  has a cycle  $C$ . Remove any edge from  $C$  resulting in a graph  $T'$ .  $T'$  is still connect and has a cost less than  $T$ .



# ALGORITHM DESIGN

TopHat Discussion 3: What greedy heuristic might work?

# ALGORITHM DESIGN

TopHat Discussion 3: What greedy heuristic might work?

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.



# ALGORITHM DESIGN

TopHat Discussion 3: What greedy heuristic might work?

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Jarník's (1929), Kruskal's (1956), Prim's (1957), Loberman and Weinberger (1957), Dijkstra's (1958) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

# ALGORITHM DESIGN

TopHat Discussion 3: What greedy heuristic might work?

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

# ALGORITHM DESIGN

TopHat Discussion 3: What greedy heuristic might work?

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Kruskal's (1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

# ALGORITHM DESIGN

TopHat Discussion 3: What greedy heuristic might work?

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Reverse-Delete (Kruskal's 1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

# ASSUME DISTINCT WEIGHTS

WLOG (WITHOUT LOSS OF GENERALITY)

## Theorem 2

*(HW Q2) If all edge weights in a connected graph are distinct, then  $G$  has a unique MST.*

# ASSUME DISTINCT WEIGHTS

WLOG (WITHOUT LOSS OF GENERALITY)

## Theorem 2

*(HW Q2) If all edge weights in a connected graph are distinct, then  $G$  has a unique MST.*

## Observation 2

*All we need is a consistent tie-breaker when  $c_{e_1} = c_{e_2}$  for some pair of edges. I.e. based on the labels of the vertices of  $e_1 \cup e_2$ .*

# ASSUME DISTINCT WEIGHTS

WLOG (WITHOUT LOSS OF GENERALITY)

## Theorem 2

*(HW Q2) If all edge weights in a connected graph are distinct, then  $G$  has a unique MST.*

## Observation 2

*All we need is a consistent tie-breaker when  $c_{e_1} = c_{e_2}$  for some pair of edges. I.e. based on the labels of the vertices of  $e_1 \cup e_2$ .*

Assumption: all edge weights are distinct.

# ANALYZING MST HEURISTICS

## Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*



# ANALYZING MST HEURISTICS

## Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*

## Proof.

# ANALYZING MST HEURISTICS

## Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*

## Proof.

By exchange argument:

# ANALYZING MST HEURISTICS

## Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*

## Proof.

By exchange argument:

- Let  $T$  be a spanning tree that does not contain  $e$ .



## ANALYZING MST HEURISTICS

### Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*

### Proof.

By exchange argument:

- Let  $T$  be a spanning tree that does not contain  $e$ .
- Let  $e' = (v', w')$ , where  $e'$  is in  $P_{v,w} \in T$ ,  $v' \in S$ , and  $w' \in V \setminus S$ .



# ANALYZING MST HEURISTICS

## Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*

## Proof.

By exchange argument:

- Let  $T$  be a spanning tree that does not contain  $e$ .
- Let  $e' = (v', w')$ , where  $e'$  is in  $P_{v,w} \in T$ ,  $v' \in S$ , and  $w' \in V \setminus S$ .
- Let  $T' = T \setminus e' \cup e$ .



# ANALYZING MST HEURISTICS

## Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*

## Proof.

By exchange argument:

- Let  $T$  be a spanning tree that does not contain  $e$ .
- Let  $e' = (v', w')$ , where  $e'$  is in  $P_{v,w} \in T$ ,  $v' \in S$ , and  $w' \in V \setminus S$ .
- Let  $T' = T \setminus e' \cup e$ .
- $T'$  is connected as  $e$  is a  $P_{v,w} \in T'$ .



# ANALYZING MST HEURISTICS

## Lemma 3

*Let  $S \subset V$  be a non-empty proper subset of the nodes, and let  $e = (v, w)$  be the minimum cost edge connecting  $S$  and  $V \setminus S$ . Then, every MST contains  $e$ .*

## Proof.

By exchange argument:

- Let  $T$  be a spanning tree that does not contain  $e$ .
- Let  $e' = (v', w')$ , where  $e'$  is in  $P_{v,w} \in T$ ,  $v' \in S$ , and  $w' \in V \setminus S$ .
- Let  $T' = T \setminus e' \cup e$ .
- $T'$  is connected as  $e$  is a  $P_{v,w} \in T'$ .
- Since  $c_e < c_{e'}$ , cost of  $T'$  is less than  $T$ .



# KRUSKAL'S ALGORITHM IS OPTIMAL

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Theorem 4

*Kruskal's Algorithm produces an MST.*



# KRUSKAL'S ALGORITHM IS OPTIMAL

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Theorem 4

*Kruskal's Algorithm produces an MST.*

## Proof.



# KRUSKAL'S ALGORITHM IS OPTIMAL

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Theorem 4

*Kruskal's Algorithm produces an MST.*

## Proof.

- Let  $e = (v, w)$  be the edge added at any step  $i$ .



# KRUSKAL'S ALGORITHM IS OPTIMAL

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Theorem 4

*Kruskal's Algorithm produces an MST.*

## Proof.

- Let  $e = (v, w)$  be the edge added at any step  $i$ .
- Since  $e$  does not create a cycle,  $v \in S$  and  $w \notin S$  (WLOG).



# KRUSKAL'S ALGORITHM IS OPTIMAL

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Theorem 4

*Kruskal's Algorithm produces an MST.*

## Proof.

- Let  $e = (v, w)$  be the edge added at any step  $i$ .
- Since  $e$  does not create a cycle,  $v \in S$  and  $w \notin S$  (WLOG).
- As  $c_e$  is the minimum cost edge, the claim follows from Lemma 3.



# PRIM'S ALGORITHM IS OPTIMAL

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Theorem 5

*Prim's Algorithm produces an MST.*

# PRIM'S ALGORITHM IS OPTIMAL

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Theorem 5

*Prim's Algorithm produces an MST.*

## Proof.

# PRIM'S ALGORITHM IS OPTIMAL

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Theorem 5

*Prim's Algorithm produces an MST.*

## Proof.

- Immediate from Lemma 3.



# PRIM'S ALGORITHM IS OPTIMAL

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Theorem 5

*Prim's Algorithm produces an MST.*

## Proof.

- Immediate from Lemma 3.
- That is, Prim's algorithm does exactly what Lemma 3 describes.





# REVERSE-DELETE IS OPTIMAL

## Reverse-Delete (Kruskal's 1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

How should we prove that it produces an MST?

# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Proof.



## REVERSE-DELETE IS OPTIMAL

### Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

### Proof.

- Let  $T$  be a spanning tree that does contain  $e$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Proof.

- Let  $T$  be a spanning tree that does contain  $e$ .
- Let  $S$  and  $V \setminus S$  be the nodes of the connected components after removing  $e$  from  $T$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Proof.

- Let  $T$  be a spanning tree that does contain  $e$ .
- Let  $S$  and  $V \setminus S$  be the nodes of the connected components after removing  $e$  from  $T$ .
- Let  $e'$  be an edge in  $C$  that connects  $S$  and  $V \setminus S$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Proof.

- Let  $T$  be a spanning tree that does contain  $e$ .
- Let  $S$  and  $V \setminus S$  be the nodes of the connected components after removing  $e$  from  $T$ .
- Let  $e'$  be an edge in  $C$  that connects  $S$  and  $V \setminus S$ .
- Let  $T' = T \setminus e \cup e'$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Proof.

- Let  $T$  be a spanning tree that does contain  $e$ .
- Let  $S$  and  $V \setminus S$  be the nodes of the connected components after removing  $e$  from  $T$ .
- Let  $e'$  be an edge in  $C$  that connects  $S$  and  $V \setminus S$ .
- Let  $T' = T \setminus e \cup e'$ .
- $T'$  is connected as  $e'$  reconnects  $S$  and  $V \setminus S$ .





## REVERSE-DELETE IS OPTIMAL

### Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

### Proof.

- Let  $T$  be a spanning tree that does contain  $e$ .
- Let  $S$  and  $V \setminus S$  be the nodes of the connected components after removing  $e$  from  $T$ .
- Let  $e'$  be an edge in  $C$  that connects  $S$  and  $V \setminus S$ .
- Let  $T' = T \setminus e \cup e'$ .
- $T'$  is connected as  $e'$  reconnects  $S$  and  $V \setminus S$ .
- Since  $c_e > c_{e'}$ , cost of  $T'$  is less than  $T$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Theorem 7

*Reverse-Delete Algorithm produces an MST.*

# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Theorem 7

*Reverse-Delete Algorithm produces an MST.*

## Proof.



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Theorem 7

*Reverse-Delete Algorithm produces an MST.*

## Proof.

- Let  $e = (v, w)$  be an edge removed at any step  $i$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Theorem 7

*Reverse-Delete Algorithm produces an MST.*

## Proof.

- Let  $e = (v, w)$  be an edge removed at any step  $i$ .
- By definition  $e$ , belongs to a cycle  $C$ .



# REVERSE-DELETE IS OPTIMAL

## Lemma 6

*Let  $C$  be any cycle in  $G$ , and let  $e$  be the most expensive edge of  $C$ . Then,  $e$  is not in any MST of  $G$ .*

## Theorem 7

*Reverse-Delete Algorithm produces an MST.*

## Proof.

- Let  $e = (v, w)$  be an edge removed at any step  $i$ .
- By definition  $e$ , belongs to a cycle  $C$ .
- As  $c_e$  is the maximum cost edge of  $C$ , the claim follows from Lemma 6.



# IMPLEMENTING PRIM'S ALGORITHM

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Key Operations

# IMPLEMENTING PRIM'S ALGORITHM

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Key Operations

- Retrieve the minimum valued edge between  $S$  and  $V \setminus S$ .



# IMPLEMENTING PRIM'S ALGORITHM

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Key Operations

- Retrieve the minimum valued edge between  $S$  and  $V \setminus S$ .
- Prim's and Dijkstra's have nearly identical implementations (but different minimizers)!

# IMPLEMENTING PRIM'S ALGORITHM

## Prim's (1957) Algorithm

- Initialize a node set  $S$  with an arbitrary node  $s$ .
- Keep the least expensive edge as long as it does not create a cycle.

## Key Operations

- Retrieve the minimum valued edge between  $S$  and  $V \setminus S$ .
- Prim's and Dijkstra's have nearly identical implementations (but different minimizers)!

## Priority Queue (min-heap)

- ExtractMin ( $O(1)$ ):  $n - 1$  times.
- ChangeKey ( $O(\log(n))$ ):  $m$  times.

Overall:  $O(m \log(n))$

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges: ( $O(m \log m)$  and, since  $m \leq n^2$ ,  $O(m \log n)$ ).

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges: ( $O(m \log m)$  and, since  $m \leq n^2$ ,  $O(m \log n)$ ).
- Maintain sets of connected components that we merge.

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges: ( $O(m \log m)$  and, since  $m \leq n^2$ ,  $O(m \log n)$ ).
- Maintain sets of connected components that we merge.
- Initialize one set per node:  $O(n)$ .

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges: ( $O(m \log m)$  and, since  $m \leq n^2$ ,  $O(m \log n)$ ).
- Maintain sets of connected components that we merge.
- Initialize one set per node:  $O(n)$ .

## Union-Find Data Structure

- Find( $x$ ): Finds the set containing  $x$ . ( $O(\log n)$  can be  $O(\alpha(n))$ )
- Union( $x, y$ ): Joins two sets  $x$  and  $y$ . ( $O(1)$ )

# UNION-FIND / DISJOINT-SET

## Key Operations

- $\text{Find}(x)$ : Finds the set containing  $x$ . ( $O(\log n)$  can be  $O(\alpha(n))$ )
- $\text{Union}(x, y)$ : Joins two sets  $x$  and  $y$ . ( $O(1)$ )



# UNION-FIND / DISJOINT-SET

## Key Operations

- $\text{Find}(x)$ : Finds the set containing  $x$ . ( $O(\log n)$  can be  $O(\alpha(n))$ )
- $\text{Union}(x, y)$ : Joins two sets  $x$  and  $y$ . ( $O(1)$ )

## Basic Container

|      |      |        |
|------|------|--------|
| node | rank | parent |
|------|------|--------|

# UNION-FIND / DISJOINT-SET

## Key Operations

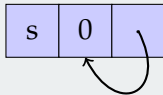
- Find( $x$ ): Finds the set containing  $x$ . ( $O(\log n)$  can be  $O(\alpha(n))$ )
- Union( $x, y$ ): Joins two sets  $x$  and  $y$ . ( $O(1)$ )

## Basic Container

|      |      |        |
|------|------|--------|
| node | rank | parent |
|------|------|--------|

## Initializing Data Structure for Kruskal's

For each node  $s$ , create a singleton set. That is each container has rank 0 and points to itself.



# UNION-FIND OPERATIONS

Find( $x$ ):  $O(\log n)$

# UNION-FIND OPERATIONS

**Find( $x$ ):  $O(\log n)$**

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else Find( $x.\text{parent}$ )

# UNION-FIND OPERATIONS

**Find( $x$ ):  $O(\log n)$**

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else Find( $x.\text{parent}$ )
- $O(\log n)$  requires balanced trees.

# UNION-FIND OPERATIONS

**Find( $x$ ):  $O(\log n)$**

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else Find( $x.\text{parent}$ )
- $O(\log n)$  requires balanced trees.
- $O(\alpha(n))$  with *path compression*.

# UNION-FIND OPERATIONS

**Find( $x$ ):  $O(\log n)$**

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else Find( $x.\text{parent}$ )
- $O(\log n)$  requires balanced trees.
- $O(\alpha(n))$  with *path compression*.

**Union( $x, y$ ):  $O(1)$**

# UNION-FIND OPERATIONS

## Find( $x$ ): $O(\log n)$

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else Find( $x.\text{parent}$ )
- $O(\log n)$  requires balanced trees.
- $O(\alpha(n))$  with *path compression*.

## Union( $x, y$ ): $O(1)$

- (WLOG)  $x.\text{rank} \geq y.\text{rank}$ :  
     $y.\text{parent} = x$



# UNION-FIND OPERATIONS

## Find( $x$ ): $O(\log n)$

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else Find( $x.\text{parent}$ )
- $O(\log n)$  requires balanced trees.
- $O(\alpha(n))$  with *path compression*.

## Union( $x, y$ ): $O(1)$

- (WLOG)  $x.\text{rank} \geq y.\text{rank}$ :  
   $y.\text{parent} = x$
- If  $x.\text{rank} = y.\text{rank}$ :  
   $x.\text{rank} := x.\text{rank} + 1$

# UNION-FIND OPERATIONS

## Find( $x$ ): $O(\log n)$

- If  $x.\text{parent}$  points to  $x$ , return  $x$ .
- Else Find( $x.\text{parent}$ )
- $O(\log n)$  requires balanced trees.
- $O(\alpha(n))$  with *path compression*.

## Union( $x, y$ ): $O(1)$

- (WLOG)  $x.\text{rank} \geq y.\text{rank}$ :  
 $y.\text{parent} = x$
- If  $x.\text{rank} = y.\text{rank}$ :  
 $x.\text{rank} := x.\text{rank} + 1$
- By using rank, we maintain balanced sets if we start with balanced sets.

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges: ( $O(m \log m)$  and, since  $m \leq n^2$ ,  $O(m \log n)$ ).
- Maintain sets of connected components that we merge.
- Initialize one set per node:  $O(n)$ .

## Union-Find Data Structure

### TH: How many Find and Unions?

- Find( $x$ ): Finds the set containing  $x$ .
- Union( $x, y$ ): Joins two sets  $x$  and  $y$ .

# IMPLEMENTING KRUSKAL'S ALGORITHM

## Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

## Key Operations

- Sorting the edges: ( $O(m \log m)$  and, since  $m \leq n^2$ ,  $O(m \log n)$ ).
- Maintain sets of connected components that we merge.
- Initialize one set per node:  $O(n)$ .

## Union-Find Data Structure

- Find( $x$ ):  $2m$  times  $O(\log n)$  (can be  $O(\alpha(n))$ ).
- Union( $x, y$ ):  $n - 1$  times  $O(1)$ .

# GRAPH EXPLORATION OVERVIEW

## BFS and DFS

- Traverses a graph  $G$  starting from some node  $s$ .
- Builds a tree  $T$ .
- No guarantee on any distance measure.

# GRAPH EXPLORATION OVERVIEW

## BFS and DFS

- Traverses a graph  $G$  starting from some node  $s$ .
- Builds a tree  $T$ .
- No guarantee on any distance measure.

## Dijkstra's

- Traverses a graph starting from some node  $s$ .
- Builds a tree  $T$ .
- All  $s$  to  $u$  paths in  $T$  are the shortest such path in  $G$ .

# GRAPH EXPLORATION OVERVIEW

## BFS and DFS

- Traverses a graph  $G$  starting from some node  $s$ .
- Builds a tree  $T$ .
- No guarantee on any distance measure.

## Dijkstra's

- Traverses a graph starting from some node  $s$ .
- Builds a tree  $T$ .
- All  $s$  to  $u$  paths in  $T$  are the shortest such path in  $G$ .

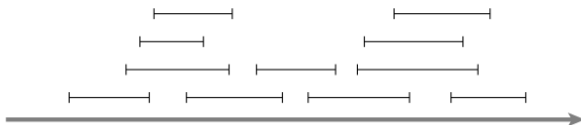
## MST Algorithms

- Explores a graph  $G$  edges.
- Builds a tree  $T$ .
- $T$  is minimum cost to connect all nodes in  $G$ .

# STAYS AHEAD: INTERVAL SCHEDULING



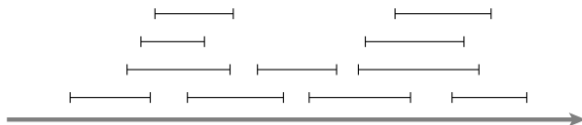
# INTERVAL SCHEDULING



## Problem Definition

- Requests:  $\sigma = \{r_1, \dots, r_n\}$

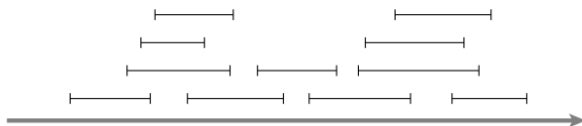
# INTERVAL SCHEDULING



## Problem Definition

- Requests:  $\sigma = \{r_1, \dots, r_n\}$
- A request  $r_i = (s_i, f_i)$ , where  $s_i$  is the start time and  $f_i$  is the finish time.

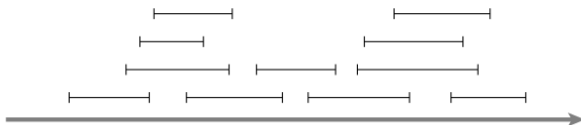
# INTERVAL SCHEDULING



## Problem Definition

- Requests:  $\sigma = \{r_1, \dots, r_n\}$
- A request  $r_i = (s_i, f_i)$ , where  $s_i$  is the start time and  $f_i$  is the finish time.
- Objective: Produce a *compatible* schedule  $S$  that has maximum cardinality.

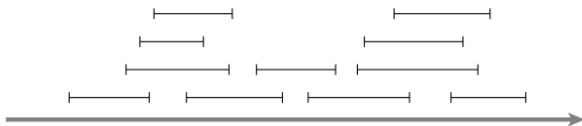
# INTERVAL SCHEDULING



## Problem Definition

- Requests:  $\sigma = \{r_1, \dots, r_n\}$
- A request  $r_i = (s_i, f_i)$ , where  $s_i$  is the start time and  $f_i$  is the finish time.
- Objective: Produce a *compatible* schedule  $S$  that has maximum cardinality.
- Compatible schedule  $S$ :  $\forall r_i, r_j \in S, f_i \leq s_j \vee f_j \leq s_i$ .

# INTERVAL SCHEDULING



## Problem Definition

- Requests:  $\sigma = \{r_1, \dots, r_n\}$
- A request  $r_i = (s_i, f_i)$ , where  $s_i$  is the start time and  $f_i$  is the finish time.
- Objective: Produce a *compatible* schedule  $S$  that has maximum cardinality.
- Compatible schedule  $S$ :  $\forall r_i, r_j \in S, f_i \leq s_j \vee f_j \leq s_i$ .

TopHat Discussion 1: What greedy heuristic might work?

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 1: Earliest First

Schedule a compatible request with the earliest start time.

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 1: Earliest First

Schedule a compatible request with the earliest start time.

Optimal?

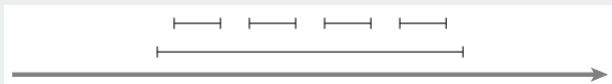
# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 1: Earliest First

Schedule a compatible request with the earliest start time.

## Optimal?

Counter-example:





# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 2: Smallest Interval

Schedule a compatible request  $r_i$  with the smallest interval  $(f_i - s_i)$ .

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 2: Smallest Interval

Schedule a compatible request  $r_i$  with the smallest interval  $(f_i - s_i)$ .

Optimal?

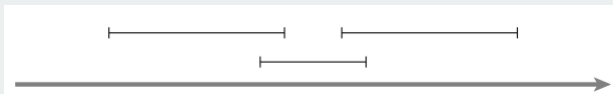
# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 2: Smallest Interval

Schedule a compatible request  $r_i$  with the smallest interval ( $f_i - s_i$ ).

## Optimal?

Counter-example:



# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 3: Fewest Conflicts

Schedule a compatible request with the fewest remaining conflicts.

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 3: Fewest Conflicts

Schedule a compatible request with the fewest remaining conflicts.

Optimal?

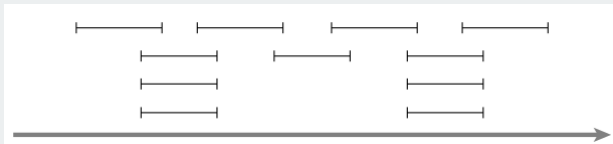
# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 3: Fewest Conflicts

Schedule a compatible request with the fewest remaining conflicts.

## Optimal?

Counter-example:



# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 4: Finish First

Schedule a compatible request with the smallest finish time.

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 4: Finish First

Schedule a compatible request with the smallest finish time.

Optimal?



# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

## Heuristic 4: Finish First

Schedule a compatible request with the smallest finish time.

## Optimal?

Counter-example? Let's try and prove it.

# EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 4: FINISH FIRST

# EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 4: FINISH FIRST

---

**Algorithm:** FINISHFIRST

---

Let  $S$  be an initially empty set.

**while**  $\sigma$  *is not empty* **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

---

# EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 4: FINISH FIRST

---

**Algorithm:** FINISHFIRST

---

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

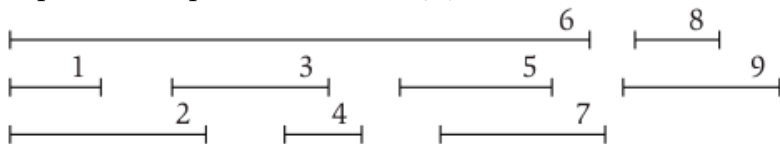
    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

---

Sample Run (TopHat Q1: What is  $|S|$ ?)



# ANALYSIS OF FINISHFIRST

## Observation 3

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

# ANALYSIS OF FINISHFIRST

## Observation 3

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- We can show the strong claim that  $S = S^*$ .

# ANALYSIS OF FINISHFIRST

## Observation 3

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- We can show the strong claim that  $S = S^*$ .
- Can there be multiple  $S^*$ ?

# ANALYSIS OF FINISHFIRST

## Observation 3

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- We can show the strong claim that  $S = S^*$ .
- Can there be multiple  $S^*$ ? Yes.



# ANALYSIS OF FINISHFIRST

## Observation 3

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- We can show the strong claim that  $S = S^*$ .
- Can there be multiple  $S^*$ ? Yes.
- Hence, we can show the weaker claim of  $|S| = |S^*|$  for this problem.

# ANALYSIS OF FINISHFIRST

## Observation 3

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- We can show the strong claim that  $S = S^*$ .
- Can there be multiple  $S^*$ ? Yes.
- Hence, we can show the weaker claim of  $|S| = |S^*|$  for this problem.
- Technique: “Always stays ahead”

# ANALYSIS OF FINISHFIRST

## Observation 3

*Immediate from the definition of FINISHFIRST,  $S$  is compatible.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- We can show the strong claim that  $S = S^*$ .
- Can there be multiple  $S^*$ ? Yes.
- Hence, we can show the weaker claim of  $|S| = |S^*|$  for this problem.
- Technique: “Always stays ahead”
  - At every time step  $i$ ,  $|S_i| \geq |S_i^*|$ .

# STAY AHEAD ANALYSIS

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

## STAY AHEAD ANALYSIS

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

### Lemma 8

*For all  $i_r, j_r$  with  $r \leq k$ , we have  $f_{i_r} \leq f_{j_r}$*

### Proof.

## STAY AHEAD ANALYSIS

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

### Lemma 8

*For all  $i_r, j_r$  with  $r \leq k$ , we have  $f_{i_r} \leq f_{j_r}$*

### Proof.

The proof is by induction.

- For  $r = 1$ , the claim is true as FINISHFIRST first selects the request with the earliest finish time.

## STAY AHEAD ANALYSIS

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

### Lemma 8

*For all  $i_r, j_r$  with  $r \leq k$ , we have  $f_{i_r} \leq f_{j_r}$*

### Proof.

The proof is by induction.

- For  $r = 1$ , the claim is true as FINISHFIRST first selects the request with the earliest finish time.
- Assume true for  $r - 1$ . By the induction hypothesis, we have that  $f_{i_{r-1}} \leq f_{j_{r-1}}$ . The only way for  $S$  to fall behind  $S^*$  would be for FINISHFIRST to choose a request  $q$  with  $f_q \geq f_{j_r}$ , but this would contradict FINISHFIRST as  $j_r \in \sigma$  at this time as implied by the induction hypothesis.

# STAY AHEAD ANALYSIS

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

## Lemma 8

*For all  $i_r, j_r$  with  $r \leq k$ , we have  $f_{i_r} \leq f_{j_r}$*

The optimality of FINISHFIRST, essentially, follows immediately from Lemma 8.



# FINISHPERFECT IS OPTIMAL

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

## Theorem 9

*FINISHPERFECT produces an optimal schedule.*

## Proof.



# FINISHFIRST IS OPTIMAL

- Label  $S = \langle i_1, \dots, i_k \rangle$  such that  $f_{i_u} < f_{i_v}$  for  $u < v$ .
- Label  $S^* = \langle j_1, \dots, j_m \rangle$  such that  $f_{j_u} < f_{j_v}$  for  $u < v$ .

## Theorem 9

*FINISHFIRST produces an optimal schedule.*

## Proof.

By way of contradiction, assume that  $|S^*| > |S|$ . This implies that  $m > k$ . Lemma 8 shows that FINISHFIRST is ahead for all the  $k$  requests. That means it would be able to add the  $(k + 1)$ -st item of  $S^*$ . As it did not, this contradicts the definition of FINISHFIRST. □

# IMPLEMENTATION AND RUNNING TIME

---

**Algorithm:** FINISHFIRST

---

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

---

## Implementation Details

# IMPLEMENTATION AND RUNNING TIME

---

**Algorithm:** FINISHPHIRST

---

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

---

## Implementation Details

- Choose request with smallest finish time:
- Remove incompatible requests:

# IMPLEMENTATION AND RUNNING TIME

---

**Algorithm:** FINISHFIRST

---

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

---

## Implementation Details

- Choose request with smallest finish time: Before processing, sort requests:  $O(n \log n)$ .
- Remove incompatible requests:

# IMPLEMENTATION AND RUNNING TIME

---

**Algorithm:** FINISHFIRST

---

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

---

## Implementation Details

- Choose request with smallest finish time: Before processing, sort requests:  $O(n \log n)$ .
- Remove incompatible requests: Advance in sorted order until a request with a compatible start time.

# IMPLEMENTATION AND RUNNING TIME

---

**Algorithm:** FINISHPHIRST

---

Let  $S$  be an initially empty set.

**while**  $\sigma$  is not empty **do**

    Choose  $r_i \in \sigma$  with the smallest finish time (break ties arbitrarily).

    Add  $r_i$  to  $S$ .

    Remove all incompatible request in  $\sigma$ .

**end**

**return**  $S$

---

## Implementation Details

- Choose request with smallest finish time: Before processing, sort requests:  $O(n \log n)$ .
- Remove incompatible requests: Advance in sorted order until a request with a compatible start time.

Overall:

$$O(n \log n) + O(n) = O(n \log n)$$

# INTERVAL EXTENSIONS

- Online variant: Requests are presented in a specific order to the algorithm. At request  $i$ , the algorithm does not know  $n$  nor  $r_{i+1}, \dots, r_n$ .



# INTERVAL EXTENSIONS

- Online variant: Requests are presented in a specific order to the algorithm. At request  $i$ , the algorithm does not know  $n$  nor  $r_{i+1}, \dots, r_n$ .
- Add a value to the intervals (online/offline). Now objective is to maximize the total value of scheduled intervals.

# INTERVAL EXTENSIONS

- Online variant: Requests are presented in a specific order to the algorithm. At request  $i$ , the algorithm does not know  $n$  nor  $r_{i+1}, \dots, r_n$ .
- Add a value to the intervals (online/offline). Now objective is to maximize the total value of scheduled intervals.
- Scheduling all intervals: Interval Colouring Problem.

# INTERVAL EXTENSIONS

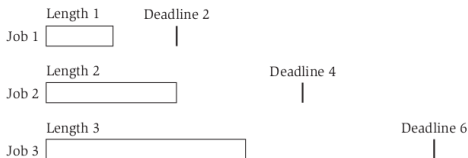
- Online variant: Requests are presented in a specific order to the algorithm. At request  $i$ , the algorithm does not know  $n$  nor  $r_{i+1}, \dots, r_n$ .
- Add a value to the intervals (online/offline). Now objective is to maximize the total value of scheduled intervals.
- Scheduling all intervals: Interval Colouring Problem.
  - Unlimited resources and the algorithm must produce multiple compatible schedules that cover all the requests (without duplicates between the schedules).

# INTERVAL EXTENSIONS

- Online variant: Requests are presented in a specific order to the algorithm. At request  $i$ , the algorithm does not know  $n$  nor  $r_{i+1}, \dots, r_n$ .
- Add a value to the intervals (online/offline). Now objective is to maximize the total value of scheduled intervals.
- Scheduling all intervals: Interval Colouring Problem.
  - Unlimited resources and the algorithm must produce multiple compatible schedules that cover all the requests (without duplicates between the schedules).
  - Objective: Minimize the number of schedules.

# EXCHANGE ARGUMENT: MINIMIZE MAX LATENESS

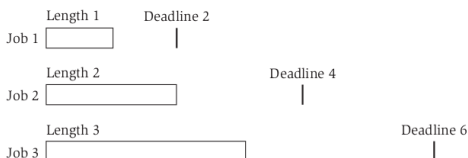
# SCHEDULING PROBLEM: MINIMIZE LATENESS



## Problem Definition

- $n$  jobs and a single machine that can process one job at a time

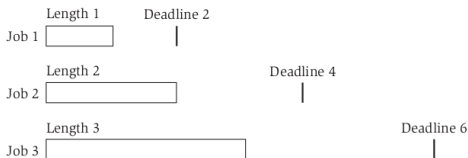
# SCHEDULING PROBLEM: MINIMIZE LATENESS



## Problem Definition

- $n$  jobs and a single machine that can process one job at a time
- For job  $i$ :

# SCHEDULING PROBLEM: MINIMIZE LATENESS

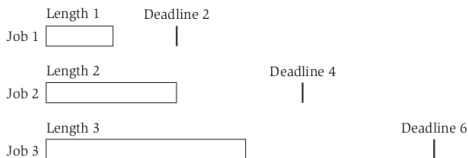


## Problem Definition

- $n$  jobs and a single machine that can process one job at a time
- For job  $i$ :
  - $t_i$  is the processing time,  $d_i$  is the deadline.



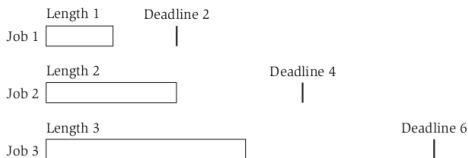
# SCHEDULING PROBLEM: MINIMIZE LATENESS



## Problem Definition

- $n$  jobs and a single machine that can process one job at a time
- For job  $i$ :
  - $t_i$  is the processing time,  $d_i$  is the deadline.
  - Lateness  $l_i = f_i - d_i$  if finish time  $f_i > d_i$ ; 0 otherwise.

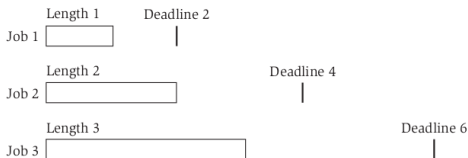
# SCHEDULING PROBLEM: MINIMIZE LATENESS



## Problem Definition

- $n$  jobs and a single machine that can process one job at a time
- For job  $i$ :
  - $t_i$  is the processing time,  $d_i$  is the deadline.
  - Lateness  $l_i = f_i - d_i$  if finish time  $f_i > d_i$ ; 0 otherwise.
- Objective: Build a schedule for all the jobs that minimizes the max lateness.

# SCHEDULING PROBLEM: MINIMIZE LATENESS



## Problem Definition

- $n$  jobs and a single machine that can process one job at a time
- For job  $i$ :
  - $t_i$  is the processing time,  $d_i$  is the deadline.
  - Lateness  $l_i = f_i - d_i$  if finish time  $f_i > d_i$ ; 0 otherwise.
- Objective: Build a schedule for all the jobs that minimizes the max lateness.

TopHat Discussion 2: What greedy heuristic might work?

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

**Heuristic 1: Increasing processing time.**

Schedule jobs by increasing  $t_i$ .

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

Heuristic 1: Increasing processing time.

Schedule jobs by increasing  $t_i$ .

Optimal?

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

Heuristic 1: Increasing processing time.

Schedule jobs by increasing  $t_i$ .

Optimal?

Counter-example: Jobs  $(t_i, d_i)$ :  $\{(1, 100), (10, 10)\}$

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

Heuristic 2: Increasing slack.

Schedule by increasing  $d_i - t_i$ .

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

Heuristic 2: Increasing slack.

Schedule by increasing  $d_i - t_i$ .

Optimal?



# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

Heuristic 2: Increasing slack.

Schedule by increasing  $d_i - t_i$ .

Optimal?

Counter-example:

Jobs  $(t_i, d_i)$ :  $\{(1, 2), (10, 10)\}$

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

**Heuristic 3: Earliest deadline first.**

Schedule by increasing  $d_i$ .

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

Heuristic 3: Earliest deadline first.

Schedule by increasing  $d_i$ .

Optimal?

# GREEDY ALGORITHMS FOR INTERVAL SCHEDULING

**Heuristic 3: Earliest deadline first.**

Schedule by increasing  $d_i$ .

**Optimal?**

Counter-example? Let's try and prove it.

# EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 3: EARLIEST DEADLINE FIRST.

## EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 3: EARLIEST DEADLINE FIRST.

---

**Algorithm:** EDF

---

Let  $J$  be the set of jobs.

Let  $S$  be an initially empty list.

**while**  $J$  is not empty **do**

    Choose  $j \in J$  with the smallest  $d_i$  (break ties arbitrarily).

    Append  $j$  to  $S$ .

**end**

**return**  $S$

---

## EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 3: EARLIEST DEADLINE FIRST.

---

**Algorithm:** EDF

---

Let  $J$  be the set of jobs.

Let  $S$  be an initially empty list.

**while**  $J$  is not empty **do**

    Choose  $j \in J$  with the smallest  $d_i$  (break ties arbitrarily).

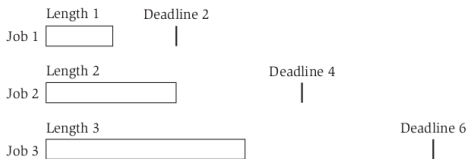
    Append  $j$  to  $S$ .

**end**

**return**  $S$

---

Sample Run (TopHat Q1: What is max lateness?)



# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*



# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ?

# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ? No.

# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ? No.
- Can there be multiple  $S^*$ ?

# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ? No.
- Can there be multiple  $S^*$ ? Yes.

# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ? No.
- Can there be multiple  $S^*$ ? Yes.
- We need to show either  $S = S^*$ , or  $S \equiv S^*$  for max lateness.

# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ? No.
- Can there be multiple  $S^*$ ? Yes.
- We need to show either  $S = S^*$ , or  $S \equiv S^*$  for max lateness.
- Technique: “Exchange Argument”

# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ? No.
- Can there be multiple  $S^*$ ? Yes.
- We need to show either  $S = S^*$ , or  $S \equiv S^*$  for max lateness.
- Technique: “Exchange Argument”
  - Start with an optimal solution  $S^*$  and transform it over a series of steps to something equivalent to  $S$  while maintaining optimality.

# ANALYSIS OF EDF

## Observation 4

*There is an optimal schedule with no idle time.*

## Showing Optimality

Let  $S^*$  be an optimal solution.

- Is it sufficient to show that  $|S| = |S^*|$ ? No.
- Can there be multiple  $S^*$ ? Yes.
- We need to show either  $S = S^*$ , or  $S \equiv S^*$  for max lateness.
- Technique: “Exchange Argument”
  - Start with an optimal solution  $S^*$  and transform it over a series of steps to something equivalent to  $S$  while maintaining optimality.
  - $S^* \equiv S_1 \equiv S_2 \equiv \dots \equiv S$  for max lateness.



# EXCHANGE ARGUMENT ANALYSIS

## Definition 10

A schedule  $A$  has an *inversion* if there are jobs  $i$  and  $j$  with  $i$  scheduled before  $j$  and  $d_j < d_i$ .

# EXCHANGE ARGUMENT ANALYSIS

## Definition 10

A schedule  $A$  has an *inversion* if there are jobs  $i$  and  $j$  with  $i$  scheduled before  $j$  and  $d_j < d_i$ .

## Lemma 11

*All schedules with no inversions and no idle time have the same lateness.*

# EXCHANGE ARGUMENT ANALYSIS

## Definition 10

A schedule  $A$  has an *inversion* if there are jobs  $i$  and  $j$  with  $i$  scheduled before  $j$  and  $d_j < d_i$ .

## Lemma 11

*All schedules with no inversions and no idle time have the same lateness.*

## Proof.



# EXCHANGE ARGUMENT ANALYSIS

## Definition 10

A schedule  $A$  has an *inversion* if there are jobs  $i$  and  $j$  with  $i$  scheduled before  $j$  and  $d_j < d_i$ .

## Lemma 11

*All schedules with no inversions and no idle time have the same lateness.*

## Proof.

- Only vary in jobs with the same deadline.
- Jobs with same deadline must be sequential.
- Ordering of jobs with same deadline won't change lateness.



# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

## Proof.



# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

## Proof.

- If  $S^*$  has an inversion, then there is a pair of jobs  $i$  and  $j$  such that  $j$  is scheduled immediately after  $i$  and has  $d_j < d_i$ .



# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

## Proof.

- If  $S^*$  has an inversion, then there is a pair of jobs  $i$  and  $j$  such that  $j$  is scheduled immediately after  $i$  and has  $d_j < d_i$ .
- We will swap  $i$  and  $j$  to create a new schedule  $S'$ . Note that  $S'$  has one less inversion than  $S^*$ .





# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

## Proof.

- If  $S^*$  has an inversion, then there is a pair of jobs  $i$  and  $j$  such that  $j$  is scheduled immediately after  $i$  and has  $d_j < d_i$ .
- We will swap  $i$  and  $j$  to create a new schedule  $S'$ . Note that  $S'$  has one less inversion than  $S^*$ .
- We need to show that  $S'$  has the same max lateness as  $S^*$ :



# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

### Proof.

- If  $S^*$  has an inversion, then there is a pair of jobs  $i$  and  $j$  such that  $j$  is scheduled immediately after  $i$  and has  $d_j < d_i$ .
- We will swap  $i$  and  $j$  to create a new schedule  $S'$ . Note that  $S'$  has one less inversion than  $S^*$ .
- We need to show that  $S'$  has the same max lateness as  $S^*$ :
  - Swapping  $i$  and  $j$  means that  $l'_j$  (lateness in  $S'$ ) is less than that in  $S^*$ .



# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

## Proof.

- If  $S^*$  has an inversion, then there is a pair of jobs  $i$  and  $j$  such that  $j$  is scheduled immediately after  $i$  and has  $d_j < d_i$ .
- We will swap  $i$  and  $j$  to create a new schedule  $S'$ . Note that  $S'$  has one less inversion than  $S^*$ .
- We need to show that  $S'$  has the same max lateness as  $S^*$ :
  - Swapping  $i$  and  $j$  means that  $l'_j$  (lateness in  $S'$ ) is less than that in  $S^*$ .
  - Lateness of  $i$  may increase, but:
$$l'_i = f'_i - d_i = f_j^* - d_i \leq f_j^* - d_j = l_j^*.$$



# ANALYSIS OF EDF

## Theorem 12

*There is an optimal schedule that has no inversions and no idle time.*

## Proof.

- If  $S^*$  has an inversion, then there is a pair of jobs  $i$  and  $j$  such that  $j$  is scheduled immediately after  $i$  and has  $d_j < d_i$ .
- We will swap  $i$  and  $j$  to create a new schedule  $S'$ . Note that  $S'$  has one less inversion than  $S^*$ .
- We need to show that  $S'$  has the same max lateness as  $S^*$ :
  - Swapping  $i$  and  $j$  means that  $l'_j$  (lateness in  $S'$ ) is less than that in  $S^*$ .
  - Lateness of  $i$  may increase, but:
$$l'_i = f'_i - d_i = f_j^* - d_i \leq f_j^* - d_j = l_j^*.$$
- Let  $S^* := S'$  and repeat until no more inversions.



# EDF IS OPTIMAL

## Corollary 13

*EDF produces an optimal schedule.*

## Proof.



# EDF IS OPTIMAL

## Corollary 13

*EDF produces an optimal schedule.*

## Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 12, there is an optimal schedule with no inversions and no idle time.
- Lemma 11 shows that these two schedules have the same max lateness.



# EDF IS OPTIMAL

## Corollary 13

*EDF produces an optimal schedule.*

## Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 12, there is an optimal schedule with no inversions and no idle time.
- Lemma 11 shows that these two schedules have the same max lateness.



Run time:

.

# EDF IS OPTIMAL

## Corollary 13

*EDF produces an optimal schedule.*

## Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 12, there is an optimal schedule with no inversions and no idle time.
- Lemma 11 shows that these two schedules have the same max lateness.

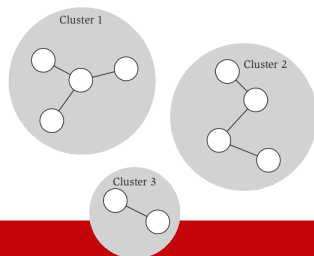


Run time: Sort the jobs by deadline:  $O(n \log n)$ .



# CLUSTERING\*

# $k$ -CLUSTERING



## Maximizing Spacing Problem

- A universe  $\mathcal{U} := \{p_1, \dots, p_n\}$  of  $n$  objects.
- Distance function  $d : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$  such that, for all  $p_i, p_j \in \mathcal{U}$ :
  - $d(p_i, p_i) = 0$
  - $d(p_i, p_j) > 0$
  - $d(p_i, p_j) = d(p_j, p_i)$
- Objective: Partition  $\mathcal{U}$  into  $k$  non-empty groups  $\mathcal{C} := C_1, \dots, C_k$  with maximum spacing:

$$\text{maximize } \min_{C_i, C_j \in \mathcal{C}} \min_{u \in C_i, v \in C_j} d(u, v)$$

# ALGORITHM DESIGN

TopHat Discussion 4: What greedy approach might work?

# ALGORITHM DESIGN

## Algorithm

- Build an MST.
- Remove  $k - 1$  largest edges.

# ALGORITHM DESIGN

## Algorithm

- Build an MST.
- Remove  $k - 1$  largest edges.

## $k$ -Clusters at max spacing?

- Start with a tree, remove  $k - 1$  edges: We get a forest of  $k$  trees.
- By definition largest edges are removed so max spacing.

# ALGORITHM DESIGN

## Algorithm

- Build an MST.
- Remove  $k - 1$  largest edges.

## $k$ -Clusters at max spacing?

- Start with a tree, remove  $k - 1$  edges: We get a forest of  $k$  trees.
- By definition largest edges are removed so max spacing.

## TopHat Q10: Which MST algorithm?

# ALGORITHM DESIGN

## Algorithm

- Build an MST.
- Remove  $k - 1$  largest edges.

## $k$ -Clusters at max spacing?

- Start with a tree, remove  $k - 1$  edges: We get a forest of  $k$  trees.
- By definition largest edges are removed so max spacing.

## TopHat Q10: Which MST algorithm?

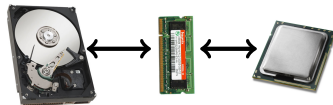
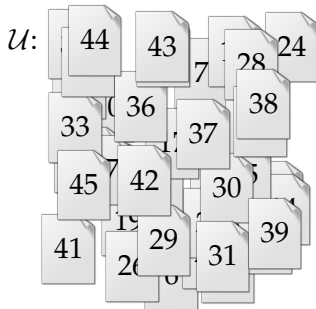
Kruskal's ( $O(m \log n)$  which is  $O(n^2 \log n)$  for clustering):

- Merge sets from lowest to most expensive edges.
- Stop when we have  $k$  sets.

# PAGING\*



# PAGING PROBLEM



Cache:

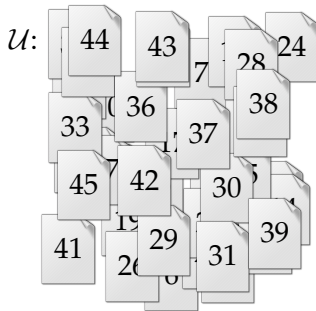


Requests:

## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

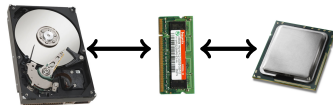
# PAGING PROBLEM



Cache:



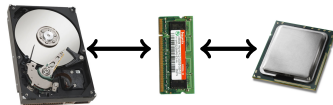
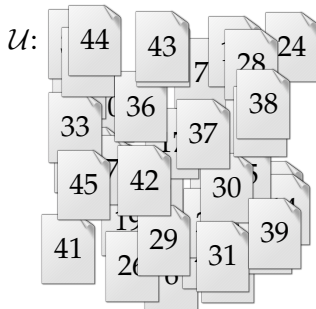
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



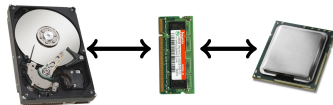
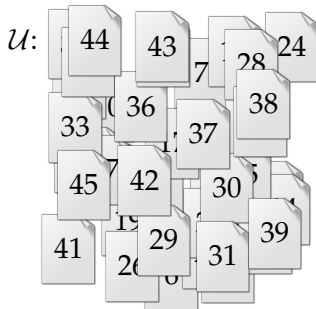
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



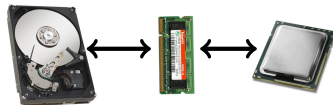
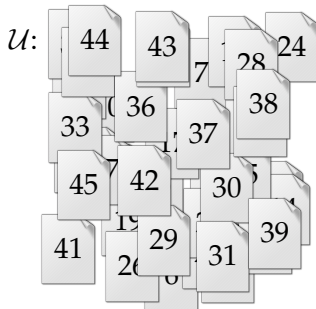
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



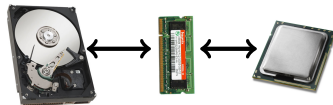
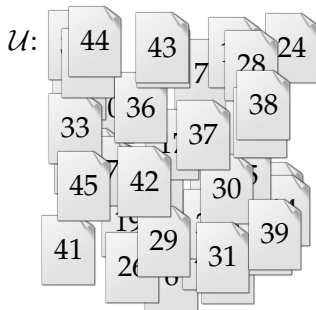
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



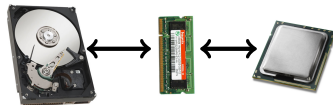
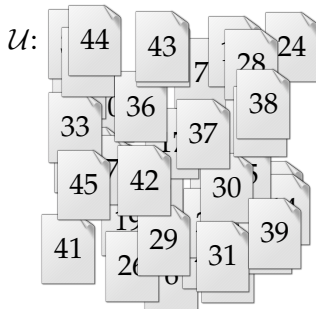
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



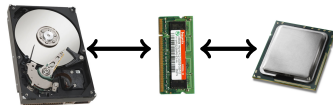
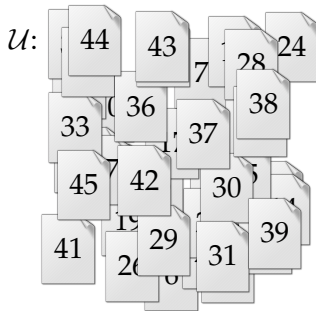
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



Requests:

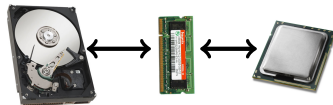
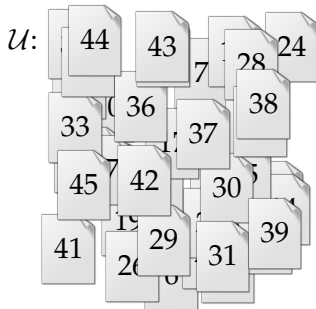


## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).



# PAGING PROBLEM



Cache:



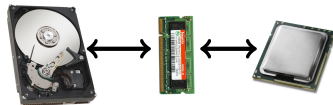
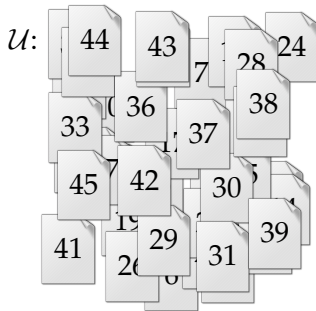
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



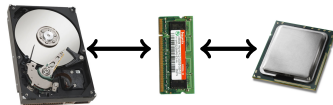
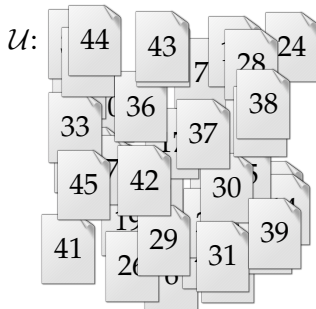
Requests:



## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



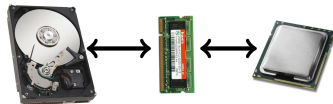
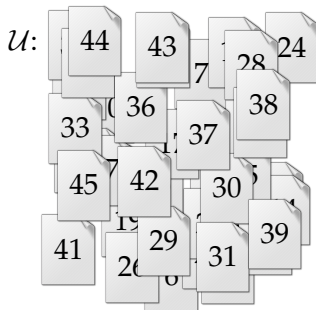
Requests:



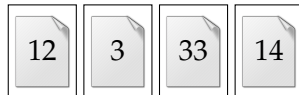
## Definition

- $\mathcal{U}$ : universe of pages ( $|\mathcal{U}| > k$ ).
- Cache of size  $k$ .
- Requests are to the pages of  $\mathcal{U}$ .
- Goal: Minimize the number of page faults (requests to pages not in the cache).

# PAGING PROBLEM



Cache:



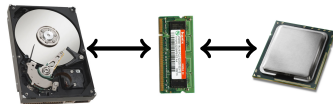
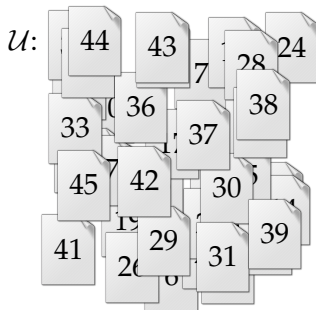
Requests:



## Eviction Strategies

- When designing an algorithm, we are picking an eviction strategy.

# PAGING PROBLEM



Cache:



Requests:



## Eviction Strategies

- When designing an algorithm, we are picking an eviction strategy.
- In the offline version, the algorithm knows the request sequence. What might be a good eviction strategy?

# OFFLINE GREEDY ALGORITHM

## Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

# OFFLINE GREEDY ALGORITHM

## Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

Small Run:

- $\mathcal{U} = \{a, b, c\}$
- $k = 2$
- $\sigma = \langle a, b, c, b, c, a, b \rangle$

# OFFLINE GREEDY ALGORITHM

## Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

Small Run:

- $\mathcal{U} = \{a, b, c\}$
- $k = 2$
- $\sigma = \langle a, b, c, b, c, a, b \rangle$
- TopHat 6: How many faults in small run?



# OFFLINE GREEDY ALGORITHM

## Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

Small Run:

- $\mathcal{U} = \{a, b, c\}$
- $k = 2$
- $\sigma = \langle a, b, c, b, c, a, b \rangle$
- TopHat 6: How many faults in small run?

TopHat 7: Which strategy to prove optimality?

# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{FF}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{FF}$  for the first  $j + 1$  items with no more faults than  $S$ .*

**Proof.**



# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

### Proof.

- If on request  $j + 1$ ,  $S$  behaves as  $S_{\text{FF}}$ . Then define  $S'$  as  $S$  and the claim follows.



# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

### Proof.

- If on request  $j + 1$ ,  $S$  behaves as  $S_{\text{FF}}$ . Then define  $S'$  as  $S$  and the claim follows.
- Otherwise, say  $S$  evicts  $u$  and  $S_{\text{FF}}$  evicts  $v$ . We will build  $S'$  by following  $S_{\text{FF}}$  for the first  $j + 1$  requests. Note that the number of faults are the same for  $S$  and  $S'$  up to  $j + 1$ , and the caches match except for  $u$  and  $v$ .



# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

### Proof.

- From  $j + 2$  onward,  $S'$  follows  $S$  until either:



# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

### Proof.

- From  $j + 2$  onward,  $S'$  follows  $S$  until either:
  - ①  $S$  evicts  $v$ . In this case,  $S'$  evicts  $u$ .



# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

### Proof.

- From  $j + 2$  onward,  $S'$  follows  $S$  until either:
  - ①  $S$  evicts  $v$ . In this case,  $S'$  evicts  $u$ .
  - ②  $S$  evicts  $g \neq v$  to bring  $u$  into the cache. In this case,  $S'$  evicts  $g$  and brings in  $v$ .





# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{FF}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{FF}$  for the first  $j + 1$  items with no more faults than  $S$ .*

### Proof.

- From  $j + 2$  onward,  $S'$  follows  $S$  until either:
  - ①  $S$  evicts  $v$ . In this case,  $S'$  evicts  $u$ .
  - ②  $S$  evicts  $g \neq v$  to bring  $u$  into the cache. In this case,  $S'$  evicts  $g$  and brings in  $v$ .
- In either case, both  $S$  and  $S'$  have a page fault, and afterwards their cache match.



# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

How do we get optimality of  $S_{\text{FF}}$  from Theorem 14?

# PROVING FF OPTIMALITY

## EXCHANGE ARGUMENT

### Theorem 14

*Let  $S$  be a schedule for the  $n$  request that make the same eviction decisions as  $S_{\text{FF}}$  for the first  $j$  items. Then, there is a schedule  $S'$  that makes the same eviction requests as  $S_{\text{FF}}$  for the first  $j + 1$  items with no more faults than  $S$ .*

### How do we get optimality of $S_{\text{FF}}$ from Theorem 14?

By induction: We begin with the optimal schedule  $S^*$  and inductively apply Theorem 14 for  $j = 1, 2, 3, \dots, n$ , which after the  $n$  iterations, produces  $S_{\text{FF}}$ .

# PREFIX CODES\*

# BINARY ENCODING

## Fixed-Width Encoding

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^k$ .  
 $\gamma(S) := \{000, 001, 010, 011, 100\}$ .
- Ex. ASCII

# BINARY ENCODING

## Fixed-Width Encoding

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^k$ .  
 $\gamma(S) := \{000, 001, 010, 011, 100\}$ .
- Ex. ASCII
- TopHat Q11: Decode 000010.

# BINARY ENCODING

## Fixed-Width Encoding

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^k$ .  
 $\gamma(S) := \{000, 001, 010, 011, 100\}$ .
- Ex. ASCII
- TopHat Q11: Decode 000010.

## Variable-Width Encoding

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{0, 1, 10, 01, 11\}$ .

# BINARY ENCODING

## Fixed-Width Encoding

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^k$ .  
 $\gamma(S) := \{000, 001, 010, 011, 100\}$ .
- Ex. ASCII
- TopHat Q11: Decode 000010.

## Variable-Width Encoding

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{0, 1, 10, 01, 11\}$ .
- TopHat Q12: How many ways to decode 0010?



# UNIQUE VARIABLE-WIDTH ENCODINGS

## Prefix Codes

Encoding of  $S$  such that no encoding of a symbol in  $S$  is a prefix of another.

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{11, 01, 001, 000, 100\}$ .

# UNIQUE VARIABLE-WIDTH ENCODINGS

## Prefix Codes

Encoding of  $S$  such that no encoding of a symbol in  $S$  is a prefix of another.

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{11, 01, 001, 000, 100\}$ .
- 0010 invalid sequence

# UNIQUE VARIABLE-WIDTH ENCODINGS

## Prefix Codes

Encoding of  $S$  such that no encoding of a symbol in  $S$  is a prefix of another.

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{11, 01, 001, 000, 100\}$ .
- 0010 invalid sequence
- TopHat 13: Decode 1101.

# UNIQUE VARIABLE-WIDTH ENCODINGS

## Prefix Codes

Encoding of  $S$  such that no encoding of a symbol in  $S$  is a prefix of another.

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{11, 01, 001, 000, 100\}$ .

## Easy Decoding

Scan left to right, once an encoding is matched, output symbol.

# UNIQUE VARIABLE-WIDTH ENCODINGS

## Prefix Codes

Encoding of  $S$  such that no encoding of a symbol in  $S$  is a prefix of another.

- Set of symbols  $S := \{a, b, c, d, e\}$ .
- Encoding function  $\gamma : S \rightarrow \{0, 1\}^*$ .  
 $\gamma(S) := \{11, 01, 001, 000, 100\}$ .

## Easy Decoding

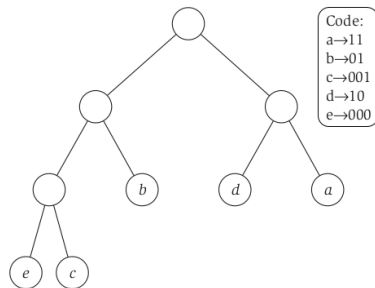
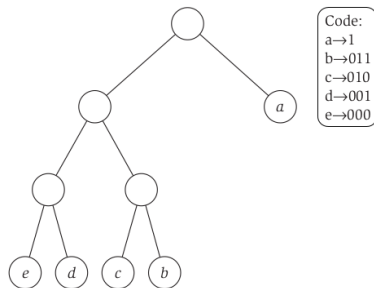
Scan left to right, once an encoding is matched, output symbol.

## Optimal Prefix Codes

- For a set of symbols  $S$ , let  $f_x$  denote the frequency of  $x$  in the text to be encoded.
- Average bits  $\text{ABL}(\gamma) := \sum_{x \in S} f_x \cdot |\gamma(x)|$ .
- Goal: Find  $\gamma$  that minimizes  $\text{ABL}$ .

# ALGORITHM DESIGN

## PREFIX BINARY TREES



# OPTIMAL PREFIX TREE IS FULL

## Theorem 15

*The binary tree corresponding to the optimal prefix code is full.*

# OPTIMAL PREFIX TREE IS FULL

## Theorem 15

*The binary tree corresponding to the optimal prefix code is full.*

## Proof.



# OPTIMAL PREFIX TREE IS FULL

## Theorem 15

*The binary tree corresponding to the optimal prefix code is full.*

## Proof.

By exchange argument:

- Let  $T$  be an optimal prefix tree with a node  $u$  with one child  $v$ .



# OPTIMAL PREFIX TREE IS FULL

## Theorem 15

*The binary tree corresponding to the optimal prefix code is full.*

## Proof.

By exchange argument:

- Let  $T$  be an optimal prefix tree with a node  $u$  with one child  $v$ .
- Let  $T'$  be  $T$  with  $u$  replaced with  $v$ .



# OPTIMAL PREFIX TREE IS FULL

## Theorem 15

*The binary tree corresponding to the optimal prefix code is full.*

## Proof.

By exchange argument:

- Let  $T$  be an optimal prefix tree with a node  $u$  with one child  $v$ .
- Let  $T'$  be  $T$  with  $u$  replaced with  $v$ .
- Distance to  $v$  decreases by 1 in  $T'$ , a contradiction.



# TOP-DOWN APPROACH

## Algorithm

- Split  $S$  into two sets such that the sets frequency are  $1/2$  the total frequency.
- Recurse on new sets until singletons.

# TOP-DOWN APPROACH

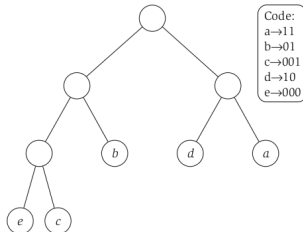
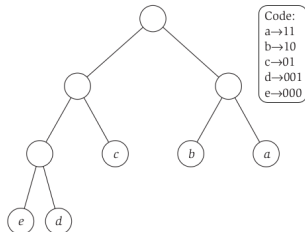
## Algorithm

- Split  $S$  into two sets such that the sets frequency are  $1/2$  the total frequency.
- Recurse on new sets until singletons.

$$f_a = .32, f_b = .25, f_c = .2, f_d = .18, f_e = .05$$

$$\text{ABL}(\text{OPT}) = 2.23$$

$$\text{ABL}(\text{TopDown}) = 2.25$$



# WHAT IF WE KNEW THE OPTIMAL TREE?

Let  $T^*$  be the optimal (unlabelled) prefix tree.

## WHAT IF WE KNEW THE OPTIMAL TREE?

Let  $T^*$  be the optimal (unlabelled) prefix tree.

### Lemma 16

*Let  $u, v$  be leaves of  $T^*$  such that  $\text{depth}(u) < \text{depth}(v)$ , where  $u$  is labelled with  $y$  and  $v$  is labelled with  $z$ . Then,  $f_y \geq f_z$ .*

## WHAT IF WE KNEW THE OPTIMAL TREE?

Let  $T^*$  be the optimal (unlabelled) prefix tree.

### Lemma 16

*Let  $u, v$  be leaves of  $T^*$  such that  $\text{depth}(u) < \text{depth}(v)$ , where  $u$  is labelled with  $y$  and  $v$  is labelled with  $z$ . Then,  $f_y \geq f_z$ .*

### Proof.



## WHAT IF WE KNEW THE OPTIMAL TREE?

Let  $T^*$  be the optimal (unlabelled) prefix tree.

### Lemma 16

*Let  $u, v$  be leaves of  $T^*$  such that  $\text{depth}(u) < \text{depth}(v)$ , where  $u$  is labelled with  $y$  and  $v$  is labelled with  $z$ . Then,  $f_y \geq f_z$ .*

### Proof.

If  $f_y < f_z$ , exchange the labelling of  $y$  and  $z$ . Since  $\text{depth}(u) < \text{depth}(v)$ ,  $\text{ABL}(T^*)$  must decrease with the new labelling. □

## WHAT IF WE KNEW THE OPTIMAL TREE?

Let  $T^*$  be the optimal (unlabelled) prefix tree.

### Lemma 16

*Let  $u, v$  be leaves of  $T^*$  such that  $\text{depth}(u) < \text{depth}(v)$ , where  $u$  is labelled with  $y$  and  $v$  is labelled with  $z$ . Then,  $f_y \geq f_z$ .*

### Labelling $T^*$

## WHAT IF WE KNEW THE OPTIMAL TREE?

Let  $T^*$  be the optimal (unlabelled) prefix tree.

### Lemma 16

*Let  $u, v$  be leaves of  $T^*$  such that  $\text{depth}(u) < \text{depth}(v)$ , where  $u$  is labelled with  $y$  and  $v$  is labelled with  $z$ . Then,  $f_y \geq f_z$ .*

### Labelling $T^*$

- Order symbols by increasing frequency.
- Assign them to leaves of  $T^*$  by decreasing depth.

## WHAT IF WE KNEW THE OPTIMAL TREE?

Let  $T^*$  be the optimal (unlabelled) prefix tree.

### Lemma 16

*Let  $u, v$  be leaves of  $T^*$  such that  $\text{depth}(u) < \text{depth}(v)$ , where  $u$  is labelled with  $y$  and  $v$  is labelled with  $z$ . Then,  $f_y \geq f_z$ .*

### Labelling $T^*$

- Order symbols by increasing frequency.
- Assign them to leaves of  $T^*$  by decreasing depth.

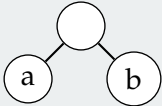
### Observation 5

*In  $T^*$ , the lowest frequency letters are siblings.*

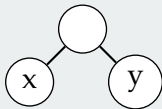
# BOTTOM-UP APPROACH

## HUFFMAN CODE

### Huffman's Algorithm

- If  $|S| = 2$ , return 
- Let  $x$  and  $y$  be the lowest frequency symbols.
- Set  $S := S \setminus \{x, y\} \cup \{xy\}$  and  $f_{xy} = f_x + f_y$ .
- $T :=$  recurse on  $S$ .

- Replace  with
- return  $T$



# HUFFMAN CODES ARE OPTIMAL

## Lemma 17

*Let  $T'$  be the tree at the  $(k - 1)$ -st step, and let  $T$  be the tree at the  $k$ -th step.  $ABL(T') = ABL(T) - f_w$ , where  $w$  is the symbol replaced in the  $k$ -th step by  $y$  and  $z$ .*

# HUFFMAN CODES ARE OPTIMAL

## Lemma 17

*Let  $T'$  be the tree at the  $(k - 1)$ -st step, and let  $T$  be the tree at the  $k$ -th step.  $ABL(T') = ABL(T) - f_w$ , where  $w$  is the symbol replaced in the  $k$ -th step by  $y$  and  $z$ .*

## Proof.

$$\begin{aligned} ABL(T) &= \sum_{x \in S} f_x \cdot \text{depth}(x) \\ &= f_y \cdot \text{depth}(y) + f_z \cdot \text{depth}(z) + \sum_{x \in S; x \notin \{y, z\}} f_x \cdot \text{depth}(x) \\ &= f_w + f_w \cdot \text{depth}(w) + \sum_{x \in S \setminus \{y, z\}} f_x \cdot \text{depth}(x) \\ &= f_w + ABL(T') \end{aligned}$$

# HUFFMAN CODES ARE OPTIMAL

## Lemma 17

*Let  $T'$  be the tree at the  $(k - 1)$ -st step, and let  $T$  be the tree at the  $k$ -th step.  $ABL(T') = ABL(T) - f_w$ , where  $w$  is the symbol replaced in the  $k$ -th step by  $y$  and  $z$ .*

## Theorem 18

*Huffman Algorithm is optimal.*

## Proof.





# HUFFMAN CODES ARE OPTIMAL

## Lemma 17

*Let  $T'$  be the tree at the  $(k - 1)$ -st step, and let  $T$  be the tree at the  $k$ -th step.  $ABL(T') = ABL(T) - f_w$ , where  $w$  is the symbol replaced in the  $k$ -th step by  $y$  and  $z$ .*

## Theorem 18

*Huffman Algorithm is optimal.*

## Proof.

By induction:



# HUFFMAN CODES ARE OPTIMAL

## Lemma 17

*Let  $T'$  be the tree at the  $(k - 1)$ -st step, and let  $T$  be the tree at the  $k$ -th step.  $ABL(T') = ABL(T) - f_w$ , where  $w$  is the symbol replaced in the  $k$ -th step by  $y$  and  $z$ .*

## Theorem 18

*Huffman Algorithm is optimal.*

## Proof.

By induction:

- Base case  $|S| = 2$



# HUFFMAN CODES ARE OPTIMAL

## Lemma 17

*Let  $T'$  be the tree at the  $(k - 1)$ -st step, and let  $T$  be the tree at the  $k$ -th step.  $ABL(T') = ABL(T) - f_w$ , where  $w$  is the symbol replaced in the  $k$ -th step by  $y$  and  $z$ .*

## Theorem 18

*Huffman Algorithm is optimal.*

## Proof.

By induction:

- Base case  $|S| = 2$
- Inductive step: We have  $T$ . By way of contradiction, assume  $ABL(Z) \leq ABL(T)$ .



# HUFFMAN CODES ARE OPTIMAL

## Lemma 17

*Let  $T'$  be the tree at the  $(k - 1)$ -st step, and let  $T$  be the tree at the  $k$ -th step.  $ABL(T') = ABL(T) - f_w$ , where  $w$  is the symbol replaced in the  $k$ -th step by  $y$  and  $z$ .*

## Theorem 18

*Huffman Algorithm is optimal.*

## Proof.

By induction:

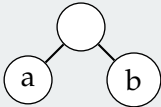
- We observed that  $y$  and  $z$  are siblings. Hence:

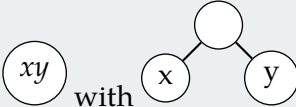

$$\begin{aligned} & ABL(Z) < ABL(T) \\ \iff & ABL(Z') + f_w < ABL(T') + f_w, \text{ by Lemma 17} \\ \iff & ABL(Z') < ABL(T'), \text{ a contradiction.} \end{aligned}$$

# BOTTOM-UP APPROACH

## HUFFMAN CODE

### Huffman's Algorithm

- If  $|S| = 2$ , return 
- Let  $x$  and  $y$  be the lowest frequency symbols.
- Set  $S := S \setminus \{x, y\} \cup \{xy\}$  and  $f_{xy} = f_x + f_y$ .
- $T :=$  recurse on  $S$ .

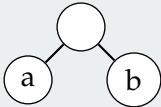
- Replace  with 
- return  $T$


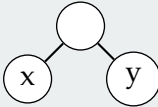
Runtime:

# BOTTOM-UP APPROACH

## HUFFMAN CODE

### Huffman's Algorithm

- If  $|S| = 2$ , return 
- Let  $x$  and  $y$  be the lowest frequency symbols.
- Set  $S := S \setminus \{x, y\} \cup \{xy\}$  and  $f_{xy} = f_x + f_y$ .
- $T :=$  recurse on  $S$ .

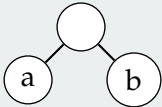
- Replace  with 
- return  $T$

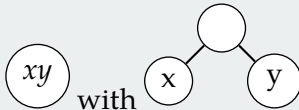

Runtime:  $|S| - 1$  recursions with find min over  $|S_i|$  elements

# BOTTOM-UP APPROACH

## HUFFMAN CODE

### Huffman's Algorithm

- If  $|S| = 2$ , return 
- Let  $x$  and  $y$  be the lowest frequency symbols.
- Set  $S := S \setminus \{x, y\} \cup \{xy\}$  and  $f_{xy} = f_x + f_y$ .
- $T :=$  recurse on  $S$ .

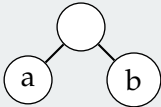
- Replace  with 
- return  $T$


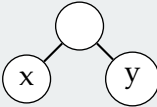
Runtime:  $O(|S|^2)$

# BOTTOM-UP APPROACH

## HUFFMAN CODE

### Huffman's Algorithm

- If  $|S| = 2$ , return 
- Let  $x$  and  $y$  be the lowest frequency symbols.
- Set  $S := S \setminus \{x, y\} \cup \{xy\}$  and  $f_{xy} = f_x + f_y$ .
- $T :=$  recurse on  $S$ .

- Replace  with 
- return  $T$

Runtime:  $O(|S|^2)$

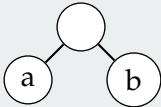
what about  $O(|S| \log |S|)$ ?


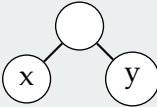


# BOTTOM-UP APPROACH

## HUFFMAN CODE

### Huffman's Algorithm

- If  $|S| = 2$ , return 
- Let  $x$  and  $y$  be the lowest frequency symbols.
- Set  $S := S \setminus \{x, y\} \cup \{xy\}$  and  $f_{xy} = f_x + f_y$ .
- $T :=$  recurse on  $S$ .

- Replace  with 
- return  $T$

Runtime:  $O(|S|^2)$

what about  $O(|S| \log |S|)$ ? Priority Queue (min-heap)

# APPENDIX

# REFERENCES

# IMAGE SOURCES I



<https://www.cse.unsw.edu.au/~cs1521/17s2/lecs/notices/slide068.html>



<http://mediablogrueil.blogspot.fr/2012/11/one-page-design-effet-de-mode-ou-reel.html>



<http://www.culturizame.es/articulo/nuestro-pequeno-diccionario-de-tecnologia>



<http://computer-help-tips.blogspot.fr/2011/04/different-types-of-computer-processors.html>

# IMAGE SOURCES II



**WISCONSIN**  
UNIVERSITY OF WISCONSIN-MADISON

<https://brand.wisc.edu/web/logos/>