# CS 577 - Greedy

Marc Renault

Department of Computer Sciences
University of Wisconsin – Madison

Summer 2023
TopHat Section 001 Join Code: 275653

# Greedy

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

Non-optimal example:

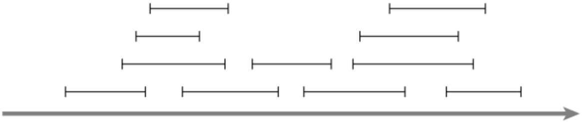
- $\sigma = \langle 1/2 - \varepsilon, 1/2 - \varepsilon, 1/2 + \varepsilon, 1/2 + \varepsilon \rangle$
- FFI: 3 bins
- OPT: 2 bins

Techniques for showing that GREEDY is optimal:

- Always stays ahead
- Exchange argument

# Stays Ahead: Interval Scheduling

# Interval Scheduling



## Problem Definition

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

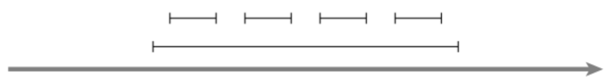
# Greedy Algorithms for Interval Scheduling

## Heuristic 1: Earliest First

Schedule a compatible request with the earliest start time.

## Optimal?

Counter-example:



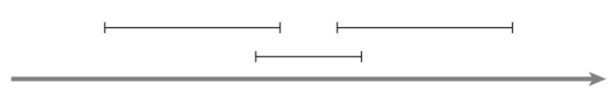
# Greedy Algorithms for Interval Scheduling

## Heuristic 2: Smallest Interval

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

## Optimal?

Counter-example:



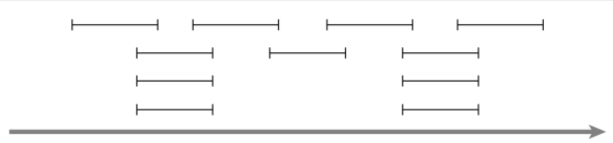
# Greedy Algorithms for Interval Scheduling

## 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.

**Optimal?**

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

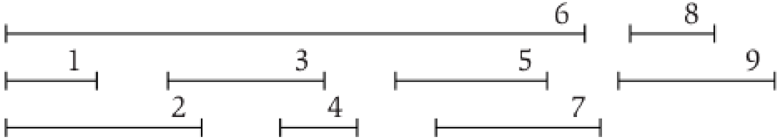
# Exercise: Formalize the algorithm (pseudocode)

## 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 1

*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 $i$, $|S_i| \geq |S_i^*|$.

# Stays Ahead Analysis

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

## Lemma 1

*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 $f_q > f_{j_r}$, but this is a contradiction.

$\square$

# Stays Ahead Analysis

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

**Lemma 1**

*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 1.

# 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 2**

*FinishFirst produces an optimal schedule.*

**Proof.**

By way of contradiction, assume that $|S^*| > |S|$. This implies that $m > k$. Lemma 1 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. $\square$

# 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.

Overall:

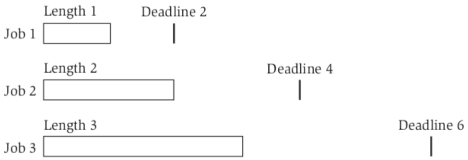
$O(n \log n) + O(n) = O(n \log n)$

# Interval Extensions

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

### 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 Minimizing Max Lateness

**Heuristic 2: Increasing slack.**

Schedule by increasing $d_i - t_i$.

**Optimal?**

Counter-example:

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

# Greedy Algorithms for Minimizing Max Lateness

**Heuristic 3: Earliest deadline first.**

Schedule by increasing $d_i$.

**Optimal?**

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

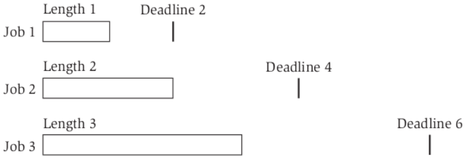
# Exercise: Formalize the algorithm (pseudocode)

## Heuristic 3: Earliest deadline first.

**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 2

*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.
  - $S^* \equiv S_1 \equiv S_2 \equiv \cdots \equiv S$ for max lateness.

# Exchange Argument Analysis

## Definition 3

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

## Lemma 4

*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 sequential.
- Ordering of jobs with same deadline won't change lateness.

□

# Analysis of EDF

## Theorem 5

*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.

$\square$

## EDF is Optimal

### Corollary 6

*EDF produces an optimal schedule.*

### Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 5, there is an optimal schedule with no inversions and no idle time.
- Lemma 4 shows that these two schedules have the same max lateness.

□

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

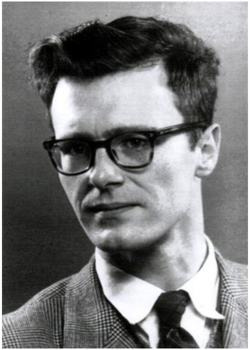
# Shortest Path

# 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?



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**

How is it greedy?
TopHat 3: Which technique to prove optimality?

# Correctness of Dijkstra's

**Theorem 7**

*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 7**

*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?
- 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**

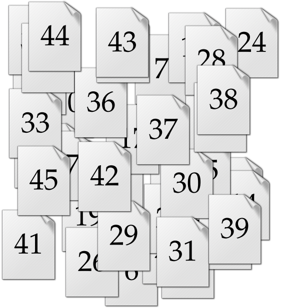
---

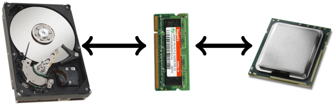
- TopHat 5: Number of iterations of the loop?
  $n - 1$
- Key Operations:
  - Finding the min: Easy in $O(m)$
- Overall: $O(mn)$
- How can we get $O(m \log n)$?

# Paging

# Paging Problem

$\mathcal{U}$:



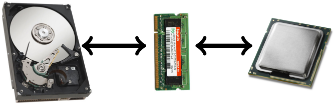
Cache: 12 3 33 14

Requests: 12 3 33 14 12 20

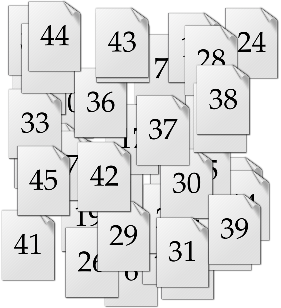
## Definition

- $\mathcal{U}$: 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



$\mathcal{U}$:




Cache: 12 | 3 | 33 | 14

Requests: 12, 3, 33, 14, 12, 20

## 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.

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 8

*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 8

*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 $v$.
  - Note: Since $S_{FF}$ evicts $v$ at $j+1$, $u$ must be requested before $v$.
- In either case, both $S$ and $S'$ have a page fault, and afterwards their cache match. $\square$

# Proving FF Optimality

## Exchange Argument

### Theorem 8

*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 8?

By induction: We begin with the optimal schedule $S^*$ and inductively apply Theorem 8 for $j = 1, 2, 3, \ldots, n$, which after the $n$ iterations, produces $S_{\text{FF}}$.

# 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 $\sum_{e \in F} c_e$.

## Observation 3

*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 $T$.

$\square$

# 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 9**

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

**Observation 4**

*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 10

*Let $S \subset V$ be an 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 $T$.

$\square$

# 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 11

*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 10.

□

# 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 12

*Prim's Algorithm produces an MST.*

### Proof.

- Immediate from Lemma 10.
- That is, Prim's algorithm does exactly what Lemma 10 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 13

*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 $T$.

$\square$

# Reverse-Delete is Optimal

**Lemma 13**

*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 14**

*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 13.

□

# 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

- 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

- `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.

| | | |
|---|---|---|
| s | 0 | |

# Union-Find Operations

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

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

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

- (WLOG) $x$`.rank` $\geq y$`.rank`:
  $y$`.parent` $= x$
- If $x$`.rank` $= y$`.rank`:
  $x$`.rank` $:= x$`.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.

## Dijktra's

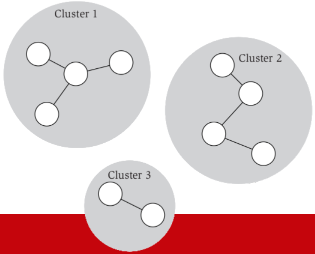
- Traverses a graph starting from some node $s$.
- Builds a tree $T$.
- All $s$ to $u$ paths in $T$ are the shortest such path in $G$.

## MST Algorithms

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

# Clustering

# $k$-Clustering




## Maximizing Spacing Problem

- A universe $\mathcal{U} := \{p_1, \ldots, p_n\}$ of $n$ objects.
- Distance function $d : \mathcal{U} \times \mathcal{U} \to \mathbb{R}$ 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 $\mathcal{C} := C_1, \ldots, C_k$ 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.

## $k$-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.

# Prefix Codes

# Binary Encoding

## Fixed-Width Encoding

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

## Variable-Width Encoding

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0,1\}^*$.
  $\gamma(S) := \{0, 1, 10, 01, 11\}$.
- 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 $\gamma : S \to \{0, 1\}^*$.
  $\gamma(S) := \{11, 01, 001, 000, 100\}$.
- 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 $\gamma : S \to \{0, 1\}^*$.
  $\gamma(S) := \{11, 01, 001, 000, 100\}$.

## Easy Decoding

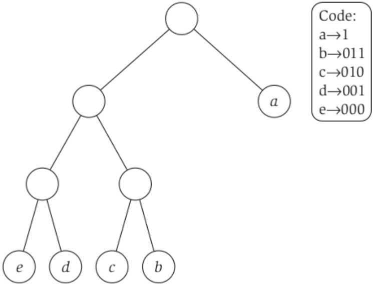
Scan left to right, once an encoding is matched, output symbol.

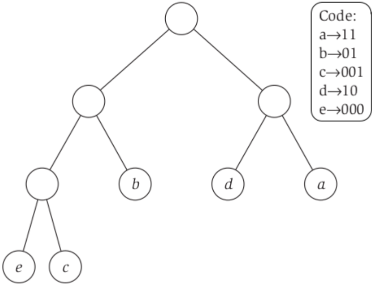
## Optimal Prefix Codes

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

# Algorithm Design

## Prefix Binary Trees



# 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.

$\square$

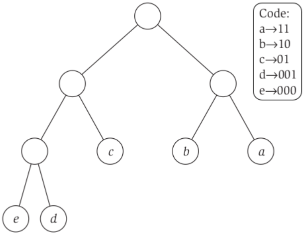
# TOP-DOWN APPROACH

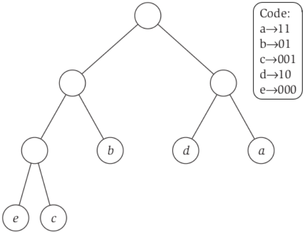
## Algorithm

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

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

$\textsc{abl}(\textsc{opt}) = 2.23$ $\textsc{abl}(\text{TopDown}) = 2.25$



# 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 $depth(u) < depth(v)$, 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 $\text{depth}(u) < \text{depth}(v)$, $\textsc{abl}(T^*)$ must decrease with the new labelling. $\square$

# 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 $depth(u) < depth(v)$, 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.

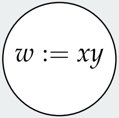
## Observation 5

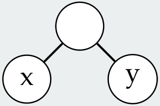
*In $T^*$, the lowest frequency letters are siblings.*

# Bottom-Up Approach

## Huffman Code

### Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
  - Let $x$ and $y$ be the lowest frequency symbols.
  - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
  - Repeat until $|S| = 1$.
- (2) Generate the tree:
  - $T :=$ root with element from $S$.
  - Replace 

with 

  - Repeat until leaves of $T$ are original symbols.

# 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. $\textsc{abl}(T') = \textsc{abl}(T) - f_w$, where $w$ is the symbol replaced in the $k$-th step by $y$ and $z$.*

## Proof.

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

□

# 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. $\textsc{abl}(T') = \textsc{abl}(T) - f_w$, 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 $\textsc{abl}(Z) \leq \textsc{abl}(T)$.

# 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. $\text{ABL}(T') = \text{ABL}(T) - f_w$, 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:

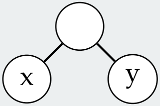
$$\begin{aligned} & \text{ABL}(Z) < \text{ABL}(T) \\ \iff \; & \text{ABL}(Z') + f_w < \text{ABL}(T') + f_w, \text{ by Lemma 17} \\ \iff \; & \text{ABL}(Z') < \text{ABL}(T'), \text{ a contradiction.} \end{aligned}$$

$\square$

# Bottom-Up Approach

## Huffman Code

### Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
  - Let $x$ and $y$ be the lowest frequency symbols.
  - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
  - Repeat until $|S| = 1$.
- (2) Generate the tree:
  - $T :=$ root with element from $S$.
  - Replace $w := xy$ with 

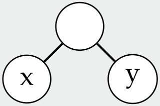
  - Repeat until leaves of $T$ are original symbols.

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

# Bottom-Up Approach

## Huffman Code

### Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
  - Let $x$ and $y$ be the lowest frequency symbols.
  - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
  - Repeat until $|S| = 1$.
- (2) Generate the tree:
  - $T :=$ root with element from $S$.
  - Replace $w := xy$ with 

  - Repeat until leaves of $T$ are original symbols.

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

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