Greedy Algorithms

Textbook reading

Chapter 4
Chapter 5
Overview

Design principle:
- Make progress towards a solution based on local criteria

Proof techniques:
- Induction
- “Stay ahead” arguments
- Exchange arguments

Problems:
- Interval scheduling
- Minimum spanning trees
- Shortest paths
- Minimum-length codes

Basic data structures:
- Binary heaps
- Union-find structure
Problem 1: Interval Scheduling

Given: Set of activities competing for time intervals on a given resource

Goal: Schedule as many non-conflicting activities as possible
Problem 1: Interval Scheduling

Given: Set of activities competing for time intervals on a given resource

Goal: Schedule as many non-conflicting activities as possible
**Problem 1: Interval Scheduling**

**Given:** Set of activities competing for time intervals on a given resource

**Goal:** Schedule as many non-conflicting activities as possible
Problem 2: Minimum Spanning Tree (1)

Given \( n \) computers, we want to connect them so that every pair of them can communicate with each other.

- We don’t care whether the connection between every pair of computers is short.
- We don’t care about fault tolerance.
- Every foot of cable costs us $1.

We want the cheapest possible network.
Problem 2: Minimum Spanning Tree (2)

Given a graph $G = (V, E)$ and an assignment of weights (costs) to the edges of $G$, a minimum spanning tree (MST) $T$ of $G$ is a spanning tree with minimum total weight

$$w(T) = \sum_{e \in T} w(e).$$
Problem 3: Single-Source Shortest Paths

**Given:**
- A graph \( G \),
- An assignment of weights to the edges of \( G \), and
- A *source vertex* \( s \).

**Goal:**
Compute the distance \( \text{dist}(s, v) \) from \( s \) to \( v \), for all \( v \in G \).

**Assumption:**
All edge weights are non-negative.
**Problem 4: Minimum-Length Codes**

**The problem:**

Given a text $T$ containing characters $c_1, c_2, \ldots, c_n$, each with frequency $f(c_i)$, find a binary code that encodes $T$ in the minimum number of bits.

**Motivation:**

- We want to transmit the text over a slow network connection.
- We want to store the text on disk, while minimizing the disk space usage.
- ...
Problem 4: Minimum-Length Codes

The problem:

Given a text $\mathcal{T}$ containing characters $c_1, c_2, \ldots, c_n$, each with frequency $f(c_i)$, find a binary code that encodes $\mathcal{T}$ in the minimum number of bits.
In an optimization problem, there are many possible solutions, each having an associated value (cost, benefit, etc.).

Our goal is to find a solution that minimizes or maximizes this value.

**Examples:**
- **Interval scheduling:** Maximize number of scheduled activities
- **Minimum spanning tree:** Find a spanning tree of minimum cost
- **Shortest paths:** Find a spanning tree that minimizes the cost of all root-vertex paths
- **Minimum-length codes:** Find a code that encodes the given text in the minimum number of bits
A Greedy Framework for Interval Scheduling

**Find-Schedule** ($S'$)

1. $S' \leftarrow \emptyset$
2. **while** $S$ is not empty
3. **do** pick an interval $I$ from $S$
4. add $I$ to $S'$
5. remove all intervals from $S$ that conflict with $I$
6. **return** $S'$
A Greedy Framework for Interval Scheduling

**FIND-SCHEDULE**($S$)
1. $S' \leftarrow \emptyset$
2. while $S$ is not empty
3. do pick an interval $I$ from $S$
4. add $I$ to $S'$
5. remove all intervals from $S$ that conflict with $I$
6. return $S'$

**Main questions:**
A Greedy Framework for Interval Scheduling

**Find-Schedule**($S$)

1. $S' \leftarrow \emptyset$
2. **while** $S$ is not empty
3. **do** pick an interval $I$ from $S$
4. add $I$ to $S'$
5. remove all intervals from $S$ that conflict with $I$
6. **return** $S'$

**Main questions:**
- Can we choose an arbitrary interval $I$ in each iteration?
A Greedy Framework for Interval Scheduling

**Find-Schedule** $(S')$
1. $S' \leftarrow \emptyset$
2. While $S$ is not empty
3. Do pick an interval $I$ from $S$
4. Add $I$ to $S'$
5. Remove all intervals from $S$ that conflict with $I$
6. Return $S'$

Main questions:
- Can we choose an arbitrary interval $I$ in each iteration?
- How do we choose interval $I$ in each iteration?
Greedy Strategies for Interval Scheduling
**Greedy Strategies for Interval Scheduling**

- Choose the interval that begins first
Choose the interval that begins first
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval

- Choose the interval that has the fewest conflicts
Greedy Strategies for Interval Scheduling

- Choose the interval that begins first

- Choose the shortest interval

- Choose the interval that has the fewest conflicts
**The Strategy That Works**

\[ \text{FIND-SCHEDULE}(S) \]
1. \( S' \leftarrow \emptyset \)
2. \textbf{while} \( S \) is not empty
3. \textbf{do} let \( I \) be the interval in \( S \) that ends first
4. \hspace{1em} add \( I \) to \( S' \)
5. \hspace{1em} remove all intervals from \( S \) that conflict with \( I \)
6. \textbf{return} \( S' \)
**The Strategy That Works**

**FIND-SCHEDULE**$(S)$

1. $S' \leftarrow \emptyset$
2. while $S$ is not empty
3. do let $I$ be the interval in $S$ that ends first
4. add $I$ to $S'$
5. remove all intervals from $S$ that conflict with $I$
6. return $S'$
The Strategy That Works

**FIND-SCHEDULE**(*S*)

1. \( S' \leftarrow \emptyset \)
2. while \( S \) is not empty
3. do let \( I \) be the interval in \( S \) that ends first
4. add \( I \) to \( S' \)
5. remove all intervals from \( S \) that conflict with \( I \)
6. return \( S' \)
Lemma: Procedure FIND-SCHEDULE finds a maximal-cardinality set of conflict-free intervals.
Lemma: Procedure FIND-SCHEDULE finds a maximal-cardinality set of conflict-free intervals.

Proof by induction:

Base case(s): Verify that the claim holds for a set of initial instances.

Inductive step: Prove that, if the claim holds for the first $k$ instances, it holds for the $(k + 1)$-st instance.
Lemma: Procedure \texttt{FIND-SCHEDULE} finds a maximal-cardinality set of conflict-free intervals.

Proof idea ("stay ahead" argument)

- Let $I_1 \prec I_2 \prec \cdots \prec I_k$ be the schedule we compute
- Let $O_1 \prec O_2 \prec \cdots \prec O_m$ be an optimal schedule

Prove by induction on $j$ that $I_j$ ends no later than $O_j$. 
Lemma: Procedure \textsc{Find-Schedule} finds a maximal-cardinality set of conflict-free intervals.

Proof idea ("stay ahead” argument)

- Let $I_1 \prec I_2 \prec \cdots \prec I_k$ be the schedule we compute
- Let $O_1 \prec O_2 \prec \cdots \prec O_m$ be an optimal schedule

Prove by induction on $j$ that $I_j$ ends no later than $O_j$.

\[ \therefore \text{ If } k < m, \text{ the algorithm could have scheduled } O_{k+1} \text{ after } I_k. \]
Implementing the Algorithm

**Find-Schedule**($S$)

1. $S' \leftarrow \emptyset \quad \triangleright \text{Represent } S' \text{ as a linked list}$
2. sort the intervals by increasing finish time
3. $\triangleright \text{Denote the intervals as } I_1, I_2, \ldots, I_n, \text{ sorted by finish time}$
4. $\triangleright \text{The start and finish times of } I_j \text{ are } s_j \text{ and } f_j$
5. $f \leftarrow -\infty$
6. for $j \leftarrow 1$ to $n$
7. do if $s_j \geq f$
8. then append $I_j$ to $S'$
9. $\quad f \leftarrow f_j$
10. return $S'$
**Implementing the Algorithm**

**Find-Schedule** ($S$)

1. $S' \leftarrow \emptyset$  \hspace{1em} \triangleright \text{Represent } S' \text{ as a linked list}
2. sort the intervals by increasing finish time
3. \hspace{1em} \triangleright \text{Denote the intervals as } I_1, I_2, \ldots, I_n, \text{ sorted by finish time}
4. \hspace{1em} \triangleright \text{The start and finish times of } I_j \text{ are } s_j \text{ and } f_j
5. $f \leftarrow -\infty$
6. \textbf{for} $j \leftarrow 1$ \textbf{to} $n$
7. \hspace{1em} \textbf{do if} $s_j \geq f$
8. \hspace{1em} \hspace{1em} \textbf{then} append $I_j$ to $S'$
9. \hspace{1em} \hspace{1em} $f \leftarrow f_j$
10. \textbf{return} $S'$

**Lemma:** A maximum-cardinality set of conflict-free intervals can be found in $O(n \lg n)$ time.
Building a Minimum Spanning Tree Greedily
**Greedy choice:** Pick the shortest edge
**Greedy choice:** Pick the shortest edge that connects two previously disconnected vertices.
**Building a Minimum Spanning Tree Greedily**

**Greedy choice:** Pick the shortest edge that connects two previously disconnected vertices.

![Graph illustration]

**Kruskal(G)**

1. \( T \leftarrow \emptyset \)
2. **while** there are two vertices that are disconnected
3. **do** let \( e \) be the cheapest edge connecting two previously disconnected vertices
4. \( T \leftarrow T \cup \{e\} \)
Assumption: No two edges have the same weight.

Theorem: Let $S$ be a proper subset of $V$, and let $e$ be the cheapest edge with exactly one endpoint in $S$. Then every minimum spanning tree of $G$ contains edge $e$.

Proof: An exchange argument
**Assumption:** No two edges have the same weight.

**Theorem:** Let $S$ be a proper subset of $V$, and let $e$ be the cheapest edge with exactly one endpoint in $S$. Then every minimum spanning tree of $G$ contains edge $e$.

**Proof:** An exchange argument
**Lemma:** Kruskal’s algorithm computes a minimum spanning tree.
Correctness of Kruskal’s Algorithm

**Lemma:** Kruskal’s algorithm computes a minimum spanning tree.
Enforcing Distinct Edge Weights

**Redefinition of edge weights:**
- Let the edges be $e_1, e_2, \ldots, e_m$.
- Define $w'(e_i) = (w(e_i), i)$.
- Define addition of pairs as $(a, b) + (c, d) = (a + c, b + d)$.
- Define comparison as $(a, b) < (c, d)$ if $a < c$ or $a = c$ and $b < d$.

**Lemma:** A minimum spanning tree w.r.t. weight function $w'$ is also a minimum spanning tree w.r.t. weight function $w$.

**Proof:**

If $w(T_1) < w(T_2)$, then $w'(T_1) < w'(T_2)$.
**Implementing Kruskal’s Algorithm**

\[ \text{Kruskal}(G) \]

1. \( T \leftarrow \emptyset \)
2. while there are two vertices that are disconnected
3. do let \( e \) be the cheapest edge connecting two previously disconnected vertices
4. \( T \leftarrow T \cup \{e\} \)

\[ \text{Kruskal}(G) \]

1. \( T \leftarrow (V, \emptyset) \)
2. sort the edges in \( G \) by increasing weight
3. for every edge \((v, w)\) of \( G \), in sorted order
4. do if \( v \) and \( w \) belong to different connected components of \( T \)
5. then add edge \((v, w)\) to \( T \)
Union-Find Data Structure

Given a set $S$ of $n$ elements, maintain a partition of $S$ into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

- **Union($x, y$):** Replace sets $S_i$ and $S_j$ such that $x \in S_i$ and $y \in S_j$ with $S_i \cup S_j$ in the current partition.

- **Find($x$):** Returns a representative $r(S_j) \in S_j$ of the set $S_j$ that contains $x$.

In particular, Find($x$) = Find($y$) if and only if $x$ and $y$ belong to the same set.
Union-Find Data Structure

Given a set $S$ of $n$ elements, maintain a partition of $S$ into subsets $S_1, S_2, \ldots, S_k$.

Support the following operations:

- **Union**($x, y$): Replace sets $S_i$ and $S_j$ such that $x \in S_i$ and $y \in S_j$ with $S_i \cup S_j$ in the current partition.

- **Find**($x$): Returns a representative $r(S_j) \in S_j$ of the set $S_j$ that contains $x$.

In particular, Find($x$) = Find($y$) if and only if $x$ and $y$ belong to the same set.
Kruskal’s Algorithm Using Union-Find

Kruskal$(G)$

1. $T \leftarrow (V, \emptyset)$
2. sort the edges in $G$ by increasing weight
3. for every edge $(v, w)$ of $G$, in sorted order
4. do if $\text{Find}(v) \neq \text{Find}(w)$
5. then add edge $(v, w)$ to $T$
6. Union$(v, w)$
Kruskal’s Algorithm Using Union-Find

**Kruskal**($G$)

1. $T \leftarrow (V, \emptyset)$
2. sort the edges in $G$ by increasing weight
3. for every edge $(v, w)$ of $G$, in sorted order
   4. do if $\text{Find}(v) \not\equiv \text{Find}(w)$
   5. then add edge $(v, w)$ to $T$
   6. $\text{Union}(v, w)$

**Lemma:** Kruskal’s algorithm takes $O(m \log n)$ time, plus the cost of $2m$ Find and $n - 1$ Union operations.
A Simple Union-Find Structure

List node:
- Pointers to predecessor and successor
- Pointer to list head
- Pointer to list tail (only valid for head node)
- List size (only valid for head node)
**Operations**

**FIND**(\(x\))
1. return key(head(\(x\)))

**UNION**(\(x, y\))
1. if listSize(head(\(x\))) < listSize(head(\(y\)))
2. then swap \(x \leftrightarrow y\)
3. pred(head(\(y\))) ← tail(head(\(x\)))
4. succ(tail(head(\(x\)))) ← head(\(y\))
5. listSize(head(\(x\))) ← listSize(head(\(x\))) + listSize(head(\(y\)))
6. tail(head(\(x\))) ← tail(head(\(y\)))
7. \(z ← head(y)\)
8. while \(z ≠ nil\)
9. do head(\(z\)) ← head(\(x\))
10. \(z ← succ(z)\)
**Lemma:** A *Find* operation takes constant time.
Lemma: A Find operation takes constant time.

Total cost of all Union operations:

\[ C(n) = \mathcal{O}\left(\sum_{x \in S} c(x)\right), \]

where \( c(x) = \) total number of times element \( x \) is touched (\( x \)'s head pointer changes).
Lemma: A Find operation takes constant time.

Total cost of all Union operations:

\[ C(n) = \mathcal{O} \left( \sum_{x \in S} c(x) \right), \]

where \( c(x) = \) total number of times element \( x \) is touched (\( x \)'s head pointer changes).

Let \( s(x, i) = \) size of the list containing \( x \) after \( x \) is touched \( i \) times.

Lemma: \( s(x, i) \geq 2^i \), for all \( x \in S \) and \( i \geq 0 \).
Corollary: $c(x) \leq \lg n$. 
Corollary: \( c(x) \leq \lg n \).

Corollary: \( C(n) \leq \mathcal{O}(n \lg n) \).
Corollary: \( c(x) \leq \lg n \).

Corollary: \( C(n) \leq \mathcal{O}(n \lg n) \).

Corollary: *Kruskal’s algorithm takes* \( \mathcal{O}((n + m) \lg(n + m)) \) *time.*
Graph Exploration and the Cut Theorem

Explored (S)

"Explorable"

Source

Unexplored
**Greedy choice:** From among all edges with exactly one endpoint in $S$, pick the cheapest.
The Abstract Data Type Priority Queue

**Operations:**

**INSERT**\((Q, x, p)\): Insert element \(x\) into priority queue \(Q\) and give it priority \(p\)

**DELETE**\((Q, x)\): Delete element \(x\) from priority queue \(Q\)

**FIND-MIN**\((Q)\): Find and return the element with minimum priority in \(Q\)

**DELETE-MIN**\((Q)\): Delete the element with minimum priority from \(Q\) and return it

**DECREASE-KEY**\((Q, x, p)\): Decrease the priority of element \(x\) to \(p\)
The Binary Heap: A Priority Queue

**Perfect binary tree:**
- All levels full, except possibly last one.
- All nodes on last level are as far left as possible.
**The Binary Heap: A Priority Queue**

**Perfect binary tree:**
- All levels full, except possibly last one.
- All nodes on last level are as far left as possible.

**Binary min-heap property:**
For the keys $k_v$ and $k_u$ of any node $v$ and its parent $u$, $k_u \leq k_v$. 
The Binary Heap: A Priority Queue

Perfect binary tree:
- All levels full, except possibly last one.
- All nodes on last level are as far left as possible.

Binary min-heap property:
For the keys $k_v$ and $k_u$ of any node $v$ and its parent $u$, $k_u \leq k_v$.

(For the keys $k_v$ and $k_u$ of any node $v$ and an ancestor $u$ of $v$, $k_u \leq k_v$.)
Implicit Representation of Perfect Binary Trees

1

2 3

4 5 7 6

11 10 9 8 12

CSci 3110 • Greedy Algorithms • 29/63
Implicit Representation of Perfect Binary Trees

1

2 3

4 5 7 6

11 10 9 8 12

left(i) = 2i
right(i) = 2i + 1
parent(i) = \lfloor i/2 \rfloor
Heap Operations

1. Insert
2. Replace
3. Heapify-Up
4. Heapify-Down
**Heap Operations**

FIND-MIN($Q$)

1. return $Q[1]$
**Heap Operations**

**FIND-MIN(Q)**
1. return $Q[1]$

**INSERT(Q, x, p)**
1. $\text{size}(Q) \leftarrow \text{size}(Q) + 1$
2. $Q[\text{size}(Q)] \leftarrow (x, p)$
3. Heapify-Up($Q, \text{size}(Q)$)
**Heap Operations**

**FIND-MIN**(\(Q\))
1. \textbf{return} \(Q[1]\)

**INSERT**(\(Q, x, p\))
1. \(\text{size}(Q) \leftarrow \text{size}(Q) + 1\)
2. \(Q[\text{size}(Q)] \leftarrow (x, p)\)
3. \text{Heapify-Up}(Q, \text{size}(Q))

**DELETE**(\(Q, i\))
1. \(Q[i] \leftarrow Q[\text{size}(Q)]\)
2. \(\text{size}(Q) \leftarrow \text{size}(Q) - 1\)
3. \textbf{if} \(Q[i] < Q[\text{parent}(i)]\)
4. \textbf{then} \text{Heapify-Up}(Q, i)
5. \textbf{else} \text{Heapify-Down}(Q, i)
Restoring the Heap Property (1)

**HEAPIFY-UP**(\(Q, i\))

1. **while** \(i \neq 1\) and \(Q[i] < Q[\text{parent}(i)]\)
2. **do** swap \(Q[i] \leftrightarrow Q[\text{parent}(i)]\)
3. \(i \leftarrow \text{parent}(i)\)
**Restoring the Heap Property (1)**

**HEAPIFY-UP**\((Q, i)\)

1. while \(i \neq 1\) and \(Q[i] < Q[\text{parent}(i)]\)
2. do swap \(Q[i] \leftrightarrow Q[\text{parent}(i)]\)
3. \(i \leftarrow \text{parent}(i)\)

---

**Lemma:** If the only violation of the heap property is between a node \(Q[i]\) and its ancestors, the procedure Heapify-Up\((Q, i)\) restores the heap property of \(Q\).
Restoring the Heap Property (2)

**HEAPIFY-DOWN**($Q, i$)

1. smallest ← $i$
2. $i ← 0$
3. \[ \textbf{while } i \neq \text{smallest} \]
4. \[ \textbf{do } i ← \text{smallest} \]
5. \[ \textbf{if } \text{left}(i) \leq \text{heap-size}(Q) \text{ and } \]
   \[ Q[\text{left}(i)] < Q[\text{smallest}] \]
6. \[ \textbf{then } \text{smallest} ← \text{left}(i) \]
7. \[ \textbf{if } \text{right}(i) \leq \text{heap-size}(Q) \text{ and } \]
   \[ Q[\text{right}(i)] < Q[\text{smallest}] \]
8. \[ \textbf{then } \text{smallest} ← \text{right}(i) \]
9. \[ \text{swap } Q[i] ↔ Q[\text{smallest}] \]
Restoring the Heap Property (2)

**HEAPIFY-DOWN**($Q, i$)

1. smallest ← $i$
2. $i$ ← 0
3. while $i \neq$ smallest
4. do $i$ ← smallest
5. if left($i$) ≤ heap-size($Q$) and $Q[\text{left}(i)] < Q[\text{smallest}]$
6. then smallest ← left($i$)
7. if right($i$) ≤ heap-size($Q$) and $Q[\text{right}(i)] < Q[\text{smallest}]$
8. then smallest ← right($i$)
9. swap $Q[i] \leftrightarrow Q[\text{smallest}]

**Lemma:** If the only violation of the heap property is between a node $Q[i]$ and its descendants, the procedure **Heapify-Down**($Q, i$) restores the heap property of $Q$. 
Complexity of Heap Operations
Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $O(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.
Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $O(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.

Observation: Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key perform one Heapify-Up or Heapify-Down operation each.
Complexity of Heap Operations

**Observation:** Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $O(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.

**Observation:** Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key perform one Heapify-Up or Heapify-Down operation each.

**Observation:** Operations Heapify-Up and Heapify-Down take time proportional to the height of the heap, that is, $O(\lg n)$. 
**Observation:** Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key take $O(1)$ time, excluding the time spent in Heapify-Up and Heapify-Down operations.

**Observation:** Operations Find-Min, Insert, Delete, Delete-Min, and Decrease-Key perform one Heapify-Up or Heapify-Down operation each.

**Observation:** Operations Heapify-Up and Heapify-Down take time proportional to the height of the heap, that is, $O(\log n)$.

**Corollary:** Operations Insert, Delete, Delete-Min, and Decrease-Key take $O(\log n)$ time. Operation Find-Min takes $O(1)$ time.
Prim’s Algorithm

\textbf{Prim}(G)

1. Mark every vertex of G as unexplored
2. Mark every edge of G as non-tree edge
3. \( s \leftarrow \) some vertex of G
4. \( Q \leftarrow V(G) \)
5. Give every vertex priority \( \infty \), except \( s \):
   \[ p(s) = 0 \]
6. \( e(s) \leftarrow \) nil
7. \textbf{while} \( Q \) is not empty
8. \hspace{1em} \textbf{do} \( v \leftarrow \) Delete-Min(\( Q \))
9. \hspace{1em} \text{Mark vertex} \( v \) \text{ as visited}
10. \hspace{1em} \text{Mark edge} \( e(v) \) \text{ as tree edge}
11. \hspace{2em} \textbf{for} every edge \( (v, w) \) \( \in \text{Adj}(v) \)
12. \hspace{3em} \textbf{do if} vertex \( w \) \text{ is unexplored and}
13. \hspace{4em} \text{then} Decrease-Key
14. \hspace{6em} \text{Key}(Q, w, w(v, w))
15. \hspace{7em} \text{e}(w) \leftarrow (v, w)
Prim's Algorithm

**Prim**($G$)

1. Mark every vertex of $G$ as unexplored
2. Mark every edge of $G$ as non-tree edge
3. $s \leftarrow$ some vertex of $G$
4. $Q \leftarrow V(G)$
5. Give every vertex priority $\infty$, except $s$:
   
   \[ p(s) = 0 \]
6. $e(s) \leftarrow$ nil
7. while $Q$ is not empty
   
   do $v \leftarrow$ Delete-Min($Q$)
8. Mark vertex $v$ as visited
9. Mark edge $e(v)$ as tree edge
10. for every edge $(v, w) \in \text{Adj}(v)$
11. do if vertex $w$ is unexplored and $p(w) > w(v, w)$
12. then Decrease-Key($Q, w, w(v, w)$)
13. 
   \[ e(w) \leftarrow (v, w) \]

**Running time:**
Prim's Algorithm

\textbf{Prim}(G)

1. Mark every vertex of \( G \) as unexplored
2. Mark every edge of \( G \) as non-tree edge
3. \( s \leftarrow \) some vertex of \( G \)
4. \( Q \leftarrow V(G) \)
5. Give every vertex priority \( \infty \), except \( s \):
   \[ p(s) = 0 \]
6. \( e(s) \leftarrow \text{nil} \)
7. \textbf{while} \( Q \) is not empty
8. \hspace{1em} \textbf{do} \hspace{1em} \( v \leftarrow \text{Delete-Min}(Q) \)
9. \hspace{2em} Mark vertex \( v \) as visited
10. \hspace{1em} Mark edge \( e(v) \) as tree edge
11. \hspace{1em} \textbf{for} \hspace{1em} every edge \( (v, w) \in \text{Adj}(v) \)
12. \hspace{2em} \textbf{do if} \hspace{1em} vertex \( w \) is unexplored and
   \hspace{3em} \( p(w) > w(v, w) \)
13. \hspace{2em} \hspace{1em} \textbf{then} \hspace{1em} \text{Decrease-Key}(Q, w, w(v, w))
14. \hspace{2em} \hspace{1em} \textbf{then} \hspace{1em} \( e(w) \leftarrow (v, w) \)

\textbf{Running time:}

- \( n \) Insert operation
## Prim’s Algorithm

**Prim(G)**

1. Mark every vertex of $G$ as unexplored
2. Mark every edge of $G$ as non-tree edge
3. $s \leftarrow$ some vertex of $G$
4. $Q \leftarrow V(G)$
5. Give every vertex priority $\infty$, except $s$:
   
   $p(s) = 0$
6. $e(s) \leftarrow \text{nil}$
7. while $Q$ is not empty
6. do $v \leftarrow \text{Delete-Min}(Q)$
9. Mark vertex $v$ as visited
10. Mark edge $e(v)$ as tree edge
11. for every edge $(v, w) \in \text{Adj}(v)$
12. do if vertex $w$ is unexplored and $p(w) > w(v, w)$
13. then Decrease-Key($Q, w, w(v, w)$)
14. $e(w) \leftarrow (v, w)$

### Running time:

- $n$ Insert operation
- $n$ Delete-Min operations
Prim’s Algorithm

**Prim\((G)\)**
1. Mark every vertex of \(G\) as unexplored
2. Mark every edge of \(G\) as non-tree edge
3. \(s \leftarrow \text{some vertex of } G\)
4. \(Q \leftarrow V(G)\)
5. Give every vertex priority \(\infty\), except \(s\):
   \[ p(s) = 0 \]
6. \(e(s) \leftarrow \text{nil}\)
7. **while** \(Q\) is not empty
   8. **do** \(v \leftarrow \text{Delete-Min}(Q)\)
   9. Mark vertex \(v\) as visited
   10. Mark edge \(e(v)\) as tree edge
   11. **for** every edge \((v, w) \in \text{Adj}(v)\)
   12. **do if** vertex \(w\) is unexplored and \(p(w) > w(v, w)\)
   13. **then** Decrease-Key \((Q, w, w(v, w))\)
   14. \(e(w) \leftarrow (v, w)\)

**Running time:**
- \(n\) Insert operation
- \(n\) Delete-Min operations
- \(\mathcal{O}(m)\) Decrease-Key operations
**Prim’s Algorithm**

\[ \text{Prim}(G) \]

1. Mark every vertex of \( G \) as unexplored
2. Mark every edge of \( G \) as non-tree edge
3. \( s \leftarrow \) some vertex of \( G \)
4. \( Q \leftarrow V(G) \)
5. Give every vertex priority \( \infty \), except \( s \):
   \[ p(s) = 0 \]
6. \( e(s) \leftarrow \) nil
7. **while** \( Q \) is not empty
8. **do** \( v \leftarrow \) Delete-Min\((Q)\)
9. Mark vertex \( v \) as visited
10. Mark edge \( e(v) \) as tree edge
11. **for** every edge \( (v, w) \in \text{Adj}(v) \)
12. **do if** vertex \( w \) is unexplored and
   \[ p(w) > w(v, w) \]
13. **then** Decrease-Key\((Q, w, w(v, w))\)
14. \( e(w) \leftarrow (v, w) \)

**Running time:**
- \( n \) Insert operation
- \( n \) Delete-Min operations
- \( \mathcal{O}(m) \) Decrease-Key operations
- Each costs \( \mathcal{O}(\lg n) \) time
**Running time:**

- $n$ Insert operation
- $n$ Delete-Min operations
- $O(m)$ Decrease-Key operations
- Each costs $O(\lg n)$ time

*Total: $O(m \lg n)$*
**Prim’s Algorithm**

**Prim(G)**
1. Mark every vertex of $G$ as unexplored
2. Mark every edge of $G$ as non-tree edge
3. $s \leftarrow$ some vertex of $G$
4. $Q \leftarrow V(G)$
5. Give every vertex priority $\infty$, except $s$:
   \[ p(s) = 0 \]
6. $e(s) \leftarrow$ nil
7. **while** $Q$ is not empty
   8. **do** $v \leftarrow$ Delete-Min($Q$)
   9. Mark vertex $v$ as visited
   10. Mark edge $e(v)$ as tree edge
   11. **for** every edge $(v, w) \in \text{Adj}(v)$
   12. **do if** vertex $w$ is unexplored and $p(w) > w(v, w)$
      \[ e(w) \leftarrow (v, w) \]
     **then** Decrease-Key($Q, w, w(v, w)$)
14.

**Running time:**
- $n$ Insert operation
- $n$ Delete-Min operations
- $O(m)$ Decrease-Key operations
- Each costs $O(lg\ n)$ time

**Total:** $O(m \ lg\ n)$

**Fibonacci heaps:**
- Delete-Min: $O(lg\ n)$
- Insert: $O(1)$
- Decrease-Key: $O(1)$

**Total:** $O(n \ lg\ n + m)$
A Greedy Choice for Shortest Paths

"Explorable"

Explored

Source

Unexplored
A Greedy Choice for Shortest Paths

**Greedy choice:** Pick the vertex $v$ with the shortest path that contains only explored vertices besides $v$. 
**Dijkstra’s Algorithm**

\[ \text{\textsc{Dijkstra}}(G, s) \]

1. Mark every vertex of \( G \) as unexplored
2. Mark every edge of \( G \) as non-tree edge
3. \( Q \leftarrow V(G) \)
4. Give every vertex priority \( \infty \), except \( s \): \( p(s) = 0 \)
5. \( e(s) \leftarrow \text{nil} \)
6. \textbf{while} \( Q \) is not empty
7. \hspace{1em} \textbf{do} \( (v, p) \leftarrow \text{Delete-Min}(Q) \)
8. \hspace{2em} Mark vertex \( v \) as visited
9. \hspace{2em} \( d(v) \leftarrow p \)
10. \hspace{1em} Mark edge \( e(v) \) as tree edge
11. \hspace{1em} \textbf{for} every edge \( (v, w) \in \text{Adj}(v) \)
12. \hspace{2em} \textbf{do if} vertex \( w \) is unexplored and \( p(w) > d(v) + w(v, w) \)
13. \hspace{3em} \textbf{then} \( \text{Decrease-Key}(Q, w, d(v) + w(v, w)) \)
14. \hspace{3em} \( e(w) \leftarrow (v, w) \)
Dijkstra’s Algorithm

Dijkstra($G, s$)
1. Mark every vertex of $G$ as unexplored
2. Mark every edge of $G$ as non-tree edge
3. $Q \leftarrow V(G)$
4. Give every vertex priority $\infty$, except $s$: $p(s) = 0$
5. $e(s) \leftarrow$ nil
6. while $Q$ is not empty
7. do $(v, p) \leftarrow$ Delete-Min($Q$)
8. Mark vertex $v$ as visited
9. $d(v) \leftarrow p$
10. Mark edge $e(v)$ as tree edge
11. for every edge $(v, w) \in$ Adj($v$)
12. do if vertex $w$ is unexplored and $p(w) > d(v) + w(v, w)$
13. then Decrease-Key($Q, w, d(v) + w(v, w)$)
14. $e(w) \leftarrow (v, w)$

Lemma: The running time of Dijkstra’s algorithm, when implemented using Fibonacci heaps, is $\mathcal{O}(n \lg n + m)$. Using binary heaps, it takes $\mathcal{O}((n + m) \lg n)$. 
Correctness of Dijkstra’s Algorithm

Lemma: If all edge weights are non-negative, Dijkstra’s algorithm correctly computes the distances of all vertices from \( s \).

Proof:

If \( \ell(P_1) > \ell(P_2) \), then \( p_v > p_w + \text{dist}(w, v) \).
\[ \therefore p_v > p_w, \text{ a contradiction.} \]
Which kinds of codes can be decoded?
Which kinds of codes can be decoded?

Consider the code:

\[
\begin{align*}
a &= 01 & m &= 10 & n &= 111 & o &= 0 \\
r &= 11 & s &= 1 & t &= 0011 & \square &= 0111
\end{align*}
\]
(Non-)Decodable Codes

Which kinds of codes can be decoded?

■ Consider the code:

\[
\begin{align*}
a &= 01 & m &= 10 & n &= 111 & o &= 0 \\
r &= 11 & s &= 1 & t &= 0011 & u &= 0111
\end{align*}
\]

■ Now you send a fan-letter to your favourite movie star. One of the sentences is

“You are a star.”

■ You encode “star” as \( \langle 1|0011|01|11 \rangle \).
Which kinds of codes can be decoded?

Consider the code:

\[
\begin{align*}
a &= 01 & m &= 10 & n &= 111 & o &= 0 \\
r &= 11 & s &= 1 & t &= 0011 & \square &= 0111
\end{align*}
\]

Now you send a fan-letter to your favourite movie star. One of the sentences is

“You are a star.”

You encode “star” as \(\langle 1|0011|01|11 \rangle\).

Your idol receives the letter and decodes the text using your coding table:

\[
\langle 100110111 \rangle = \langle 10|0|11|0|111 \rangle = “\text{moron}”
\]

Oops, you have just insulted your idol.
(Non-)Decodable Codes

Which kinds of codes can be decoded?

- Consider the code:

  \[
  \begin{align*}
  a &= 01 \\
  m &= 10 \\
  n &= 111 \\
  o &= 0 \\
  r &= 11 \\
  s &= 1 \\
  t &= 0011 \\
  \uparrow &= 0111
  \end{align*}
  \]

- Now you send a fan-letter to your favourite movie star. One of the sentences is

  “You are a star.”

- You encode “star” as \langle 1|0011|01|11 \rangle.

- Your idol receives the letter and decodes the text using your coding table:

  \[
  \langle 100110111 \rangle = \langle 10|0|11|0|111 \rangle = “moron”
  \]

- Oops, you have just insulted your idol.

Using ASCII code, this would not have happened. Why?
Prefix Codes

A *prefix code* $C$ has the property that there are no two characters $x_1 \neq x_2$ such that $C(x_1)$ is a prefix of $C(x_2)$.

**Examples:**

- Fixed-length codes:
  - ASCII
  - 16-bit integers
- Huffman codes
  
  $c = 010 \quad e = 000 \quad i = 0010 \quad k = 00110$
  $m = 00111 \quad n = 01100 \quad o = 0111 \quad p = 01101$
  $r = 110 \quad s = 111 \quad t = 1000 \quad u = 1001$
  $\square = 101$
- Number encoding: $C(i) = 0^i 1$
Prefix Codes Can Be Decoded

It suffices to show that the first character can be decoded unambiguously. (Subsequent characters are decoded iteratively.)

Assume that there are two characters $c$ and $c'$ that could be the first character of the text, and assume that $|C(c)| \leq |C(c')|$. Then $C(c)$ is a prefix of $C(c')$, a contradiction.
Prefix Codes Can Be Decoded

It suffices to show that the first character can be decoded unambiguously. (Subsequent characters are decoded iteratively.)

Assume that there are two characters $c$ and $c'$ that could be the first character of the text, and assume that $|C(c)| \leq |C(c')|$. 

\[
\begin{align*}
C(c) &\quad \underline{01010110101010100011101010110101101101} \ldots \\
C(c') &\quad \underline{101011010101010101010101010101101101101} \\
\end{align*}
\]

Then $C(c)$ is a prefix of $C(c')$, a contradiction.

\[\therefore \text{ We want a minimum-length prefix code.}\]
Let Cost(\(\mathcal{T}, C\)) be the cost of encoding a text \(\mathcal{T}\) using code \(C\)

Let \(|C(x)|\) denote the number of bits used to encode character \(x\)

Let \(f(x)\) be the frequency of character \(x\) (number of times \(x\) occurs in \(\mathcal{T}\))

Then

\[
\text{Cost}(\mathcal{T}, C) = \sum_x f(x) \cdot |C(x)|.
\]
Every prefix code can be represented as a binary tree as follows:

- Edges are labelled
  - parent–left child = 0
  - parent–right child = 1
- Leaves correspond to characters
- Code of a character = labelling of edges from root to corresponding leaf

**Lemma:** Every internal node in a binary tree corresponding to an optimal prefix code has two children.

```
c = 010
e = 000
i = 0010
k = 00110
m = 00111
n = 01100
o = 0111
p = 01101
r = 110
s = 111
t = 1000
u = 1001
\square = 101
```
Huffman’s Algorithm

i:1  k:1  m:1  n:1  p:1  t:1  u:1  o:2  r:2  s:2  u:2  e:3  c:4
Huffman’s Algorithm

![Huffman Tree Diagram]

- i:1
- k:1
- m:1
- n:1
- p:1
- t:1
- u:1
- o:2
- r:2
- s:2
- န:2
- e:3
- c:4

- m:1
- n:1
- p:1
- t:1
- u:1
- o:2
- r:2
- s:2
- န:2

2

i:1
k:1

e:3
c:4
CSci 3110 • Greedy Algorithms • 47/63
The Final Code

\[ \begin{align*}
\text{a} &= 000 & \text{i} &= 0010 & \text{k} &= 0011 & \text{m} &= 0100 \\
\text{n} &= 0101 & \text{p} &= 0110 & \text{t} &= 0111 & \text{e} &= 100 \\
\text{u} &= 1010 & \text{o} &= 1011 & \text{c} &= 110 & \text{r} &= 1110 \\
\text{s} &= 1111
\end{align*} \]
Implementing Huffman’s Algorithm

**Huffman**(*T*)

1. Compute the set *C* of characters in *T* and determine their frequencies *f*(c)
2. Create one node *v*_c for each character in *c* and insert it into priority queue *Q*
3. **while** |*Q*| > 1
4. **do** *v* ← **DELETE** _MIN_(*Q*)
5. *w* ← **DELETE** _MIN_(*Q*)
6. Create a new node *u*
7. *f*(u) ← *f*(v) + *f*(w)
8. Make *v* the left child of *u*
9. Make *w* the right child of *u*
10. **INSERT**(Q, *u*)
Implementing Huffman’s Algorithm

Huffman($T$)

1. Compute the set $C$ of characters in $T$ and determine their frequencies $f(c)$
2. Create one node $v_c$ for each character in $c$ and insert it into priority queue $Q$
3. while $|Q| > 1$
   4. do $v \leftarrow$ DELETEMIN($Q$)
   5. $w \leftarrow$ DELETEMIN($Q$)
   6. Create a new node $u$
   7. $f(u) \leftarrow f(v) + f(w)$
   8. Make $v$ the left child of $u$
   9. Make $w$ the right child of $u$
10. INSERT($Q$, $u$)

Lemma: Huffman’s algorithm takes $\mathcal{O}((n + m) \log n)$ time, where $m$ is the number of characters in $T$ and $n$ is the number of distinct characters in $T$. 
Huffman’s Algorithm is Greedy
Huffman’s Algorithm is Greedy

- By merging two trees $T_1$ and $T_2$, we add one bit to the code of every character in $T_1$ and $T_2$. 
Huffman’s Algorithm is Greedy

- By merging two trees $T_1$ and $T_2$, we add one bit to the code of every character in $T_1$ and $T_2$.
- By merging the trees with minimum frequency, we grow the encoding of the fewest characters and thereby add the fewest bits to the encoding of $T$. 
The Structure of an Optimal Tree

Lemma: There is an optimal prefix code for $T$ in which the two least frequent characters are sibling leaves.

Proof:

Assumption: $f(x) \leq f(y) \leq f(x') \leq f(y')$
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

Cost\((\mathcal{T}, C_{\mathcal{T}^{'}})\) - Cost\((\mathcal{T}, C_{\mathcal{T}})\) =
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

\[
\text{Cost}(T, C_{T'}) - \text{Cost}(T, C_T) = f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

\[
\text{Cost}(\mathcal{T}, C_{T'}) - \text{Cost}(\mathcal{T}, C_T) = f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') - \\
\quad f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y') \\
= f(x)d_T(x') + f(y)d_T(y') + f(x')d_T(x) + f(y')d_T(y) - \\
\quad f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

\[
\text{Cost}(\mathcal{T}, C_{T'}) - \text{Cost}(\mathcal{T}, C_T) = f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]

\[
= f(x)d_T(x') + f(y)d_T(y') + f(x')d_T(x) + f(y')d_T(y) - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]

\[
= (f(x) - f(x'))(d_T(x') - d_T(x)) + (f(y) - f(y'))(d_T(y') - d_T(y))
\]
Assumption:
\[ f(x) \leq f(y) \leq f(x') \leq f(y') \]

\[
\text{Cost}(\mathcal{T}, C_{T'}) - \text{Cost}(\mathcal{T}, C_T) = f(x)d_{T'}(x) + f(y)d_{T'}(y) + f(x')d_{T'}(x') + f(y')d_{T'}(y') - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]
\[
= f(x)d_T(x) + f(y)d_T(y) + f(x')d_T(x) + f(y')d_T(y) - f(x)d_T(x) - f(y)d_T(y) - f(x')d_T(x') - f(y')d_T(y')
\]
\[
= (f(x) - f(x'))(d_T(x') - d_T(x)) + (f(y) - f(y'))(d_T(y') - d_T(y))
\]
\[
\leq 0.
\]
Understanding Huffman’s Algorithm

The previous lemma captures the idea Huffman’s algorithm is based on:
The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, $x$ and $y$, siblings and replace all their occurrences in $T$ with a new character $z$
The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, $x$ and $y$, siblings and replace all their occurrences in $\mathcal{T}$ with a new character $z$

- The resulting text $\mathcal{T}'$ contains $n - 1$ distinct characters
The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, $x$ and $y$, siblings and replace all their occurrences in $T$ with a new character $z$

- The resulting text $T'$ contains $n - 1$ distinct characters

- “Recursively” find an optimal prefix code $C'$ for $T'$
The previous lemma captures the idea Huffman’s algorithm is based on:

- Make the two least frequent characters, $x$ and $y$, siblings and replace all their occurrences in $T$ with a new character $z$

- The resulting text $T'$ contains $n - 1$ distinct characters

- “Recursively” find an optimal prefix code $C'$ for $T'$

- Compute $C$ as

$$
C(c) = \begin{cases} 
C'(c) & \text{if } c \notin \{x, y\} \\
C'(z) \circ 0 & \text{if } c = x \\
C'(z) \circ 1 & \text{if } c = y 
\end{cases}
$$
Correctness of Huffman’s Algorithm

Lemma: Huffman’s algorithm computes a minimum-length prefix code for text $T$. 
Correctness of Huffman’s Algorithm

**Lemma:** Huffman’s algorithm computes a minimum-length prefix code for text $T$.

**Proof by induction on $n$:**

Base case: $(n = 2)$

![Diagram showing a tree with two leaves labeled 'x' and 'y']
Inductive step: \((n > 2)\)

\[ \text{Cost}(S) = \text{Cost}(T) - f(x) - f(y) \]

\[ \text{Cost}(S') = \text{Cost}(T') - f(x) - f(y) \]
Inductive step: \((n > 2)\)

\[
\text{Cost}(S) = \text{Cost}(T) - f(x) - f(y)
\]

\[
\text{Cost}(S') = \text{Cost}(T') - f(x) - f(y)
\]

\[
\therefore \, \text{Cost}(T') < \text{Cost}(T) \Rightarrow \text{Cost}(S') < \text{Cost}(S)
\]
Inductive step: \((n > 2)\)

- \(\text{Cost}(S) = \text{Cost}(T) - f(x) - f(y)\)
- \(\text{Cost}(S') = \text{Cost}(T') - f(x) - f(y)\)

\[
\therefore \text{Cost}(T') < \text{Cost}(T) \Rightarrow \text{Cost}(S') < \text{Cost}(S)
\]

This is a contradiction.
Greedy algorithms use natural local criteria to make progress towards a solution.

This is a vague concept.

Many good heuristics are greedy:
- Simple
- Work well in practice

Proof that a greedy algorithm produces an optimal solution:
- Induction
- “Stay ahead” arguments
- Exchange arguments