Greedy Algorithms: Greedy Stays Ahead

Algorithm Design & Analysis

Spring 2019
Outline

Greedy Algorithms

Single Resource Scheduling

Shortest Path: Dijkstra’s Algorithm
Greedy Algorithms

Greed is Good

Applications

- Resource scheduling
- Job scheduling with deadlines
- Caching
- Shortest paths in networks: including internet packet routing
- Minimum-cost spanning subgraphs
- Data compression
- Minimum-weight basis for vector space
- Approximation algorithms for hard problems
**Greed is Good**

Greedy algorithms build solutions by making locally optimal choices.
Greed is Good

Greedy algorithms build solutions by making locally optimal choices.
For a number of problems an appropriate greedy approach yields a globally optimal solution.
Greed is Good

Greedy algorithms build solutions by making locally optimal choices.
For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications
Greed is Good

Greedy algorithms build solutions by making locally optimal choices. For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
Greed is Good

Greedy algorithms build solutions by making locally optimal choices. For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
- Job scheduling with deadlines
Greed is Good

Greedy algorithms build solutions by making locally optimal choices. For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
- Job scheduling with deadlines
- Caching
Greedy algorithms build solutions by making locally optimal choices. For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
- Job scheduling with deadlines
- Caching
- Shortest paths in networks: including internet packet routing
Greedy algorithms build solutions by making locally optimal choices. For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
- Job scheduling with deadlines
- Caching
- Shortest paths in networks: including internet packet routing
- Minimum-cost spanning subgraphs
Greed is Good

Greedy algorithms build solutions by making locally optimal choices.
For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
- Job scheduling with deadlines
- Caching
- Shortest paths in networks: including internet packet routing
- Minimum-cost spanning subgraphs
- Data compression
Greed is Good

Greedy algorithms build solutions by making locally optimal choices.
For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
- Job scheduling with deadlines
- Caching
- Shortest paths in networks: including internet packet routing
- Minimum-cost spanning subgraphs
- Data compression
- Minimum-weight basis for vector space
Greed is Good

Greedy algorithms build solutions by making locally optimal choices. For a number of problems an appropriate greedy approach yields a globally optimal solution.

Applications

- Resource scheduling
- Job scheduling with deadlines
- Caching
- Shortest paths in networks: including internet packet routing
- Minimum-cost spanning subgraphs
- Data compression
- Minimum-weight basis for vector space
- Approximation algorithms for hard problems
Two Proof Techniques

Two fundamental approaches to proving correctness of greedy algorithms
Two Proof Techniques

Two fundamental approaches to proving correctness of greedy algorithms

- **Greedy Stays Ahead**: Partial greedy solution is, at all times, as good as an "equivalent" portion of any other solution.
Two Proof Techniques

Two fundamental approaches to proving correctness of greedy algorithms

- **Greedy Stays Ahead**: Partial greedy solution is, at all times, as good as an "equivalent" portion of any other solution.
- **Exchange Property**: An optimal solution can be transformed into a greedy solution without sacrificing optimality.
Single Resource Scheduling

The Problem: Given: A list of requests to use a single resource for specific time periods.
Single Resource Scheduling

The Problem: Given: A list of requests to use a single resource for specific time periods.

The input: A list $L = \{r_1, \ldots, r_n\}$ of intervals, each with start time $s(r_i)$ and finish time $f(r_i)$.
Single Resource Scheduling

The Problem: Given: A list of requests to use a single resource for specific time periods.

The input: A list \( L = \{ r_1, \ldots, r_n \} \) of intervals, each with start time \( s(r_i) \) and finish time \( f(r_i) \)

The Goal: Identify a subset of compatible intervals (no two intersect) of maximum size

Question: How to be greedy? Start times, interval lengths (short to long, long to short), fewest conflicts, finish times, ...?

Let's play:

\[(1, 2), (3, 7), (4, 6), (5, 8), (10, 11), (12, 13), (9, 14), (8, 15)\]

Answer: Order by increasing \( f(r_i) \) (finish times)

Idea: Show that first \( k \) choices made by greedy are at least as good as \( k \) earliest ending intervals in any other solution.
**Single Resource Scheduling**

*The Problem:* Given: A list of requests to use a single resource for specific time periods.

*The input:* A list $L = \{r_1, \ldots, r_n\}$ of intervals, each with start time $s(r_i)$ and finish time $f(r_i)$

*The Goal:* Identify a subset of *compatible intervals* (no two intersect) of maximum size

*Question:* How to be greedy? Start times, interval lengths (short to long, long to short), fewest conflicts, finish times, ...?
Single Resource Scheduling

The Problem: Given: A list of requests to use a single resource for specific time periods.

The input: A list \( L = \{r_1, \ldots, r_n\} \) of intervals, each with start time \( s(r_i) \) and finish time \( f(r_i) \)

The Goal: Identify a subset of compatible intervals (no two intersect) of maximum size

Question: How to be greedy? Start times, interval lengths (short to long, long to short), fewest conflicts, finish times, ...?

Let’s play:

\((11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)\)
Single Resource Scheduling

The Problem: Given: A list of requests to use a single resource for specific time periods.

The input: A list $L = \{r_1, \ldots, r_n\}$ of intervals, each with start time $s(r_i)$ and finish time $f(r_i)$

The Goal: Identify a subset of compatible intervals (no two intersect) of maximum size

Question: How to be greedy? Start times, interval lengths (short to long, long to short), fewest conflicts, finish times, ...?

Let's play:
$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$

Answer: Order by increasing $f(r_i)$ (finish times)
Single Resource Scheduling

The Problem: Given: A list of requests to use a single resource for specific time periods.

The input: A list \( L = \{r_1, \ldots, r_n\} \) of intervals, each with start time \( s(r_i) \) and finish time \( f(r_i) \)

The Goal: Identify a subset of compatible intervals (no two intersect) of maximum size

Question: How to be greedy? Start times, interval lengths (short to long, long to short), fewest conflicts, finish times, ...?

Let’s play:
\[(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)\]

Answer: Order by increasing \( f(r_i) \) (finish times)

Idea: Show that first \( k \) choices made by greedy are at least as good as \( k \) earliest ending intervals in any other solution.
Single Resource Scheduling

The algorithm

Let's try it:

\((11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)\)

Order by increasing \(f(r_i)\) (finish times)

\((1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (12, 16), (11, 17), (14, 18), (3, 15)\)
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
**Single Resource Scheduling**

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{}$

\[(1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (3, 15), (12, 16), (11, 17), (14, 18)\]
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is compatible with $S$)
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = {}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is compatible with $S$)

Let's try it:

$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$
**Single Resource Scheduling**

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \emptyset$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is *compatible with* $S$)

*Let’s try it:*

$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$

Order by increasing $f(r_i)$ (finish times)
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is compatible with $S$)

Let’s try it:

$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$

Order by increasing $f(r_i)$ (finish times)

$(1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (3, 15), (12, 16), (11, 17), (14, 18)$
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is compatible with $S$)

Let’s try it:
(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)

Order by increasing $f(r_i)$ (finish times)
(1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (3, 15), (12, 16), (11, 17), (14, 18)

Apply greedy algorithm:
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is compatible with $S$)

Let’s try it:

$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$

Order by increasing $f(r_i)$ (finish times)

$(1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (3, 15), (12, 16), (11, 17), (14, 18)$

Apply greedy algorithm: $(1, 4)$
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is compatible with $S$)

Let’s try it:

$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$

Order by increasing $f(r_i)$ (finish times)

$(1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (3, 15), (12, 16), (11, 17), (14, 18)$

Apply greedy algorithm: $(1, 4), (5, 8)$
Single Resource Scheduling

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = {}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is compatible with $S$)

Let’s try it:
$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$

Order by increasing $f(r_i)$ (finish times)
$(1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (3, 15), (12, 16), (11, 17), (14, 18)$

Apply greedy algorithm: $(1, 4), (5, 8), (9, 13)$
**Single Resource Scheduling**

The algorithm

- Sort intervals by increasing $f(r_i)$ (finish times)
- Let $S = \{\}$
- for $i = 1..n$
  - if $i^{th}$ interval intersects no interval of $S$, add it to $S$
    (if so, we say that the $i^{th}$ interval is *compatible with* $S$)

*Let’s try it:*

$(11, 17), (3, 15), (5, 8), (14, 18), (7, 10), (2, 6), (12, 16), (9, 13), (1, 4)$

Order by increasing $f(r_i)$ (finish times)

$(1, 4), (2, 6), (5, 8), (7, 10), (9, 13), (3, 15), (12, 16), (11, 17), (14, 18)$

Apply greedy algorithm: $(1, 4), (5, 8), (9, 13), (14, 18)$
Single Resource Scheduling: The Details

Lemma Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.


**Single Resource Scheduling: The Details**

*Lemma* Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

---

**Proof**

**Base Case:** True for $i = 1$: $g_1$ has left-most finish time of all.

**Induction:** Assume true for all $j < i$, and now consider $i$.

1. $f(g_i - 1) \leq f(o_i - 1)$ (induction)
2. So, $f(g_i) \leq f(o_i) \leq s(o_i)$
3. So, $o_i$ is compatible with $g_1, \ldots, g_i - 1$
4. But $g_i$ is the interval compatible with $g_1, \ldots, g_i - 1$ that has the earliest finishing time.
5. So $f(g_i) \leq f(o_i)$.

**Corollary** It cannot be, in above Lemma, that $m > k$.

*Why?*
Single Resource Scheduling: The Details

Lemma Let \( g_1, \ldots, g_k \) be the intervals selected by the greedy algorithm in the order selected; let \( o_1, \ldots, o_m \) be any other set of compatible intervals, ordered by increasing finish time.

Then, for any \( i \leq \min\{k, m\} \), \( f(g_i) \leq f(o_i) \)

Proof
Single Resource Scheduling: The Details

**Lemma** Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

**Proof**

**Base Case:** True for $i = 1$: $g_1$ has left-most finish time of all
**Single Resource Scheduling: The Details**

*Lemma* Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

*Proof*

**Base Case:** True for $i = 1$: $g_1$ has left-most finish time of all

**Induction:** Assume true for all $j < i$, and now consider $i$. 
Single Resource Scheduling: The Details

Lemma Let \( g_1, \ldots, g_k \) be the intervals selected by the greedy algorithm in the order selected; let \( o_1, \ldots, o_m \) be any other set of compatible intervals, ordered by increasing finish time.

Then, for any \( i \leq \min\{k, m\} \), \( f(g_i) \leq f(o_i) \)

Proof

Base Case: True for \( i = 1 \): \( g_1 \) has left-most finish time of all

Induction: Assume true for all \( j < i \), and now consider \( i \).
- \( f(g_{i-1}) \leq f(o_{i-1}) \), (induction)
**Single Resource Scheduling: The Details**

*Lemma* Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

*Proof*

**Base Case:** True for $i = 1$: $g_1$ has left-most finish time of all

**Induction:** Assume true for all $j < i$, and now consider $i$.

- $f(g_{i-1}) \leq f(o_{i-1})$, (induction)
- So, $f(g_{i-1}) \leq f(o_{i-1}) \leq s(o_i)$
**Single Resource Scheduling: The Details**

*Lemma* Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

*Proof*

**Base Case:** True for $i = 1$: $g_1$ has left-most finish time of all

**Induction:** Assume true for all $j < i$, and now consider $i$.

- $f(g_{i-1}) \leq f(o_{i-1})$, (induction)
- So, $f(g_{i-1}) \leq f(o_{i-1}) \leq s(o_i)$
- So, $o_i$ is compatible with $g_1, \ldots g_{i-1}$
Single Resource Scheduling: The Details

Lemma Let \( g_1, \ldots, g_k \) be the intervals selected by the greedy algorithm in the order selected; let \( o_1, \ldots, o_m \) be any other set of compatible intervals, ordered by increasing finish time.

Then, for any \( i \leq \min\{k, m\} \), \( f(g_i) \leq f(o_i) \)

Proof

**Base Case:** True for \( i = 1 \): \( g_1 \) has left-most finish time of all

**Induction:** Assume true for all \( j < i \), and now consider \( i \).

- \( f(g_{i-1}) \leq f(o_{i-1}) \), (induction)
- So, \( f(g_{i-1}) \leq f(o_{i-1}) \leq s(o_i) \)
- So, \( o_i \) is compatible with \( g_1, \ldots g_{i-1} \)
- But \( g_i \) is the interval compatible with \( g_1, \ldots g_{i-1} \) that has the earliest finishing time.

Corollary It cannot be, in above Lemma, that \( m > k \).

Why?
Single Resource Scheduling: The Details

**Lemma** Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

**Proof**

**Base Case:** True for $i = 1$: $g_1$ has left-most finish time of all

**Induction:** Assume true for all $j < i$, and now consider $i$.

- $f(g_{i-1}) \leq f(o_{i-1})$, (induction)
- So, $f(g_{i-1}) \leq f(o_{i-1}) \leq s(o_i)$
- So, $o_i$ is compatible with $g_1, \ldots g_{i-1}$
- But $g_i$ is the interval compatible with $g_1, \ldots g_{i-1}$ that has the earliest finishing time.
- So $f(g_i) \leq f(o_i)$. 

**Corollary** It cannot be, in above Lemma, that $m > k$. Why?
**Single Resource Scheduling: The Details**

*Lemma* Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

*Proof*

**Base Case:** True for $i = 1$: $g_1$ has left-most finish time of all

**Induction:** Assume true for all $j < i$, and now consider $i$.

- $f(g_{i-1}) \leq f(o_{i-1})$, (induction)
- So, $f(g_{i-1}) \leq f(o_{i-1}) \leq s(o_i)$
- So, $o_i$ is compatible with $g_1, \ldots, g_{i-1}$
- But $g_i$ is the interval compatible with $g_1, \ldots, g_{i-1}$ that has the earliest finishing time.
- So $f(g_i) \leq f(o_i)$.

**Corollary** It cannot be, in above Lemma, that $m > k$. 
Single Resource Scheduling: The Details

Lemma Let $g_1, \ldots, g_k$ be the intervals selected by the greedy algorithm in the order selected; let $o_1, \ldots, o_m$ be any other set of compatible intervals, ordered by increasing finish time.

Then, for any $i \leq \min\{k, m\}$, $f(g_i) \leq f(o_i)$

Proof

Base Case: True for $i = 1$: $g_1$ has left-most finish time of all

Induction: Assume true for all $j < i$, and now consider $i$.

- $f(g_{i-1}) \leq f(o_{i-1})$, (induction)
- So, $f(g_{i-1}) \leq f(o_{i-1}) \leq s(o_i)$
- So, $o_i$ is compatible with $g_1, \ldots g_{i-1}$
- But $g_i$ is the interval compatible with $g_1, \ldots g_{i-1}$ that has the earliest finishing time.
- So $f(g_i) \leq f(o_i)$.

Corollary It cannot be, in above Lemma, that $m > k$. Why?
Shortest $s - t$ Path in a Weighted Graph
Shortest $s - t$ Path in a Weighted Graph

Given: A graph $G = (V, E)$ with positive edge weights: that is, each edge $e \in E$ has a value $w(e) > 0$
Shortest \( s - t \) Path in a Weighted Graph

Given: A graph \( G = (V, E) \) with positive edge weights: that is, each edge \( e \in E \) has a value \( w(e) > 0 \)

**Definition**

Given a graph with positive edge weights, the *weighted path length* of a path \( P \) is the sum of the weights of the edges in the path.
Shortest $s - t$ Path in a Weighted Graph

Given: A graph $G = (V, E)$ with positive edge weights: that is, each edge $e \in E$ has a value $w(e) > 0$

**Definition**

Given a graph with positive edge weights, the *weighted path length* of a path $P$ is the sum of the weights of the edges in the path.

That is, $w(P) = \sum_{e \in P} w(e)$. We call this the *path length* of $P$.
Shortest $s$ – $t$ Path in a Weighted Graph

Given: A graph $G = (V, E)$ with positive edge weights: that is, each edge $e \in E$ has a value $w(e) > 0$

Definition
Given a graph with positive edge weights, the weighted path length of a path $P$ is the sum of the weights of the edges in the path.
That is, $w(P) = \sum_{e \in P} w(e)$. We call this the path length of $P$

The Problem: Given a graph $G = (V, E)$ with positive edge weights $w()$, and vertices $s, t \in V$, find the minimum-weight (shortest) path from $s$ to $t$. 
The Design

*Dijkstra’s Algorithm* finds the shortest paths from $s$ to *all* other vertices in $G$. 
The Design

Dijkstra’s Algorithm finds the shortest paths from $s$ to all other vertices in $G$.

The Idea: Dijkstra’s algorithm has the following key components.
The Design

Dijkstra’s Algorithm finds the shortest paths from $s$ to all other vertices in $G$.

The Idea: Dijkstra’s algorithm has the following key components

- It evolves a tree, rooted at $s$, of shortest paths to the vertices closest to $s$
Dijkstra’s Algorithm finds the shortest paths from \( s \) to all other vertices in \( G \).

**The Idea:** Dijkstra’s algorithm has the following key components

- It evolves a tree, rooted at \( s \), of shortest paths to the vertices closest to \( s \)
- It keeps a *conservative estimate* (that is, over-estimate) \( \text{dist}() \) of the shortest path length to vertices not yet in the tree
Dijkstra’s Algorithm finds the shortest paths from $s$ to all other vertices in $G$.

**The Idea:** Dijkstra’s algorithm has the following key components

- It evolves a tree, rooted at $s$, of shortest paths to the vertices closest to $s$
- It keeps a *conservative estimate* (that is, over-estimate) $\text{dist}()$ of the shortest path length to vertices not yet in the tree
- It selects the next vertex to add to the tree based on lowest estimate (Greedy: choose locally best next move)
Dijkstra’s Algorithm finds the shortest paths from $s$ to all other vertices in $G$.

**The Idea:** Dijkstra’s algorithm has the following key components

- It evolves a tree, rooted at $s$, of shortest paths to the vertices closest to $s$
- It keeps a *conservative estimate* (that is, over-estimate) $\text{dist}()$ of the shortest path length to vertices not yet in the tree
- It selects the next vertex to add to the tree based on lowest estimate (Greedy: choose locally best next move)

Let’s see an example....
Shortest \( s - t \) Path in a Weighted Graph
Estimate at vertex \( v \) is weight of shortest path \( P \) from \( A \) to \( v \) such that \( P \) consists of a path in \( T \) followed by a single edge from \( T \) to \( G - T \).
The Algorithm

Algorithm 1 Single Source Shortest Paths

1: procedure Dijkstra(G, s) \quad \triangleright \ G = (V, E) is connected
2: \quad T = \emptyset; \ S = \{s\}; \ dist[s] \leftarrow 0
3: \quad for all neighbors v of s do
4: \quad \quad dist[v] \leftarrow w(s, v); \ prior[v] \leftarrow s
5: \quad for all non-neighbors v of s do
6: \quad \quad dist[v] \leftarrow \infty
7: \quad while S \neq V do
8: \quad \quad Select v \in V - S with minimum dist[v]
9: \quad \quad Add v to S; add \{v, prior[v]\} to T
10: \quad \quad for each neighbor u \in V - S of v do
11: \quad \quad \quad if dist[v] + w(v, u) < dist[u] then
12: \quad \quad \quad \quad dist[u] = dist[v] + w(v, u)
13: \quad \quad \quad \quad prior[u] \leftarrow v
Correctness Analysis

**Note:** The edges \( \{v, prior[v]\} \) form \( T \); \( prior[v] \) is the vertex last used to update the value \( dist[v] \).
**Correctness Analysis**

**Note:** The edges \( \{v, prior[v]\} \) form \( T \); \( prior[v] \) is the vertex last used to update the value \( \text{dist}[v] \).

That is, current value of \( \text{dist}[v] = \text{dist}[prior[v]] + w(v, prior[v]) \)
Correctness Analysis

**Note:** The edges \( \{v, \text{prior}[v]\} \) form \( T \); \( \text{prior}[v] \) is the vertex last used to update the value \( \text{dist}[v] \).

That is, current value of \( \text{dist}[v] = \text{dist}[\text{prior}[v]] + w(v, \text{prior}[v]) \)

**Theorem**
After each iteration of the while loop, the set of marked edges form a tree \( T \) with root \( s \)
Correctness Analysis

Note: The edges \( \{v, \text{prior}[v]\} \) form \( T \); \( \text{prior}[v] \) is the vertex last used to update the value \( \text{dist}[v] \).

That is, current value of \( \text{dist}[v] = \text{dist}[\text{prior}[v]] + w(v, \text{prior}[v]) \)

Theorem
After each iteration of the while loop, the set of marked edges form a tree \( T \) with root \( s \)

Theorem
After each iteration of the while loop, \( T \) contains shortest paths (in \( G \)) from \( s \) to every other vertex of \( T \)
The Proof

Proof.

By induction on $|V(T)|$. Clear when $|V(T)| = 1$. Suppose the result holds for some $k = |V(T)| \geq 1$, then loop iterates again.
The Proof

Proof.
By induction on $|V(T)|$. Clear when $|V(T)| = 1$. Suppose the result holds for some $k = |V(T)| \geq 1$, then loop iterates again

- Let $v$ be the vertex added to $S$ and $\{v, x = prior[v]\}$ the edge added to $T$

...
The Proof

Proof.

By induction on $|V(T)|$. Clear when $|V(T)| = 1$. Suppose the result holds for some $k = |V(T)| \geq 1$, then loop iterates again.

- Let $v$ be the vertex added to $S$ and $\{v, x = \text{prior}[v]\}$ the edge added to $T$
- Let $P$ be the unique $s - x$ path in $T$, followed by the edge $\{x, v\}$
The Proof

Proof.

By induction on \(|V(T)|\). Clear when \(|V(T)| = 1\). Suppose the result holds for some \(k = |V(T)| \geq 1\), then loop iterates again

- Let \(v\) be the vertex added to \(S\) and \(\{v, x = prior[v]\}\) the edge added to \(T\)
- Let \(P\) be the unique \(s - x\) path in \(T\), followed by the edge \(\{x, v\}\)
- Suppose some \(s - v\) path \(P'\) in \(G\) is shorter (lower weight)
Proof.

By induction on $|V(T)|$. Clear when $|V(T)| = 1$. Suppose the result holds for some $k = |V(T)| \geq 1$, then loop iterates again.

- Let $v$ be the vertex added to $S$ and $\{v, x = \text{prior}[v]\}$ the edge added to $T$.
- Let $P$ be the unique $s - x$ path in $T$, followed by the edge $\{x, v\}$.
- Suppose some $s - v$ path $P'$ in $G$ is shorter (lower weight).
  - Let $e = \{x', v'\}$ be the first edge along $P'$ such that $x' \in S$ and $v' \notin S$. 

\[ \square \]
**The Proof**

*Proof.*

By induction on \(|V(T)|\). Clear when \(|V(T)| = 1\). Suppose the result holds for some \(k = |V(T)| \geq 1\), then loop iterates again

- Let \(v\) be the vertex added to \(S\) and \(\{v, x = \text{prior}[v]\}\) the edge added to \(T\)
- Let \(P\) be the unique \(s - x\) path in \(T\), followed by the edge \(\{x, v\}\)
- Suppose some \(s - v\) path \(P'\) in \(G\) is shorter (lower weight)
  - Let \(e = \{x', v'\}\) be the first edge along \(P'\) such that \(x' \in S\) and \(v' \notin S\).
  - Claim: The initial portion of \(P'\) from \(s\) to \(v'\) has lower weight than \(P\)

\[\square\]
The Proof

Proof.
By induction on $|V(T)|$. Clear when $|V(T)| = 1$. Suppose the result holds for some $k = |V(T)| \geq 1$, then loop iterates again

- Let $v$ be the vertex added to $S$ and $\{v, x = \text{prior}[v]\}$ the edge added to $T$
- Let $P$ be the unique $s - x$ path in $T$, followed by the edge $\{x, v\}$
- Suppose some $s - v$ path $P'$ in $G$ is shorter (lower weight)
  - Let $e = \{x', v'\}$ be the first edge along $P'$ such that $x' \in S$ and $v' \notin S$. 
  - Claim: The initial portion of $P'$ from $s$ to $v'$ has lower weight than $P$
  - Contradiction: $v'$ should have been chosen instead of $v$
Resource Analysis

How can we efficiently implement Dijkstra’s Algorithm? We need to be able to

• Visit every neighbor of a vertex.
• Maintain sets of visited (V) and unvisited vertices; mark certain edges.
• Select the unvisited vertex that minimizes dist().
• Update dist() values for unvisited vertices.
Resource Analysis

How can we efficiently implement Dijkstra’s Algorithm? We need to be able to

- Visit every neighbor of a vertex.
Resource Analysis

How can we efficiently implement Dijkstra’s Algorithm? We need to be able to

- Visit every neighbor of a vertex.
- Maintain sets of visited ($S$) and unvisited vertices; mark certain edges
How can we efficiently implement Dijkstra’s Algorithm? We need to be able to

- Visit every neighbor of a vertex.
- Maintain sets of visited (S) and unvisited vertices; mark certain edges
- Select the unvisited vertex that minimizes dist()
Resource Analysis

How can we efficiently implement Dijkstra’s Algorithm? We need to be able to

- Visit every neighbor of a vertex.
- Maintain sets of visited \( (S) \) and unvisited vertices; mark certain edges
- Select the unvisited vertex that minimizes \( \text{dist}() \)
- Update \( \text{dist}() \) values for unvisited vertices
The Structures

We use the following structures
The Structures

We use the following structures

- an adjacency list graph structure for $G$
The Structures

We use the following structures

- an adjacency list graph structure for $G$
- similar structure for $T$ (or just use $\text{prior}[]$)
The Structures

We use the following structures

- an adjacency list graph structure for $G$
- similar structure for $T$ (or just use $prior[]$)
- A priority queue to store unvisited vertices
Updating the PQ

How to update priorities in the priority queue efficiently.
Updating the PQ

How to update priorities in the priority queue efficiently.

- Recall vertices are represented by 1, \ldots, n
**Updating the PQ**

How to update priorities in the priority queue efficiently.

- Recall vertices are represented by $1, \ldots, n$
- Maintain an array $PQIndex[1..n]$ that holds the index of each vertex $v$ in the priority queue
Updating the PQ

How to update priorities in the priority queue efficiently.

- Recall vertices are represented by $1, \ldots, n$
- Maintain an array $PQIndex[1..n]$ that holds the index of each vertex $v$ in the priority queue
- If we update $dist[u]$ for some $u$, we then heapify-up from $u$’s location in the priority queue to restore heap property
How to update priorities in the priority queue efficiently:

- Recall vertices are represented by 1, \ldots, n
- Maintain an array $PQIndex[1..n]$ that holds the index of each vertex $v$ in the priority queue
- If we update $dist[u]$ for some $u$, we then heapify-up from $u$’s location in the priority queue to restore heap property
- Every time we swap two heap elements, we update $PQIndex$ for the two vertices
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $dist[]$, $prior[]$ and $PQIndex[]$
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $dist[]$, $prior[]$ and $PQIndex[]$

Time Complexity
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $dist[]$, $prior[]$ and $PQIndex[]$

Time Complexity

- A traversal of $G$ (every edge of $G$ is visited at most once): $O(n \log n + m)$
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $dist[]$, $prior[]$ and $PQIndex[]$

Time Complexity

- A traversal of $G$ (every edge of $G$ is visited at most once): $O(n \log n + m)$
  - The log $n$ is because next vertex selection takes $O(\log n)$ time
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $dist[]$, $prior[]$ and $PQIndex[]$

Time Complexity

- A traversal of $G$ (every edge of $G$ is visited at most once): $O(n \log n + m)$
  - The $\log n$ is because next vertex selection takes $O(\log n)$ time
- Construction of $T$: time proportional to its size: $O(n)$
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $dist[]$, $prior[]$ and $PQIndex[]$

Time Complexity

- A traversal of $G$ (every edge of $G$ is visited at most once): $O(n \log n + m)$
  - The $\log n$ is because next vertex selection takes $O(\log n)$ time
- Construction of $T$: time proportional to its size: $O(n)$
- Creation of priority queue: $O(n)$
**Time and Space Complexity**

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $\text{dist}[]$, $\text{prior}[]$ and $\text{PQIndex}[]$

**Time Complexity**

- A traversal of $G$ (every edge of $G$ is visited at most once):
  $O(n \log n + m)$
  - The log $n$ is because next vertex selection takes $O(\log n)$ time
- Construction of $T$: time proportional to its size: $O(n)$
- Creation of priority queue: $O(n)$
- $n$ deleteMin operations from priority queue: $O(n \log n)$
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $dist[]$, $prior[]$ and $PQIndex[]$

Time Complexity

- A traversal of $G$ (every edge of $G$ is visited at most once): $O(n \log n + m)$
  - The $\log n$ is because next vertex selection takes $O(\log n)$ time
- Construction of $T$: time proportional to its size: $O(n)$
- Creation of priority queue: $O(n)$
- $n$ deleteMin operations from priority queue: $O(n \log n)$
- At most one heapify-up or -down for each edge of $G$: $O(m \log n)$
Time and Space Complexity

We use $O(n + m)$ space: storage of $G$, $T$, the priority queue, and $\text{dist}[]$, $\text{prior}[]$ and $\text{PQIndex}[]$.

Time Complexity

- A traversal of $G$ (every edge of $G$ is visited at most once): $O(n \log n + m)$
  - The $\log n$ is because next vertex selection takes $O(\log n)$ time.
- Construction of $T$: time proportional to its size: $O(n)$
- Creation of priority queue: $O(n)$
- $n$ deleteMin operations from priority queue: $O(n \log n)$
- At most one heapify-up or -down for each edge of $G$: $O(m \log n)$

Total time:

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

$O(m \log n)$ if $G$ is connected (since $m \geq n - 1$).