Greedy Algorithms

CS 4104: Data and Algorithm Analysis

Yoseph Berhanu Alebachew

May 11, 2025

Virginia Tech

Table of contents

Reminders

- Course Survey (Due tomorrow)
- Homework 1 released

Objective

Objective

- Start discussion of different ways of designing algorithms.

Objective

- Start discussion of different ways of designing algorithms.
  - Greedy algorithms
  - Divide and conquer
  - Dynamic programming

Objective

- Start discussion of different ways of designing algorithms.
  - Greedy algorithms
  - Divide and conquer
  - Dynamic programming
- Discuss principles that can solve a variety of problem types

Objective

- Start discussion of different ways of designing algorithms.
  - Greedy algorithms
  - Divide and conquer
  - Dynamic programming
- Discuss principles that can solve a variety of problem types
- Design an algorithm, prove its correctness, analyse its complexity

Objective

- Start discussion of different ways of designing algorithms.
  - Greedy algorithms
  - Divide and conquer
  - Dynamic programming
- Discuss principles that can solve a variety of problem types
- Design an algorithm, prove its correctness, analyse its complexity
- Greedy algorithms: make the current best choice.

Interval Scheduling

Interval Scheduling Problem

Room / Time	A	B	C	D	E	F	G	H
Paral.#1 11:00-12:30 Monday 16 Sept.	**Block 1A.** Pedagogic methods: Studio teaching (4-1)	**Block 1B.** Digital technology in landscape education (2-1)	**Block 1C.** Curricula: Assessment and programme development	**Block 1D.** Teaching transdisciplinary approaches to landscape (4-1)	**Block 1E.** History of landscape education (3-1)	**Block 1F.** The ELC and landscape education	**Block 1G.** Pedagogic methods: Multisensory	**Block 1H.** [Special session] Landscape architecture education in a global research context
Paral.#2 13:30-15:00 Monday 16 Sept.	**Block 2A.** Pedagogic methods: Studio teaching (4-2)	**Block 2B.** Pedagogic methods: sustainability, ecology and planting design	**Block 2C.** Pedagogic methods: Understanding site	**Block 2D.** Teaching transdisciplinary approaches to landscape (4-2)	**Block 2E.** History of landscape education (3-2)	**Block 2F.** [Special session] UNISCAPE meeting: Landscape education after 20 years of the ELC	**Block 2G.** [Workshop] Stonesensing: Evoking meaning with stones	**Block 2H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-1)
Paral.#3 15:30-17:00 Monday 16 Sept.	**Block 3A.** Pedagogic methods: Studio teaching (4-3)	**Block 3B.** [Special session] The history and future of teaching digital methods in landscape architecture	**Block 3C.** The making of a profession	**Block 3D.** Teaching transdisciplinary approaches to landscape (4-3)	**Block 3E.** History of landscape education (3-3)	**Block 3F.** [Special session] Challenges and opportunities of landscape architecture education in the Arab world: The experience of the American University of Beirut	**Block 3G.** [Workshop] Learning to read the landscape: a methodological framework	**Block 3H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-2)
Paral.#4 10:30-12:00 Tuesday 17 Sept.	**Block 4A.** Pedagogic methods: Studio teaching (4-4)	**Block 4B.** Pedagogic methods: Student engagement and motivation	**Block 4C.** Pedagogic methods: Fieldwork	**Block 4D.** Landscape education: Ethics and values	**Block 4E.** Pedagogic methods: Teaching in a global context	**Block 4F.** [Special session] Bridging national and disciplinary boundaries: Concepts of sustainability in landscape and urban planning education	**Block 4G.** [Special session] Professional mythologies or academic consistency? Reframing the basic concepts in landscape architecture education	**Block 4H.** [Workshop] An asset to education: Introducing archives of landscape architecture in academic education
Paral.#5 14:00-15:30 Tuesday 17 Sept.	**Block 5A.** Pedagogic methods: Design thinking	**Block 5B.** Digital technology in landscape education (2-2)	**Block 5C.** Educating in a multicultural context	**Block 5D.** Teaching transdisciplinary approaches to landscape (4-4)	**Block 5E.** Pedagogic methods: Integrating theory	**Block 5F.** Visions for landscape education	**Block 5G.** [Workshop] The power of imagined landscapes	

- **Input:** Start and end time of each workshop.
- **Goal:** Compute the largest number of workshops you can be on in one day
- **Constraints:**
  - Cannot be in two places at one time.
  - Workshops may overlap.

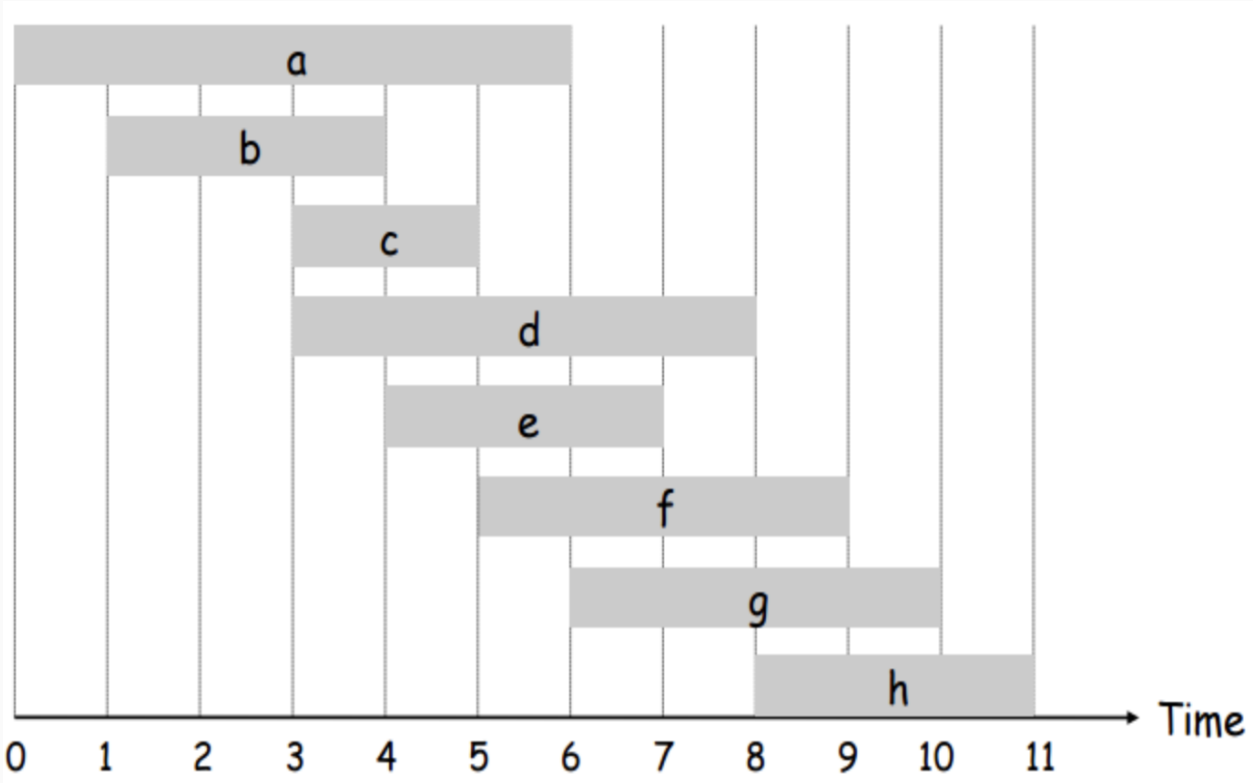
Interval Scheduling Problem

Room / Time	A	B	C	D	E	F	G	H
Paral.#1 11:00-12:30 Monday 16 Sept.	**Block 1A.** Pedagogic methods: Studio teaching (4-1)	**Block 1B.** Digital technology in landscape education (2-1)	**Block 1C.** Curricula: Assessment and programme development	**Block 1D.** Teaching transdisciplinary approaches to landscape (4-1)	**Block 1E.** History of landscape education (3-1)	**Block 1F.** The ELC and landscape education	**Block 1G.** Pedagogic methods: Multisensory	**Block 1H.** [Special session] Landscape architecture education in a global research context
Paral.#2 13:30-15:00 Monday 16 Sept.	**Block 2A.** Pedagogic methods: Studio teaching (4-2)	**Block 2B.** Pedagogic methods: sustainability, ecology and planting design	**Block 2C.** Pedagogic methods: Understanding site	**Block 2D.** Teaching transdisciplinary approaches to landscape (4-2)	**Block 2E.** History of landscape education (3-2)	**Block 2F.** [Special session] UNISCAPE meeting: Landscape education after 20 years of the ELC	**Block 2G.** [Workshop] Stonesensing: Evoking meaning with stones	**Block 2H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-1)
Paral.#3 15:30-17:00 Monday 16 Sept.	**Block 3A.** Pedagogic methods: Studio teaching (4-3)	**Block 3B.** [Special session] The history and future of teaching digital methods in landscape architecture	**Block 3C.** The making of a profession	**Block 3D.** Teaching transdisciplinary approaches to landscape (4-3)	**Block 3E.** History of landscape education (3-3)	**Block 3F.** [Special session] Challenges and opportunities of landscape architecture education in the Arab world: The experience of the American University of Beirut	**Block 3G.** [Workshop] Learning to read the landscape: a methodological framework	**Block 3H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-2)
Paral.#4 10:30-12:00 Tuesday 17 Sept.	**Block 4A.** Pedagogic methods: Studio teaching (4-4)	**Block 4B.** Pedagogic methods: Student engagement and motivation	**Block 4C.** Pedagogic methods: Fieldwork	**Block 4D.** Landscape education: Ethics and values	**Block 4E.** Pedagogic methods: Teaching in a global context	**Block 4F.** [Special session] Bridging national and disciplinary boundaries: Concepts of sustainability in landscape and urban planning education	**Block 4G.** [Special session] Professional mythologies or academic consistency? Reframing the basic concepts in landscape architecture education	**Block 4H.** [Workshop] An asset to education: Introducing archives of landscape architecture in academic education
Paral.#5 14:00-15:30 Tuesday 17 Sept.	**Block 5A.** Pedagogic methods: Design thinking	**Block 5B.** Digital technology in landscape education (2-2)	**Block 5C.** Educating in a multicultural context	**Block 5D.** Teaching transdisciplinary approaches to landscape (4-4)	**Block 5E.** Pedagogic methods: Integrating theory	**Block 5F.** Visions for landscape education	**Block 5G.** [Workshop] The power of imagined landscapes	

Note

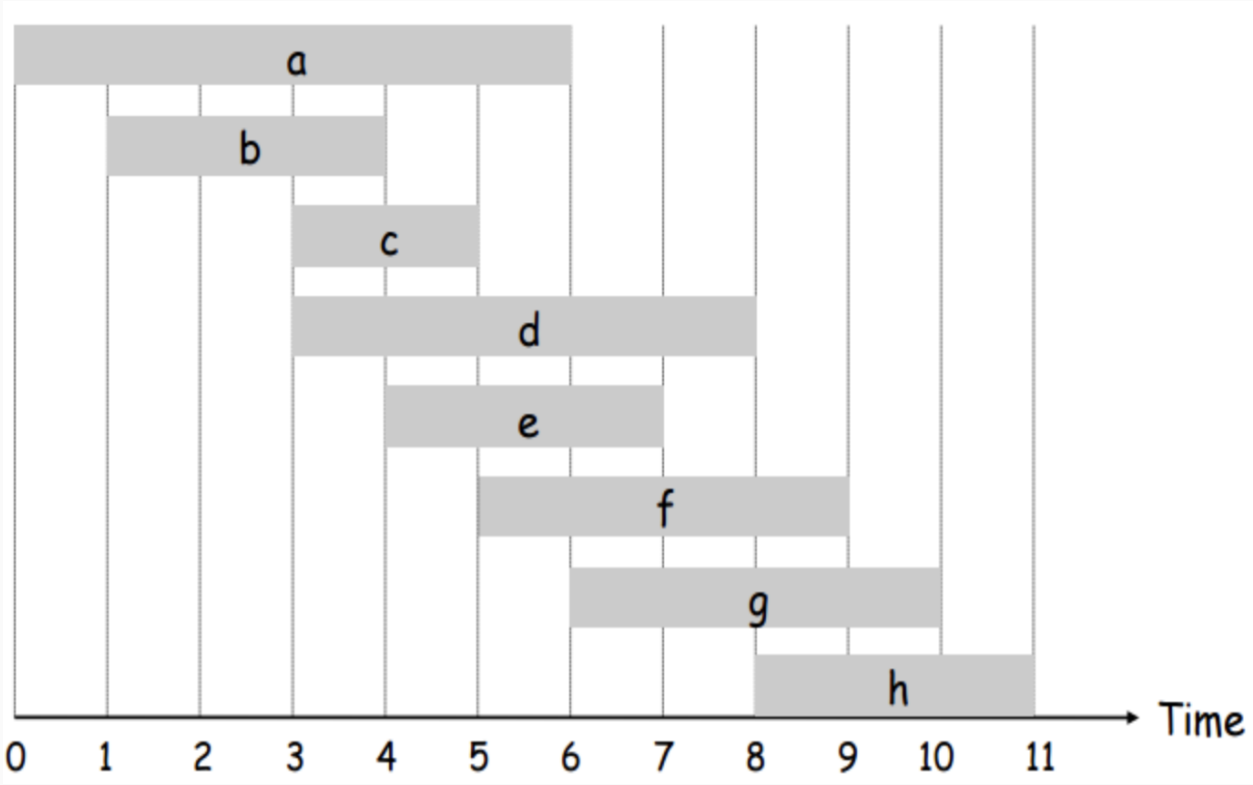
Notice that we don't have a preference for workshops. We are interested in the maximum **number** of workshops we can attend.

Interval Scheduling Problem



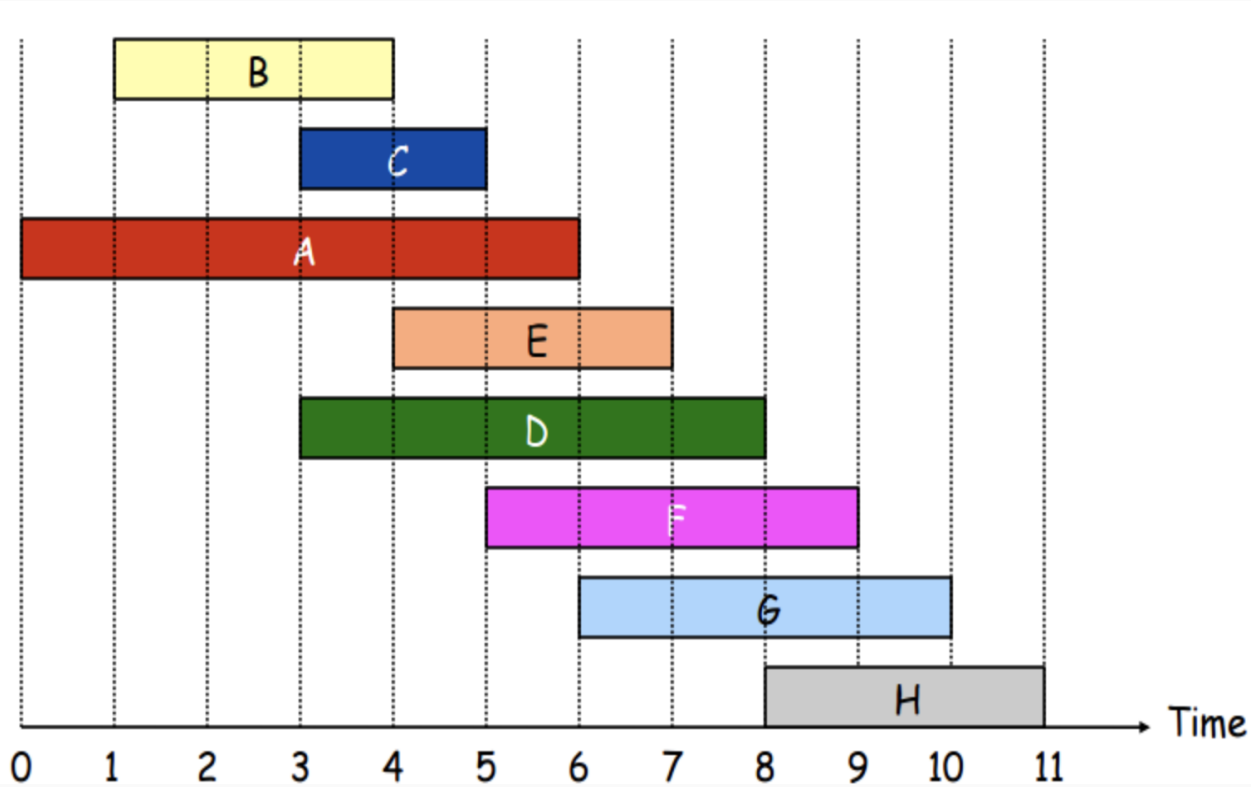
- **Input**: Set $(s(i), f(i)), 1 \leq i \leq n$ of start and finish times of n workshops.
- **Solution**: The largest subset of mutually compatible workshops.

Interval Scheduling Problem



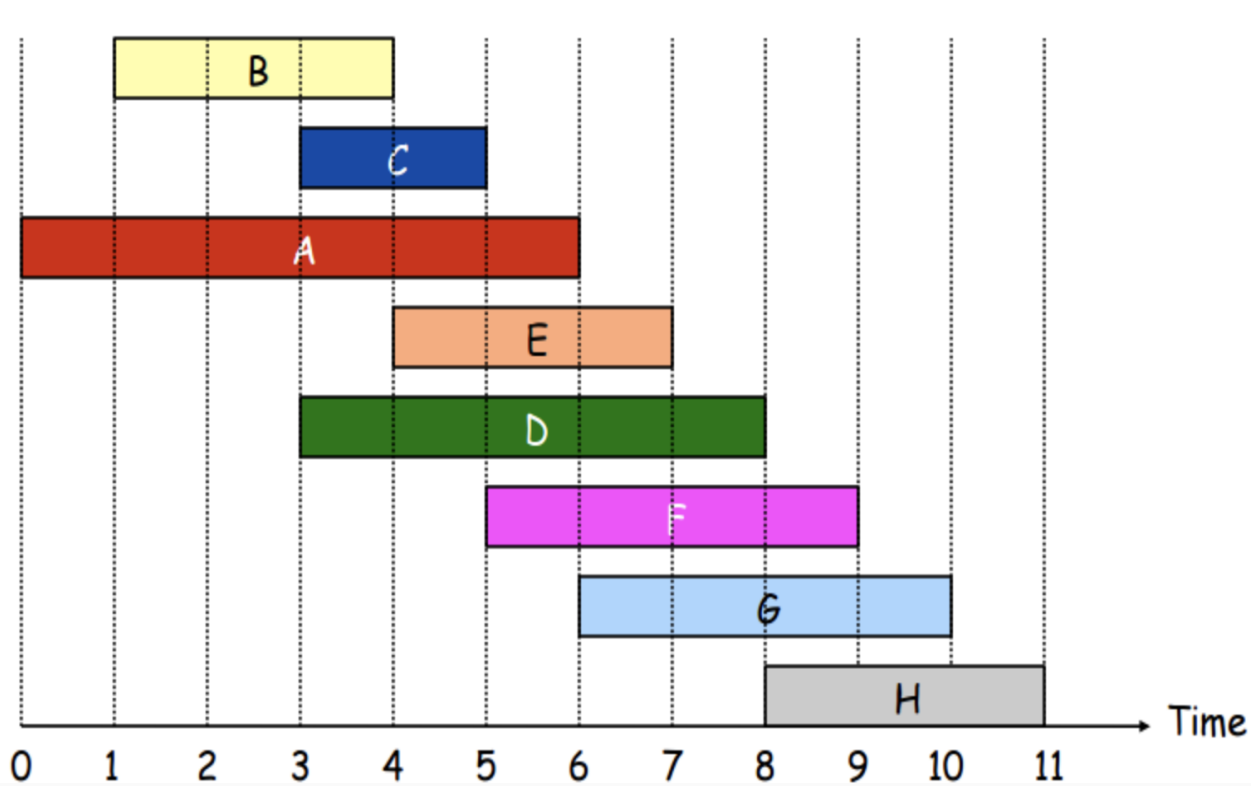
- Two workshops are *compatible* if they do not overlap.
- This problem models the situation where you have a resource, a set of fixed jobs, and you want to schedule as many jobs as possible.
- For any input set of jobs, our algorithm must provably compute the **largest** set of compatible jobs.

Example: Compatibility



Poll 1

Example: Compatibility



Poll 1

- Which jobs/workshops are compatible?
  - A and F : **No**
  - B and G : **Yes**
  - C and H : **Yes**
  - D and E : **No**
  - E and G : **No**
  - F and D : **No**

Interval Scheduling Problem: Brute Force Algorithm

- Generate all possible subsets of the given intervals.
- Check if the intervals in each subset are mutually compatible.
- Track the largest subset that is compatible.
- Is this algorithm efficient ? Poll 2

Interval Scheduling Problem: Brute Force Algorithm

- Generate all possible subsets of the given intervals.
- Check if the intervals in each subset are mutually compatible.
- Track the largest subset that is compatible.
- Is this algorithm efficient ? Poll 2
  - There are 2^n subsets and checking each subset for compatibility takes $O(n)$ time.
  - The brute force algorithm has a time complexity of $O(n \cdot 2^n)$, where n is the number of intervals.
  - This approach is computationally expensive for large n

Interval Scheduling Problem: Greedy Algorithm Intuition

- Process jobs in some order.
- Add next job to the result if it is compatible with the jobs already in the result.
- **Key question**: in what order should we process the jobs? Poll 3

Interval Scheduling Problem: Greedy Algorithm Intuition

- Process jobs in some order.
- Add next job to the result if it is compatible with the jobs already in the result.
- **Key question**: in what order should we process the jobs? Poll 3
  - **Earliest start time**: Increasing order of start time s(i).
  - **Earliest finish time**: Increasing order of finish time f(i).
  - **Shortest interval**: Increasing order of length f(i) - s(i).
  - **Fewest conflicts**: Increasing order of the number of conflicting jobs.

Interval Scheduling Problem: Greedy Algorithm with Earliest Finish Time

Algorithm 1 Schedule intervals in order of earliest Finish time (EFT)

```
function intervalScheduling(S):
    A = []
    sort(S) // Sort based on finish time
    while S is not empty:
        // pop the workshop with the earliest finish time
        workshop = S.pop()
        A.push(workshop) // Add to the return list
        for w in S:
            if not compatibleWith(workshop, w):
                S.remove(w)
    return A
```

Interval Scheduling Problem: Greedy Algorithm with Earliest Finish Time

Algorithm 2 Schedule intervals in order of earliest Finish time (EFT)

```
function intervalScheduling(S):
    A = []
    sort(S) // Sort based on finish time
    while S is not empty:
        // pop the workshop with the earliest finish time
        workshop = S.pop()
        A.push(workshop) // Add to the return list
        for w in S:
            if not compatibleWith(workshop, w):
                S.remove(w)
    return A
```

- **Claim:** A is a compatible set of jobs. Poll 4

Earliest Finish Time Intuition for Proof of Correctness

- We need to prove that $|A|$ (the number of jobs in A) is the largest possible in any set of mutually compatible jobs.
- **Key idea:** The algorithm always makes the optimal choice by selecting the job that finishes the earliest.
- This strategy leaves the most room for the remaining jobs, maximizing the number of compatible jobs.

Earliest Finish Time : Detailed Proof of Correctness

- Let O be an optimal set of workshops. Poll 5

Earliest Finish Time : Detailed Proof of Correctness

- Let O be an optimal set of workshops. Poll 5
  - We have to show that $|O| == |A|$
  - Show elements in $|O|$ and $|A|$ are the same
- Let $A = \{a_1, a_2, \ldots, a_k\}$ be the set of jobs selected by the algorithm, ordered by finish time.

Earliest Finish Time : Detailed Proof of Correctness

- Let O be an optimal set of workshops. Poll 5
  - We have to show that $|O| == |A|$
  - Show elements in $|O|$ and $|A|$ are the same
- Let $A = \{a_1, a_2, \ldots, a_k\}$ be the set of jobs selected by the algorithm, ordered by finish time.
- Let $O = \{o_1, o_2, \ldots, o_m\}$ be an optimal set of jobs, ordered by finish time, with $m \geq k$.
- We use induction to show that for all $1 \leq r \leq k$, the finish time of a_r is less than or equal to the finish time of o_r.

Proof of Correctness: Detailed

- Induction
  - **Initialization:** $r = 1$. The first job a_1 selected by the algorithm has the earliest finish time, so $f(a_1) \leq f(o_1)$.
  - **Maintenance:** Assume $f(a_i) \leq f(o_i)$ for all $i \leq r$.
    - Since a_{r+1} is chosen to have the earliest finish time after a_r, $f(a_{r+1}) \leq f(o_{r+1})$.
    - This maintains the property for a_{r+1}.
- Now let's show that $k = m$ (**termination**)
- Since both A and O are sets of mutually compatible jobs:
  - If there are more jobs in O than A, there must be some job in O that can be replaced by a job in A without causing conflicts.
- Thus, the size of A is equal to the size of O:
  - Both A and O contain the maximum number of compatible jobs.
  - Therefore, the algorithm's solution is optimal.

Greedy Property

Note

- The greedy property of the algorithm is the maximize the available time after picking a job to add to the list.
  - i.e., Leave maximum room for the remaining jobs/workshops

Interval Partitioning

Interval Partitioning Problem

Room / Time	A	B	C	D	E	F	G	H
Paral.#1 11:00-12:30 Monday 16 Sept.	**Block 1A.** Pedagogic methods: Studio teaching (4-1)	**Block 1B.** Digital technology in landscape education (2-1)	**Block 1C.** Curricula: Assessment and programme development	**Block 1D.** Teaching transdisciplinary approaches to landscape (4-1)	**Block 1E.** History of landscape education (3-1)	**Block 1F.** The ELC and landscape education	**Block 1G.** Pedagogic methods: Multisensory	**Block 1H.** [Special session] Landscape architecture education in a global research context
Paral.#2 13:30-15:00 Monday 16 Sept.	**Block 2A.** Pedagogic methods: Studio teaching (4-2)	**Block 2B.** Pedagogic methods: sustainability, ecology and planting design	**Block 2C.** Pedagogic methods: Understanding site	**Block 2D.** Teaching transdisciplinary approaches to landscape (4-2)	**Block 2E.** History of landscape education (3-2)	**Block 2F.** [Special session] UNISCAPE meeting: Landscape education after 20 years of the ELC	**Block 2G.** [Workshop] Stonesensing: Evoking meaning with stones	**Block 2H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-1)
Paral.#3 15:30-17:00 Monday 16 Sept.	**Block 3A.** Pedagogic methods: Studio teaching (4-3)	**Block 3B.** [Special session] The history and future of teaching digital methods in landscape architecture	**Block 3C.** The making of a profession	**Block 3D.** Teaching transdisciplinary approaches to landscape (4-3)	**Block 3E.** History of landscape education (3-3)	**Block 3F.** [Special session] Challenges and opportunities of landscape architecture education in the Arab world: The experience of the American University of Beirut	**Block 3G.** [Workshop] Learning to read the landscape: a methodological framework	**Block 3H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-2)
Paral.#4 10:30-12:00 Tuesday 17 Sept.	**Block 4A.** Pedagogic methods: Studio teaching (4-4)	**Block 4B.** Pedagogic methods: Student engagement and motivation	**Block 4C.** Pedagogic methods: Fieldwork	**Block 4D.** Landscape education: Ethics and values	**Block 4E.** Pedagogic methods: Teaching in a global context	**Block 4F.** [Special session] Bridging national and disciplinary boundaries: Concepts of sustainability in landscape and urban planning education	**Block 4G.** [Special session] Professional mythologies or academic consistency? Reframing the basic concepts in landscape architecture education	**Block 4H.** [Workshop] An asset to education: Introducing archives of landscape architecture in academic education
Paral.#5 14:00-15:30 Tuesday 17 Sept.	**Block 5A.** Pedagogic methods: Design thinking	**Block 5B.** Digital technology in landscape education (2-2)	**Block 5C.** Educating in a multicultural context	**Block 5D.** Teaching transdisciplinary approaches to landscape (4-4)	**Block 5E.** Pedagogic methods: Integrating theory	**Block 5F.** Visions for landscape education	**Block 5G.** [Workshop] The power of imagined landscapes	

- **Input:** Start and end time of each workshop.
- **Goal:** Compute the minimum number of rooms required to hold all workshops without conflicts.
- **Constraints:**
  - Workshop times may overlap.
  - Two workshops can't happen in the same room at the same time.

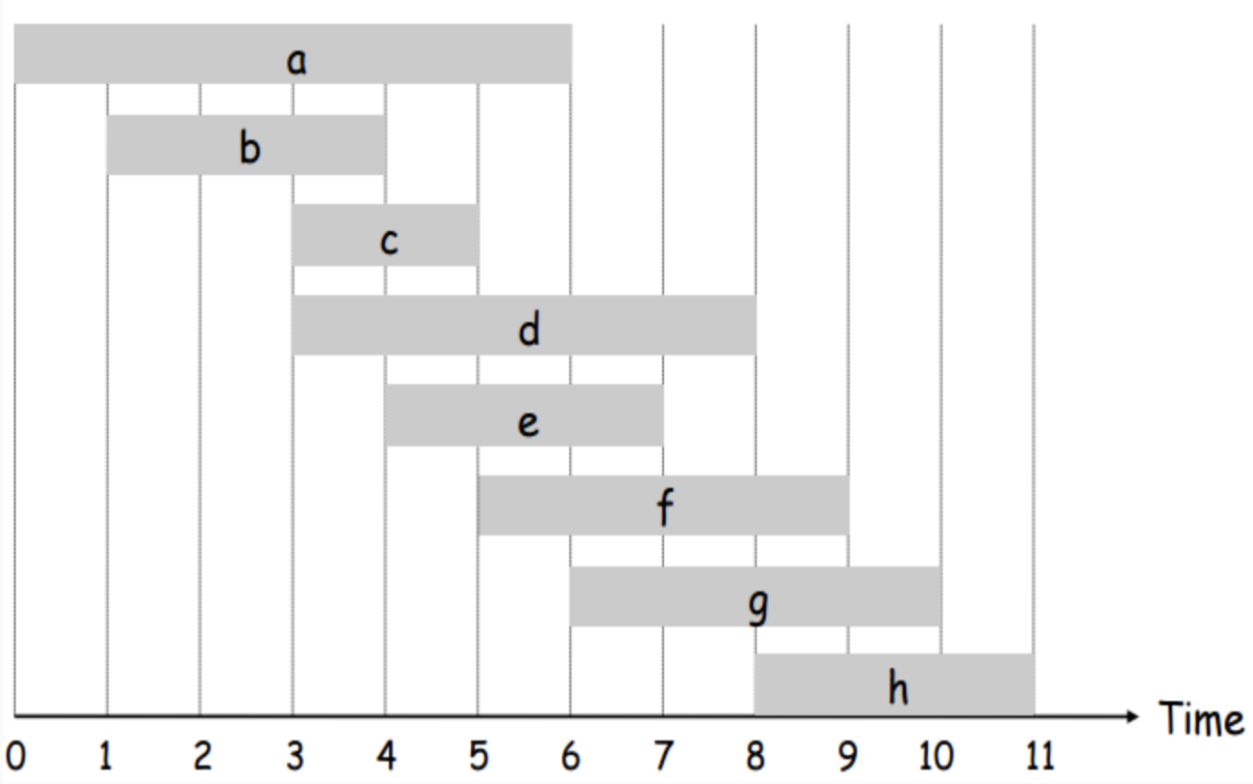
Interval Partitioning Problem

Room / Time	A	B	C	D	E	F	G	H
Paral.#1 11:00-12:30 Monday 16 Sept.	**Block 1A.** Pedagogic methods: Studio teaching (4-1)	**Block 1B.** Digital technology in landscape education (2-1)	**Block 1C.** Curricula: Assessment and programme development	**Block 1D.** Teaching transdisciplinary approaches to landscape (4-1)	**Block 1E.** History of landscape education (3-1)	**Block 1F.** The ELC and landscape education	**Block 1G.** Pedagogic methods: Multisensory	**Block 1H.** [Special session] Landscape architecture education in a global research context
Paral.#2 13:30-15:00 Monday 16 Sept.	**Block 2A.** Pedagogic methods: Studio teaching (4-2)	**Block 2B.** Pedagogic methods: sustainability, ecology and planting design	**Block 2C.** Pedagogic methods: Understanding site	**Block 2D.** Teaching transdisciplinary approaches to landscape (4-2)	**Block 2E.** History of landscape education (3-2)	**Block 2F.** [Special session] UNISCAPE meeting: Landscape education after 20 years of the ELC	**Block 2G.** [Workshop] Stonesensing: Evoking meaning with stones	**Block 2H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-1)
Paral.#3 15:30-17:00 Monday 16 Sept.	**Block 3A.** Pedagogic methods: Studio teaching (4-3)	**Block 3B.** [Special session] The history and future of teaching digital methods in landscape architecture	**Block 3C.** The making of a profession	**Block 3D.** Teaching transdisciplinary approaches to landscape (4-3)	**Block 3E.** History of landscape education (3-3)	**Block 3F.** [Special session] Challenges and opportunities of landscape architecture education in the Arab world: The experience of the American University of Beirut	**Block 3G.** [Workshop] Learning to read the landscape: a methodological framework	**Block 3H.** [Workshop] New practices of collaboration: Exploring landscape architectural teaching, learning and practice contexts (2-2)
Paral.#4 10:30-12:00 Tuesday 17 Sept.	**Block 4A.** Pedagogic methods: Studio teaching (4-4)	**Block 4B.** Pedagogic methods: Student engagement and motivation	**Block 4C.** Pedagogic methods: Fieldwork	**Block 4D.** Landscape education: Ethics and values	**Block 4E.** Pedagogic methods: Teaching in a global context	**Block 4F.** [Special session] Bridging national and disciplinary boundaries: Concepts of sustainability in landscape and urban planning education	**Block 4G.** [Special session] Professional mythologies or academic consistency? Reframing the basic concepts in landscape architecture education	**Block 4H.** [Workshop] An asset to education: Introducing archives of landscape architecture in academic education
Paral.#5 14:00-15:30 Tuesday 17 Sept.	**Block 5A.** Pedagogic methods: Design thinking	**Block 5B.** Digital technology in landscape education (2-2)	**Block 5C.** Educating in a multicultural context	**Block 5D.** Teaching transdisciplinary approaches to landscape (4-4)	**Block 5E.** Pedagogic methods: Integrating theory	**Block 5F.** Visions for landscape education	**Block 5G.** [Workshop] The power of imagined landscapes	

Note

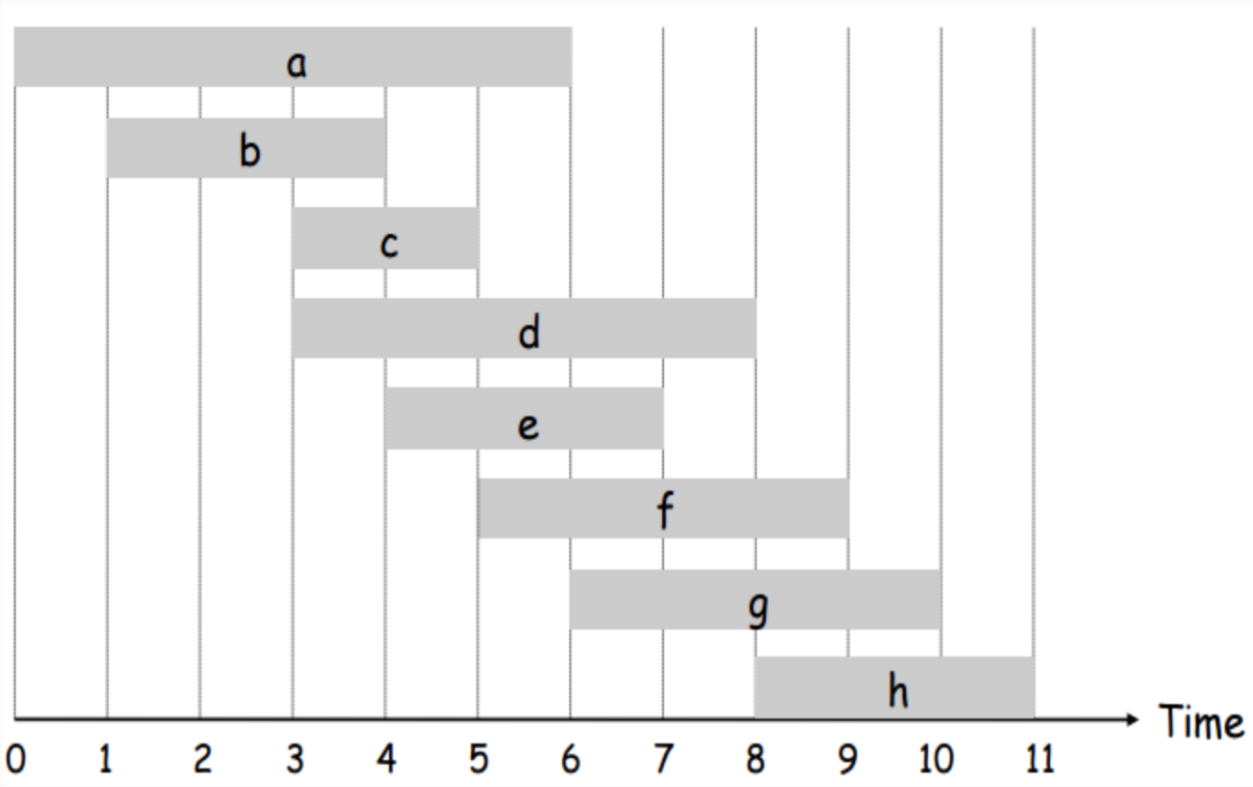
Notice that we need to ensure that no two overlapping workshops are scheduled in the same room.

Interval Partitioning Problem



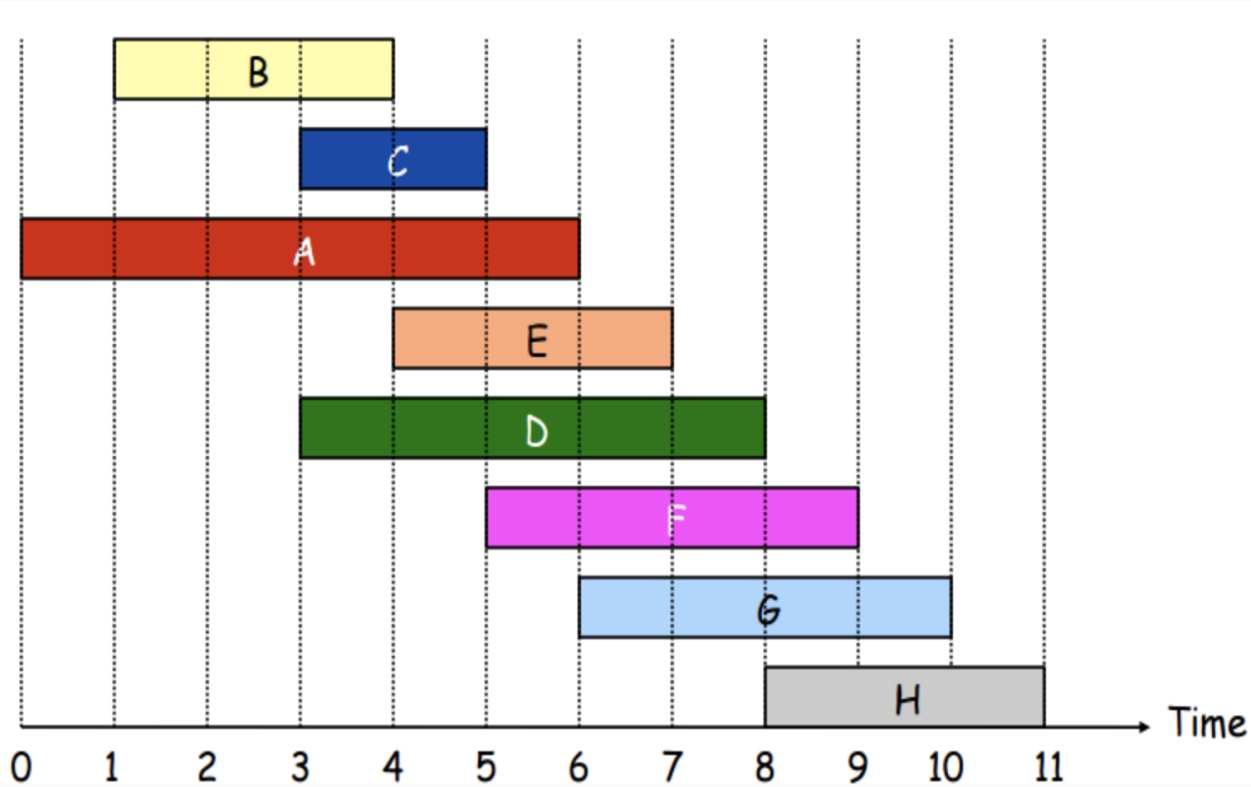
- **Input**: Set $(s(i), f(i)), 1 \leq i \leq n$ of start and finish times of n workshops.
- **Solution**: The minimum number of rooms required to hold all workshops.

Interval Partitioning Problem



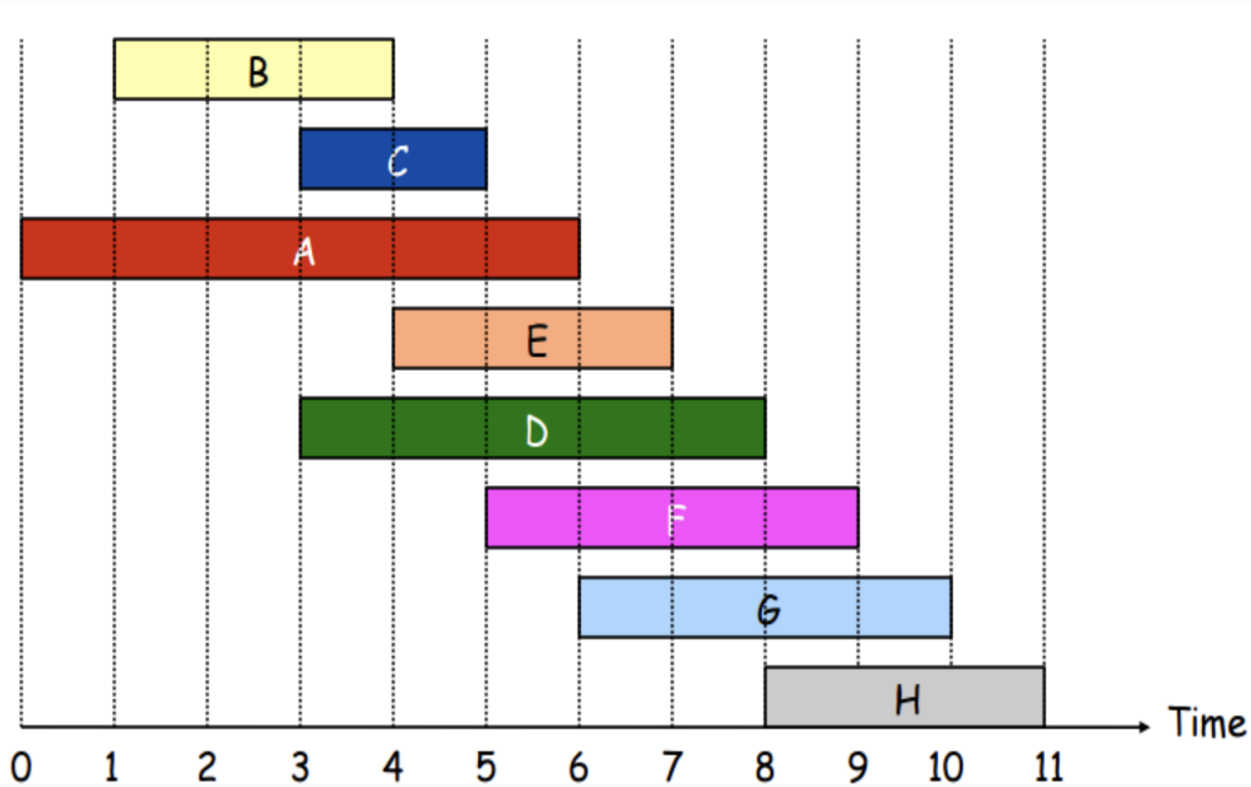
- Two workshops need separate rooms if they overlap.
- This problem models the situation where you have a set of workshops and you want to minimize the number of rooms required.
- For any input set of workshops, our algorithm must provably compute the minimum number of rooms required.

Example: Room Assignment



Poll 6

Example: Room Assignment



Poll 6

- How many rooms are needed for the following workshops?
  - A, B and D : **3 room**
  - B, C and D : **3 room**
  - A, G and H : **2 room**
  - C, F and H : **2 rooms**
  - C, E and H : **2 room**
  - B, E and H : **1 rooms**

Interval Partitioning Problem Greedy Algorithm: Idea

- Sort the workshops by their start times.
- Use a priority queue to manage end times of currently assigned rooms.
- For each workshop:
  - If the workshop can reuse an existing room (its start time is after the earliest end time), assign it to that room.
  - Otherwise, allocate a new room.

min-heap Data Structure

- A **min-heap** is a binary tree-based data structure where the parent node is always less than or equal to its child nodes.
- This property ensures that the smallest element is always at the root of the heap.
- Efficient operations:
  - Retrieving the minimum element: $O(1)$ time.
  - Insertion and deletion of the minimum element: $O(\log n)$ time.

Interval Partitioning Problem Greedy Algorithm: Pseudocode

Algorithm 3 Assign rooms using a priority queue

```
function intervalPartitioning(S):
    endTimes = [] // min-heap based priority queue sorted by end times
    sort(S) // Sort intervals by start time
    for each workshop in S:
        if endTimes is not empty and endTimes[0] ≤ start(workshop):
            room = endTimes.heappop(endTimes) // Reuse room
        else:
            room = new room(workshop) // create new room
        endTimes.heappush(endTimes)
    return len(endTimes)
```

Question: In this algorithm we are only checking for the first `endTimes`. why is that ? Poll 7

Interval Partitioning Problem Greedy Algorithm: Pseudocode

Algorithm 4 Assign rooms using a priority queue

```
function intervalPartitioning(S):
    endTimes = [] // min-heap based priority queue sorted by end times
    sort(S) // Sort intervals by start time
    for each workshop in S:
        if endTimes is not empty and endTimes[0] ≤ start(workshop):
            room = endTimes.heappop(endTimes) // Reuse room
        else:
            room = new room(workshop) // create new room
        endTimes.heappush(endTimes)
    return len(endTimes)
```

Claim: The size of the priority queue at the end of the algorithm gives the minimum number of rooms required.

Interval Partitioning Algorithm: Greedy Property

Note

The greedy property of the algorithm ensures that we are always minimizing the number of rooms used by reusing rooms whenever possible.

Interval Partitioning Greedy Algorithm: Runtime Analysis

- Sorting the intervals by their start times takes $O(n \log n)$ time.
  - For each interval, we perform the following operations:
    - Checking the earliest end time in the priority queue (min-heap) takes $O(1)$ time.
    - Adding a new end time to the priority queue (min-heap) takes $O(\log k)$ time, where k is the number of rooms.
    - Removing the earliest end time from the priority queue (min-heap) takes $O(\log k)$ time, where k is the number of rooms.
  - Thus, processing each interval takes $O(\log k)$ time.
- **Total runtime**
  - Sorting the intervals: $O(n \log n)$
  - Processing each interval: $O(n \log k)$
  - In the worst case, $k = n$ (all intervals need separate rooms), so the processing time is $O(n \log n)$.
  - Therefore, the total runtime of the algorithm is $O(n \log n) + O(n \log n) = O(n \log n)$.

Minimizing Lateness

Scheduling to Minimise Lateness

- Job i has a length $t(i)$ and a deadline $d(i)$.
- We want to schedule all n jobs on one resource.
- Our goal is to assign a starting time s(i) to each job such that each job is delayed as little as possible.
- A job i is late if $f(i) > d(i)$
  - Notice $f(i)$ is not a given, it depends on how we decide to schedule a job
- The **lateness** of the job is

$$\max(0, f(i) - d(i))$$

- The **lateness of a schedule** is max

$$\max_{1 \leq i \leq n} (\max(0, f(i) - d(i)))$$

  - Note that the lateness is defined with a **max** not a **sum**

Scheduling to Minimise Lateness: Example

- Example 1:

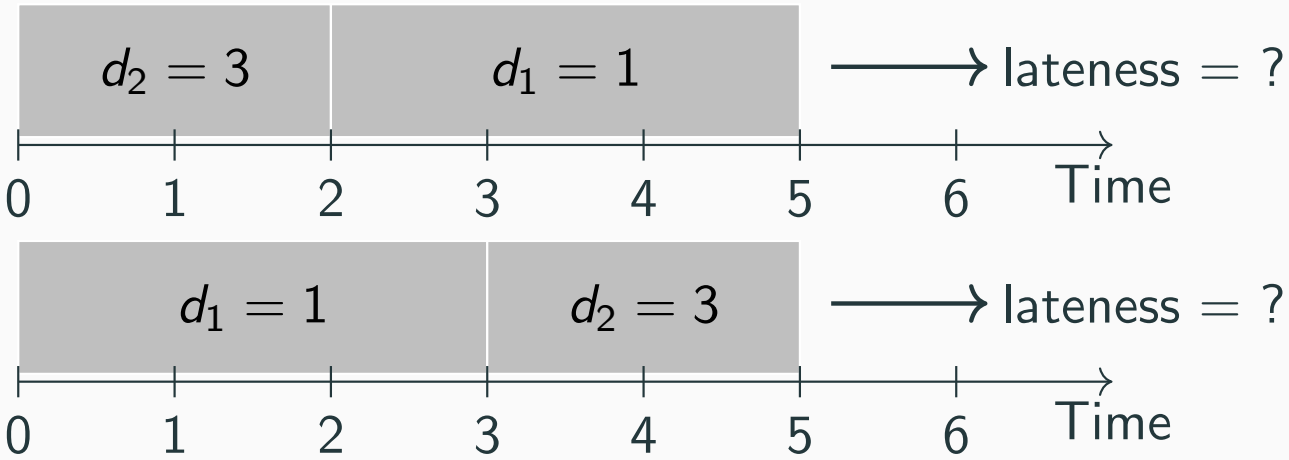
i	**1**	**2**
t_i	3	2
d_i	1	3

Scheduling to Minimise Lateness: Example

- Example 1:

i	1	2
t_i	3	2
d_i	1	3

- Which of the following is better Poll 8

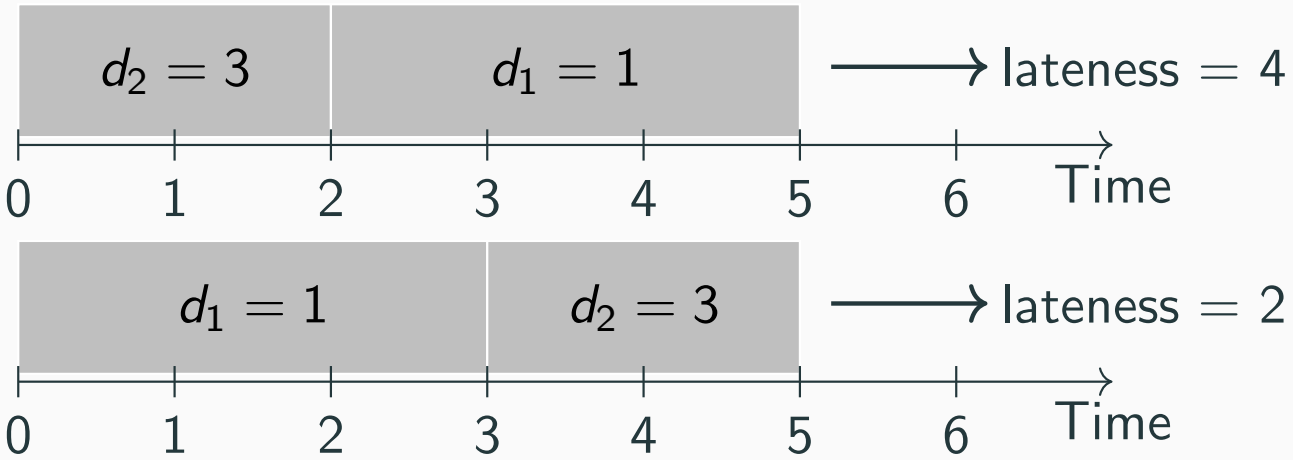


Scheduling to Minimise Lateness: Example

- Example 1:

i	1	2
t_i	3	2
d_i	1	3

- Which of the following is better Poll 8



Scheduling to Minimise Lateness: Example

	1	2	3	4	5	6	
t_j	3	2	1	4	3	2	← time required
d_j	6	8	9	9	14	15	← deadline

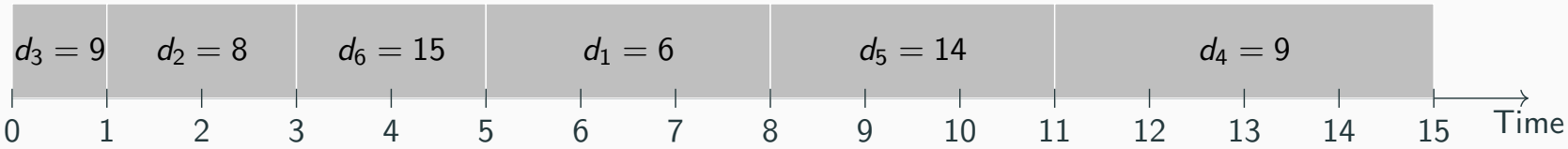
← job number

Scheduling to Minimise Lateness: Example

	1	2	3	4	5	6	
t_j	3	2	1	4	3	2	← time required
d_j	6	8	9	9	14	15	← deadline

← job number

- What's the delay for the following scheduling? Poll 9

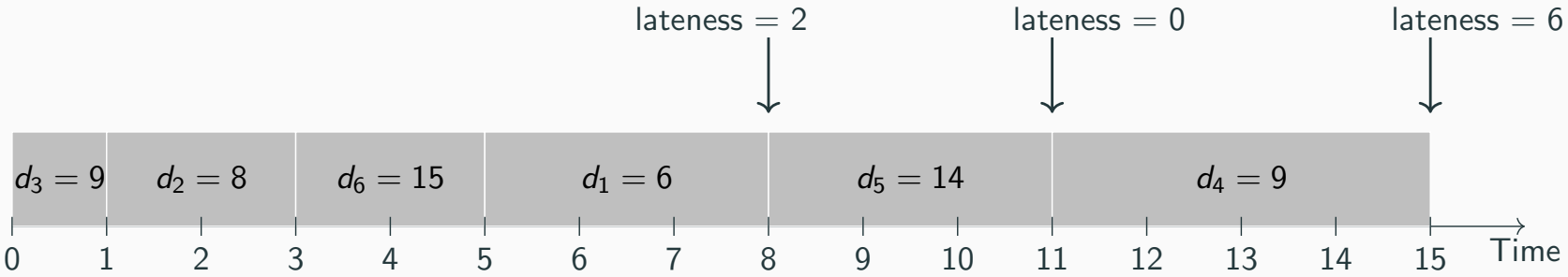


Scheduling to Minimise Lateness: Example

	1	2	3	4	5	6	
t_j	3	2	1	4	3	2	← time required
d_j	6	8	9	9	14	15	← deadline

← job number

- What's the delay for the following scheduling? Poll 9



Scheduling to Minimise Lateness: Problem

- **Input**: Set $(t(i), d(i)), 1 \leq i \leq n$ of length and deadline of n tasks.
- **Solution**: Set $(s(i), 1 \leq i \leq n$ of start times such that $\max_{1 \leq i \leq n}(\max(0, s(i) + t(i) - d(i)))$ is as small as possible

Minimizing Lateness: Brute Force Approach

- Generate all possible permutations of jobs.
- Calculate the maximum lateness for each permutation.
- Choose the permutation with the smallest maximum lateness.

Minimizing Lateness: Brute Force Algorithm

Algorithm 5 Minimize Lateness using Brute Force

```
function bruteForceMinimizeLateness(jobs):
    n = len(jobs)
    bestSchedule = None
    minLateness = ∞
foreach permutation in permutations(jobs):
    currentTime, maxLateness = 0
    for job in permutation:
        currentTime += duration(job)
        lateness = max(0, currentTime - deadline(job))
        maxLateness = max(maxLateness, lateness)
    if maxLateness < minLateness:
        minLateness = maxLateness
        bestSchedule = permutation
return bestSchedule
```

Minimizing Lateness: Brute Force Algorithm Time Complexity

- Generating Permutations:
  - There are $n!$ (factorial) permutations of n jobs.
- Evaluating Each Permutation:
  - For each permutation, compute the maximum lateness by iterating over n jobs.
  - Each evaluation takes $O(n)$ time.
- Overall Time Complexity:
  - Generating permutations: $O(n!)$
  - Evaluating each permutation: $O(n)$
  - Combined time complexity: $O(n \cdot n!)$
- The brute force algorithm is computationally expensive and impractical for large input sizes due to the factorial growth rate.

Scheduling to Minimise Lateness: Sort the jobs

- **Key question**: In what order should we schedule the jobs? Poll 10

Scheduling to Minimise Lateness: Sort the jobs

- **Key question**: In what order should we schedule the jobs? Poll 10
  - **Shortest length** Increasing order of length $t(i)$.

Scheduling to Minimise Lateness: Sort the jobs

- **Key question**: In what order should we schedule the jobs? Poll 10
  - **Shortest length** Increasing order of length $t(i)$.
    - Shortest job may have a very late deadline.

i	1	2
$t(i)$	1	10
$d(i)$	100	10

Scheduling to Minimise Lateness: Sort the jobs

- **Key question**: In what order should we schedule the jobs? Poll 10
  - **Shortest length** Increasing order of length $t(i)$.
    - Shortest job may have a very late deadline.

i	1	2
$t(i)$	1	10
$d(i)$	100	10

  - **Shortest slack time** Increasing order of $d(i) - t(i)$.

Scheduling to Minimise Lateness: Sort the jobs

- **Key question**: In what order should we schedule the jobs? Poll 10
  - **Shortest length** Increasing order of length $t(i)$.
    - Shortest job may have a very late deadline.

i	1	2
$t(i)$	1	10
$d(i)$	100	10

  - **Shortest slack time** Increasing order of $d(i) - t(i)$.
    - Job with smallest slack may take a long time.

i	1	2
$t(i)$	1	10
$d(i)$	2	10

Scheduling to Minimise Lateness: Sort the jobs

- **Key question**: In what order should we schedule the jobs? Poll 10
  - **Shortest length** Increasing order of length $t(i)$.
    - Shortest job may have a very late deadline.

i	1	2
$t(i)$	1	10
$d(i)$	100	10

  - **Shortest slack time** Increasing order of $d(i) - t(i)$.
    - Job with smallest slack may take a long time.

i	1	2
$t(i)$	1	10
$d(i)$	2	10

  - **Earliest deadline** Increasing order of deadline $d(i)$.

Scheduling to Minimise Lateness: Sort the jobs

- **Key question**: In what order should we schedule the jobs? Poll 10
  - **Shortest length** Increasing order of length $t(i)$.
    - Shortest job may have a very late deadline.

i	1	2
$t(i)$	1	10
$d(i)$	100	10

  - **Shortest slack time** Increasing order of $d(i) - t(i)$.
    - Job with smallest slack may take a long time.

i	1	2
$t(i)$	1	10
$d(i)$	2	10

  - **Earliest deadline** Increasing order of deadline $d(i)$.
    - Does it make sense to tackle jobs with earliest deadlines first? Poll 11

Minimizing Lateness: Earliest Deadline First

Algorithm 6 Minimize Lateness using Earliest Deadline First (EDF)

```
function minimizeLateness(jobs):
    A = []
    sort(jobs) // Sort jobs based on deadline
    currentTime = 0
    lateness = 0
    for job in jobs:
        A.append(job)
        currentTime += job.duration
        if lateness ¡ currentTime - job.duration:
            lateness = currentTime - job.duration
    return A, lateness
```

Proof of correctness

- We can use loop invariant
- Once sorted, executing the tasks in order requires $O(n)$ time.
- Therefore, the overall runtime of the EDF algorithm is dominated by the sorting step, making it $O(n \log n)$.

Runtime Analysis

- Sorting the tasks by their deadlines takes $O(n \log n)$ time.
- Once sorted, executing the tasks in order requires $O(n)$ time.
- Therefore, the overall runtime of the EDF algorithm is dominated by the sorting step, making it $O(n \log n)$.

Fractional Knapsack

Fractional Knapsack Problem

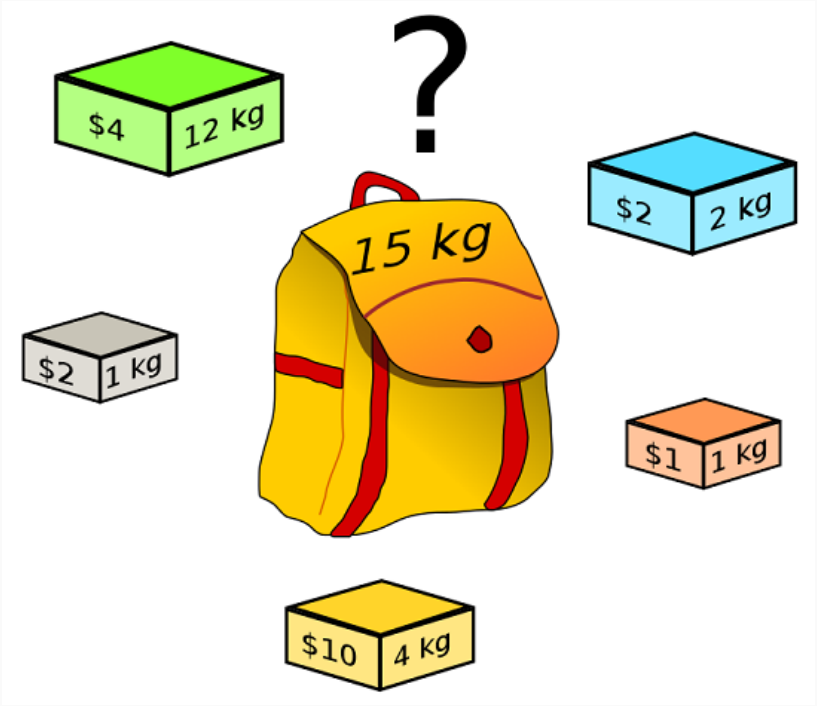


Image Credit: https://www.hackerearth.com/practice/notes/the-knapsack-problem/

- **Input:** Knapsack capacity, items with their quantities and values.
- **Goal:** Maximize value without exceeding capacity.
- **Constraints:**
  - Items can be taken in fractions to obtain proportional value.

Fractional Knapsack Problem

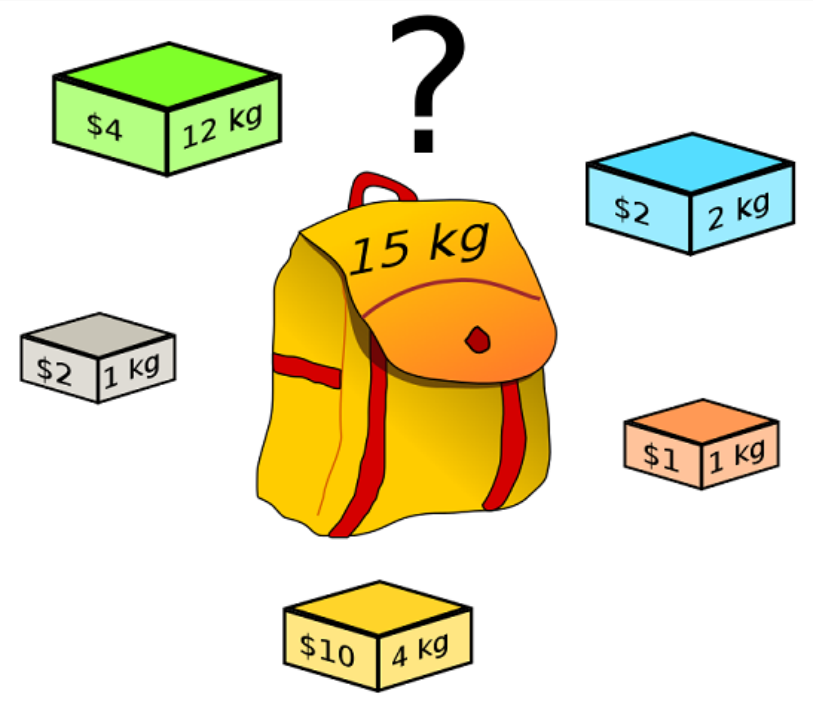
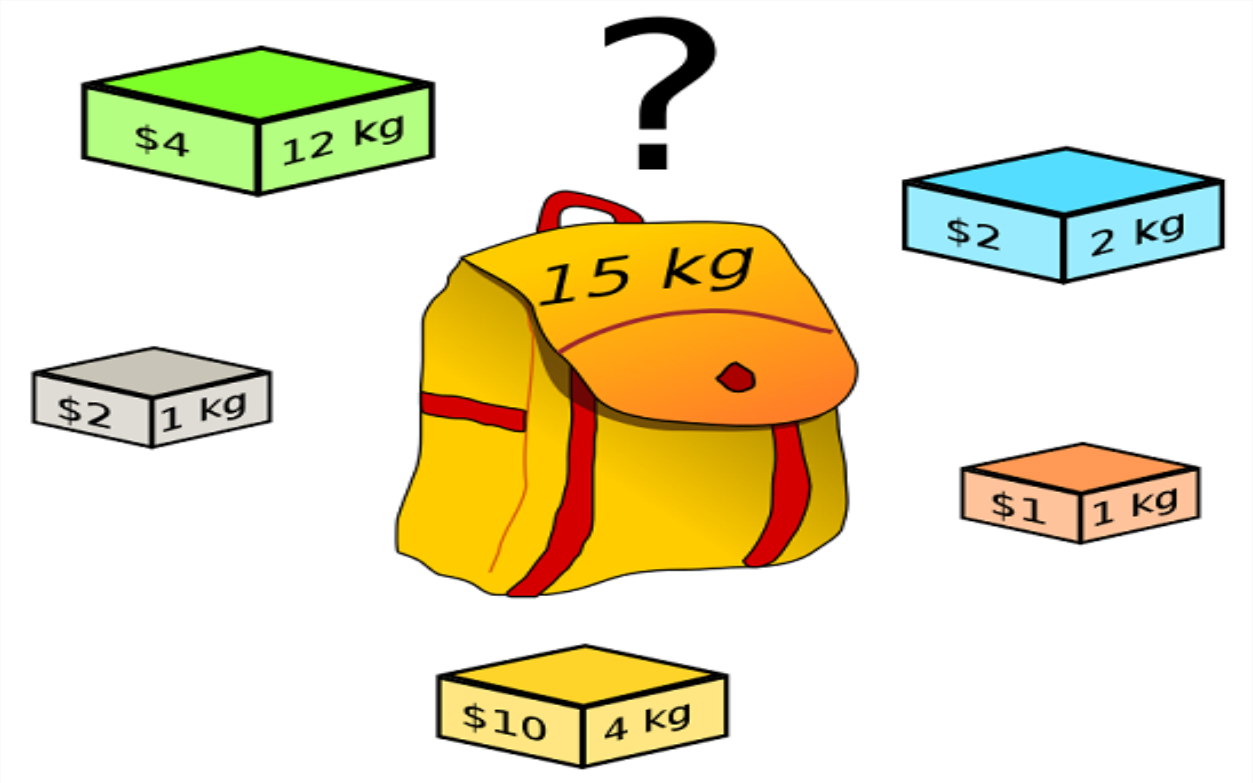


Image Credit: https://www.hackerearth.com/practice/notes/the-knapsack-problem/

Question

Which of these items should I take to maximize value?

Fractional Knapsack Problem



- **Input**: List $(w(i), v(i)), 1 \leq i \leq n,$ and C where C is the size of our knapsack, $w(i)$ is the size of item i and $v(i)$ is its value
- **Output**: The maximum value we can get without exceeding C

Fractional Knapsack: Greedy Property

- What should consider to be the greedy property
- When deciding to take an item or not Poll 12

Fractional Knapsack: Greedy Property

- What should consider to be the greedy property
- When deciding to take an item or not Poll 12
  - Take the item with the smallest size
  - Take the item with the maximum value
    - What if the item with the maximum value is more than the remaining space in our knapsack?

Fractional Knapsack: Greedy Property

- What should consider to be the greedy property
- When deciding to take an item or not Poll 12
  - Take the item with the smallest size
  - Take the item with the maximum value
    - What if the item with the maximum value is more than the remaining space in our knapsack?
- How to sort the items Poll 13

Fractional Knapsack: Greedy Property

- What should consider to be the greedy property
- When deciding to take an item or not Poll 12
  - Take the item with the smallest size
  - Take the item with the maximum value
    - What if the item with the maximum value is more than the remaining space in our knapsack?
- How to sort the items Poll 13
  - By size
  - By value
  - By value per unit of size

Fractional Knapsack Algorithm: Pseudocode

Algorithm 7 Fractional Knapsack

Require: Items with weights w_i and values v_i, knapsack capacity W

Ensure: Maximum total value in the knapsack

1: Calculate the value-to-weight ratio $r_i = \frac{v_i}{w_i}$ for each item i
2: Sort items by r_i in descending order
3: Initialize total value $V \leftarrow 0$
4: Initialize remaining capacity $C \leftarrow W$
5: **for** each item i in sorted order **do**
6: **if** $w_i \leq C$ **then**
7: Take the whole item i
8: $C \leftarrow C - w_i$
9: $V \leftarrow V + v_i$
10: **else**
11: Take fraction $\frac{C}{w_i}$ of item i
12: $V \leftarrow V + v_i \times \frac{C}{w_i}$
13: Break the loop
14: **end if**

Fractional Knapsack: Proof of Correctness

- **Loop Invariant**: At the start of each iteration i of the loop, the total value V and the remaining capacity C represent an optimal solution for the subset of items considered so far.
- **Initialization**:
  - Before the first iteration, no items have been considered.
  - Total value V is initialized to 0.
  - Remaining capacity C is initialized to the knapsack capacity W.
  - The loop invariant holds trivially since no items have been added yet.

Fractional Knapsack: Proof of Correctness

- **Maintenance**:
  - Assume the loop invariant holds before the i^{th} iteration.
  - Consider item i with weight w_i and value v_i.
  - If $w_i \leq C$, the entire item is added to the knapsack.
  - The value v_i is added to V, and w_i is subtracted from C.
  - If $w_i > C$, a fraction $\frac{C}{w_i}$ of the item is added.
  - The value $v_i \times \frac{C}{w_i}$ is added to V, and C becomes 0.
  - In both cases, the updated V and C represent an optimal solution for the considered items.
  - The loop invariant is maintained.
- **Termination**:
  - The loop terminates when all items have been considered or the knapsack is full.
  - At this point, the total value V and the remaining capacity C represent an optimal solution.
  - The loop invariant guarantees that the solution is optimal.
  - Therefore, the fractional knapsack algorithm correctly computes the maximum value.

Fractional Knapsack: Time Complexity Analysis

- The time complexity of the fractional knapsack algorithm is determined by the sorting step and the loop.
- **Sorting**:
  - The items are sorted by their value-to-weight ratio r_i in descending order.
  - This step takes $O(n \log n)$ time, where n is the number of items.
- **Loop**:
  - The loop iterates over each item, processing it in constant time $O(1)$.
  - Therefore, the loop takes $O(n)$ time.
- **Overall Time Complexity**:
  - The total time complexity of the algorithm is dominated by the sorting step.
  - Thus, the overall time complexity is $O(n \log n)$.

Graph Recap

Definitions - Graphs

- **Graph**: A collection of nodes vertices (V) and edges (E) connecting pairs of nodes.
- **Directed vs Undirected**
  - **Directed Graph**: Edges have a direction, represented as ordered pairs (u, v) indicating the path from vertex u to vertex v.
  - **Undirected Graph**: Edges do not have a direction, represented as unordered pairs $\{u, v\}$, allowing movement between vertices u and v in both directions.
- **Subgraph**: A graph formed from a subset of the vertices and edges of another graph.
- **Degree of a Vertex**: The number of edges incident to the vertex. For a directed graph, there are **in-degree** and **out-degree**.

Definitions - Types of Graphs

- **Weighted Graph**: A graph where each edge (u, v) has a numerical value (weight) $w(u, v)$ associated with it.
- **Multi Graph**: A graph that can have multiple edges (parallel edges) between two vertices u and v.
- **Complete Graph**: A graph where every pair of distinct vertices u and v is connected by a unique edge.
- **Bipartite Graph**: A graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V.

Definitions - Paths

- **Path**: A sequence of edges connecting a sequence of vertices.
- **Simple Path**: A path that does not repeat any vertices.
- **Weight of Path**: The sum of the weights of the edges in a path. For a path $P = \{v_1, v_2, \ldots, v_k\}$, the weight $w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$.
- **Shortest Path**: The path between two vertices that has the smallest total weight or length.
- **Distance**: The length or weight of the shortest path between two vertices u and v, denoted as $d(u, v)$.
- **Connected Graph**: A graph in which there is a path between every pair of vertices.

Definitions - Trees

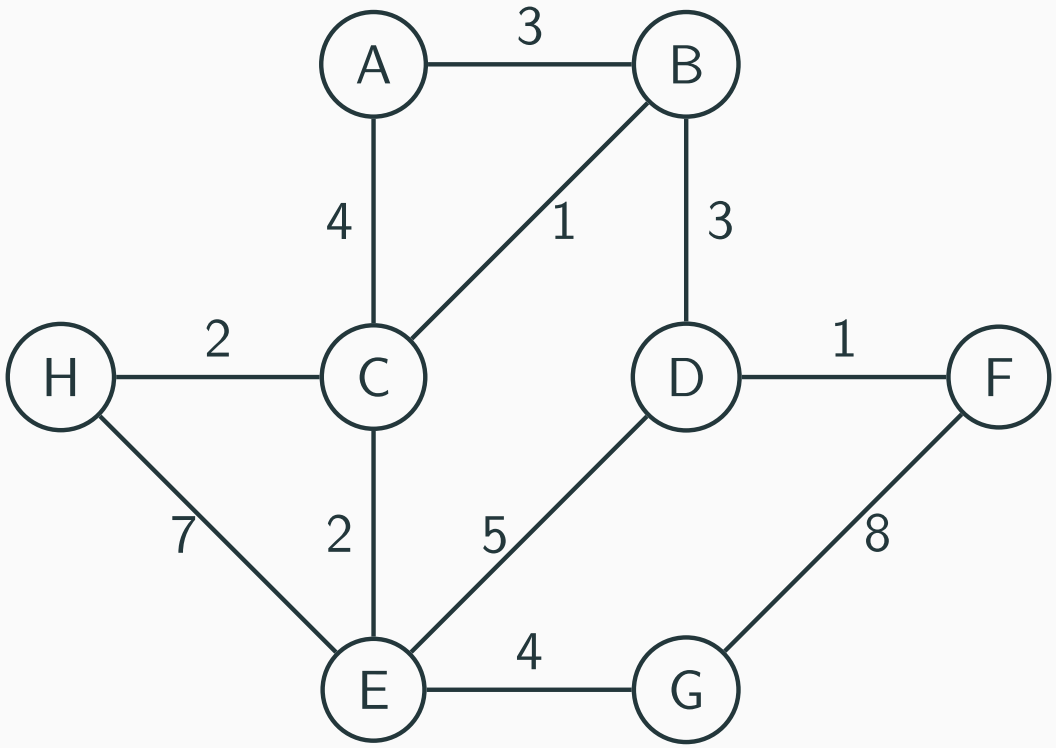
- **Cycle**: A path that starts and ends at the same vertex without repeating any edges or vertices (except the starting/ending vertex).
- **Tree**: A connected acyclic graph.
- **Spanning Tree**: A subset of the edges of a graph that forms a tree and connects all vertices of the graph.
  - A tree connecting all the vertices in a graph.
- **Minimum Spanning Tree**: A spanning tree with the minimum total weight of edges.

Greedy Algorithm

- A greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece
- It always chooses the next piece that offers the most immediate benefit
- Properties of Greedy Algorithms:
  - **Greedy Choice Property**:
    - A global optimum can be arrived at by selecting a local optimum.
    - This means making a choice that looks best at the moment.
  - **Optimal Substructure**:
    - A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems.

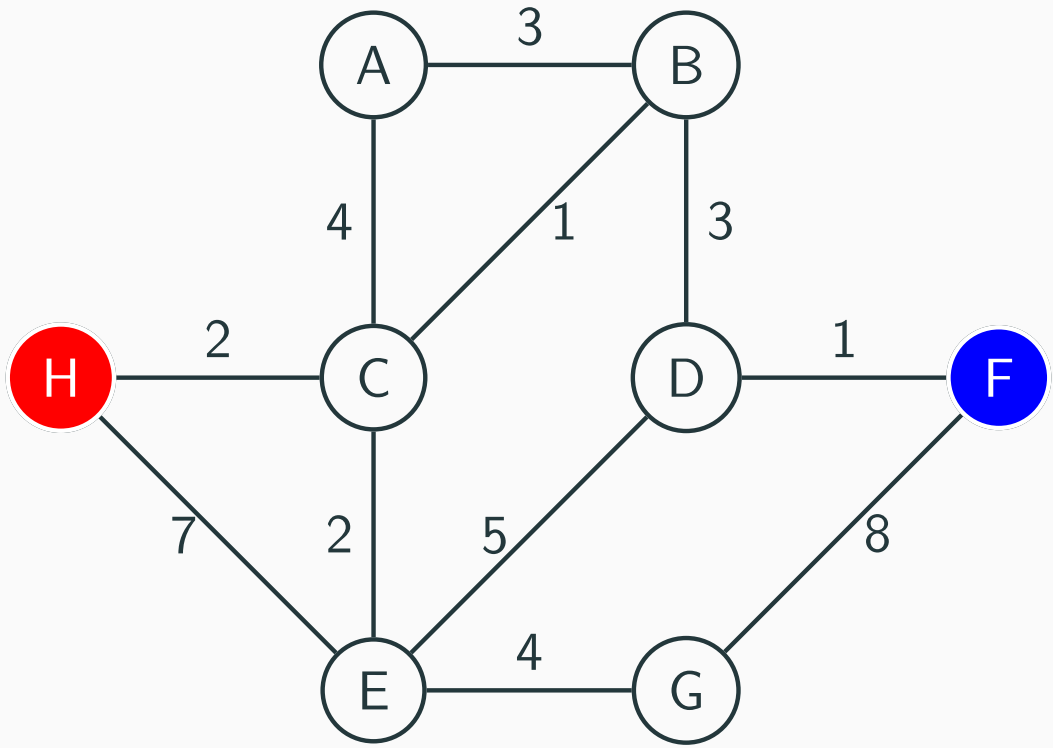
Single Source Shortest Path

Shortest Path: The problem



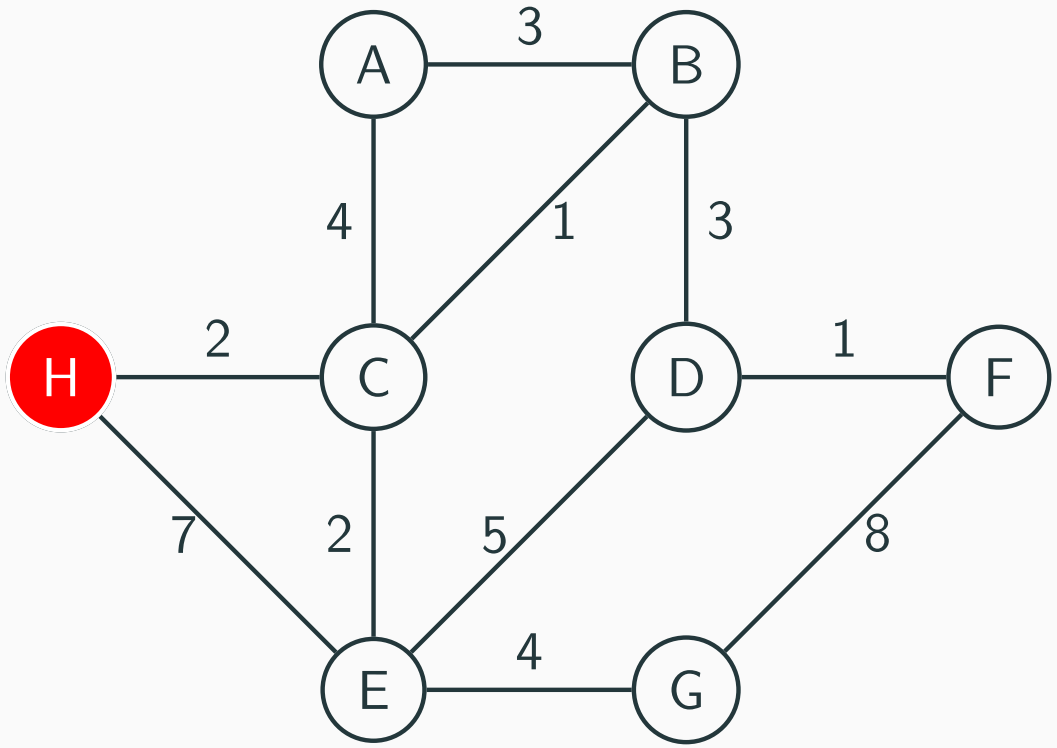
- Find the shortest path from a single source vertex to another vertex in a graph.

Shortest Path: The problem



- Find the shortest path from a single source vertex to another vertex in a graph.
  - E.g., Starting from H what is the shortest path to F

Shortest Path: The problem



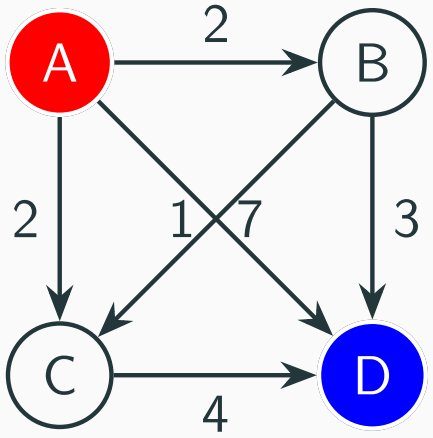
- Find the shortest path from a single source vertex to another vertex in a graph.
  - E.g., Starting from H what is the shortest path to F
  - Find the shortest path from a single source vertex to all other vertices in a graph.

Shortest Path: The Problem

- **Real-World Examples**:
  - Finding the quickest route between two locations using a GPS.
  - Optimizing data packet routing in a computer network.
  - Planning the shortest path for a delivery route.
- **Properties**:
  - **Non-Negative Weights**: Many shortest path algorithms assume non-negative edge weights.
  - **Optimal Substructure**: Shortest paths exhibit optimal substructure, meaning the shortest path between two vertices contains within it the shortest path between intermediate vertices.

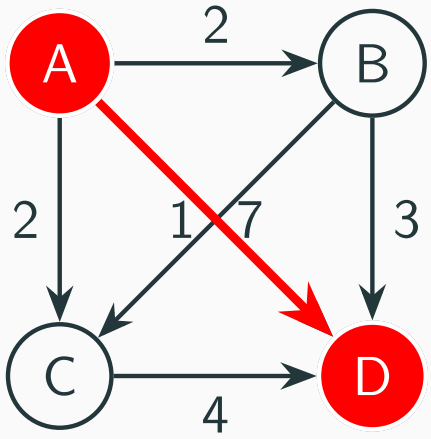
Shortest Path: Brute Force Approaches

- **Exhaustive Search**:
  - Generate all simple paths between two vertices.
  - Calculate the path weight for each path.
  - Pick the one with the lowest weight (shortest path).
- **Time Complexity**:
  - Brute force methods have exponential time complexity, typically $O(n!)$ for a graph with n vertices.
  - This makes brute force methods impractical for large graphs.
- Consider a simple graph:
  - Paths from A to D: Poll 1
    - A $\rightarrow$ D: Weight = ?
    - A $\rightarrow$ C $\rightarrow$ D: Weight = ?
    - A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D: Weight = ?
    - A $\rightarrow$ B $\rightarrow$ D: Weight = ?



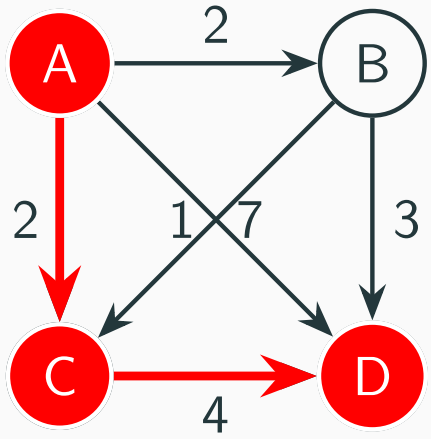
Shortest Path: Brute Force Approaches

- **Exhaustive Search**:
  - Generate all simple paths between two vertices.
  - Calculate the path weight for each path.
  - Pick the one with the lowest weight (shortest path).
- **Time Complexity**:
  - Brute force methods have exponential time complexity, typically $O(n!)$ for a graph with n vertices.
  - This makes brute force methods impractical for large graphs.
- Consider a simple graph:
  - Paths from A to D:
    - A $\rightarrow$ D: Weight $= 7$



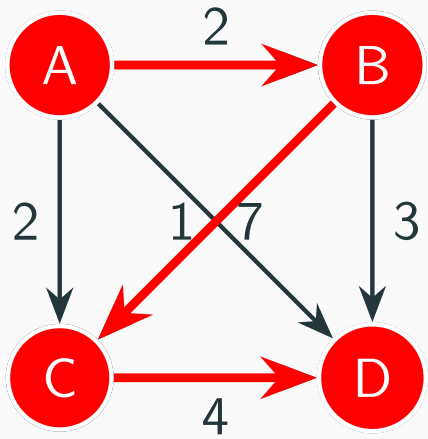
Shortest Path: Brute Force Approaches

- **Exhaustive Search**:
  - Generate all simple paths between two vertices.
  - Calculate the path weight for each path.
  - Pick the one with the lowest weight (shortest path).
- **Time Complexity**:
  - Brute force methods have exponential time complexity, typically $O(n!)$ for a graph with n vertices.
  - This makes brute force methods impractical for large graphs.
- Consider a simple graph:
  - Paths from A to D:
    - A $\rightarrow$ D: Weight $= 7$
    - A $\rightarrow$ C $\rightarrow$ D: Weight $= 6$



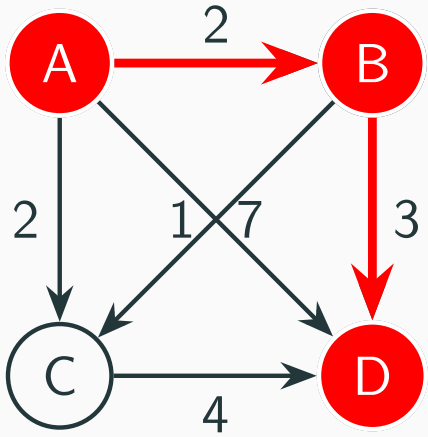
Shortest Path: Brute Force Approaches

- **Exhaustive Search**:
  - Generate all simple paths between two vertices.
  - Calculate the path weight for each path.
  - Pick the one with the lowest weight (shortest path).
- **Time Complexity**:
  - Brute force methods have exponential time complexity, typically $O(n!)$ for a graph with n vertices.
  - This makes brute force methods impractical for large graphs.
- Consider a simple graph:
  - Paths from A to D:
    - A $\rightarrow$ D: Weight $= 7$
    - A $\rightarrow$ C $\rightarrow$ D: Weight $= 6$
    - A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D: Weight $= 7$



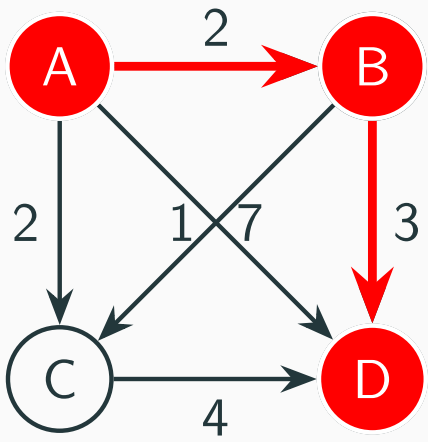
Shortest Path: Brute Force Approaches

- **Exhaustive Search**:
  - Generate all simple paths between two vertices.
  - Calculate the path weight for each path.
  - Pick the one with the lowest weight (shortest path).
- **Time Complexity**:
  - Brute force methods have exponential time complexity, typically $O(n!)$ for a graph with n vertices.
  - This makes brute force methods impractical for large graphs.
- Consider a simple graph:
  - Paths from A to D:
    - A $\rightarrow$ D: Weight $= 7$
    - A $\rightarrow$ C $\rightarrow$ D: Weight $= 6$
    - A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D: Weight $= 7$
    - A $\rightarrow$ B $\rightarrow$ D: Weight $= 5$



Shortest Path: Brute Force Approaches

- **Exhaustive Search**:
  - Generate all simple paths between two vertices.
  - Calculate the path weight for each path.
  - Pick the one with the lowest weight (shortest path).
- **Time Complexity**:
  - Brute force methods have exponential time complexity, typically $O(n!)$ for a graph with n vertices.
  - This makes brute force methods impractical for large graphs.
- Consider a simple graph:
  - Paths from A to D:
    - A $\rightarrow$ D: Weight $= 7$
    - A $\rightarrow$ C $\rightarrow$ D: Weight $= 6$
    - A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D: Weight $= 7$
    - A $\rightarrow$ B $\rightarrow$ D: Weight $= 5$
  - Shortest path is A $\rightarrow$ B $\rightarrow$ D with weight 5.



General Structure of Shortest Path Algorithms

- **Initialization**:
  - Set the initial distances from the source to all vertices as infinity, except for the source itself, which is set to 0.
  - Initialize the predecessor (or parent) for each vertex as undefined.
- **Relaxation**:
  - For each **edge** (u, v), if the distance to v through u is shorter than the current known distance to v, update the distance to v.
  - Repeat this process iteratively, ensuring that the shortest known distances are updated.
  - This step ensures that the shortest path estimates improve progressively.
- **Finalization**:
  - Once all vertices are processed, the shortest path from the source to each vertex is determined.
  - The final distances represent the shortest paths from the source to all other vertices.

Dijkstra Algorithm: Intuition

- At each step choose the edge (u, v) with the lowest weight
- This happens on the relaxation step
- This is a greedy choice as it is only making decision based on the currently observable distance
- By always choosing the vertex with the smallest known distance, Dijkstra's algorithm efficiently finds the shortest path.
- Only works for graphs with non-negative weights.

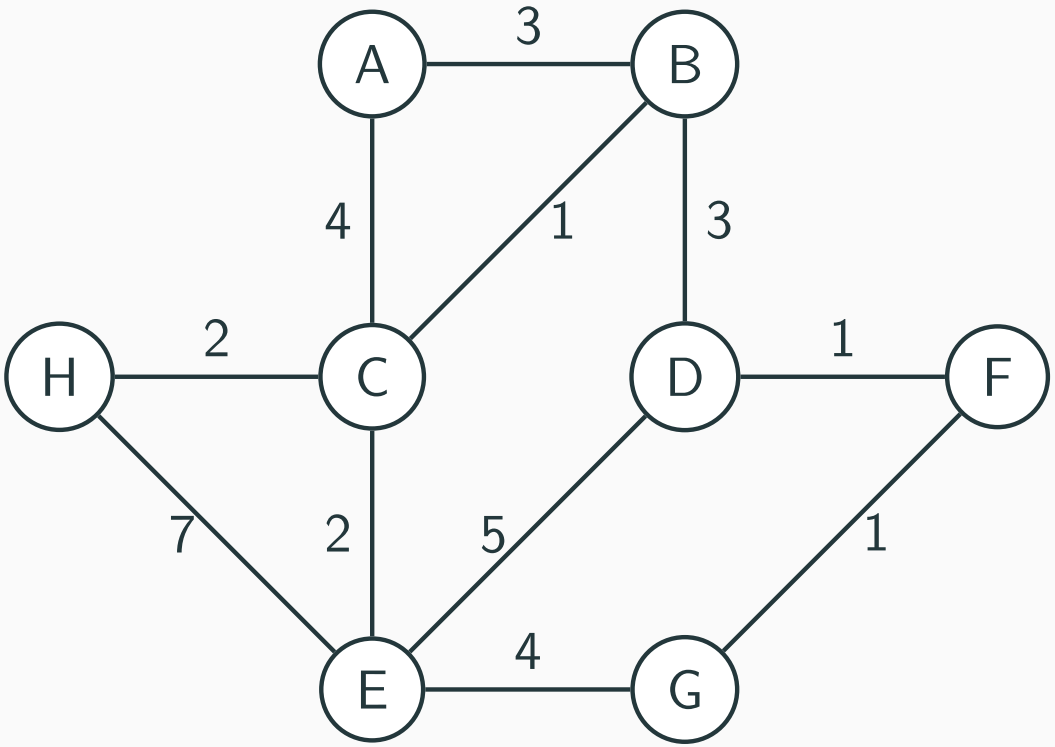
Dijkstra Algorithm: Pseudo-code

Algorithm 8 Dijkstra's Algorithm

1: **Input:** Graph $G = (V, E)$, source vertex s
2: **Output:** Shortest paths from s to all other vertices
3: Initialize distances $d[v] \leftarrow \infty$ for all $v \in V$ except $d[s] \leftarrow 0$
4: Initialize an empty priority queue Q
5: Insert s into Q with priority 0
6: **while** Q is not empty **do**
7: Extract vertex u with the smallest distance $d[u]$ from Q
8: **for** each neighbor v of u **do**
9: **if** $d[u] + w(u, v) < d[v]$ **then**
10: $d[v] \leftarrow d[u] + w(u, v)$
11: Insert v into Q with priority $d[v]$
12: **end if**
13: **end for**
14: **end while**

Dijkstra Algorithm: Example

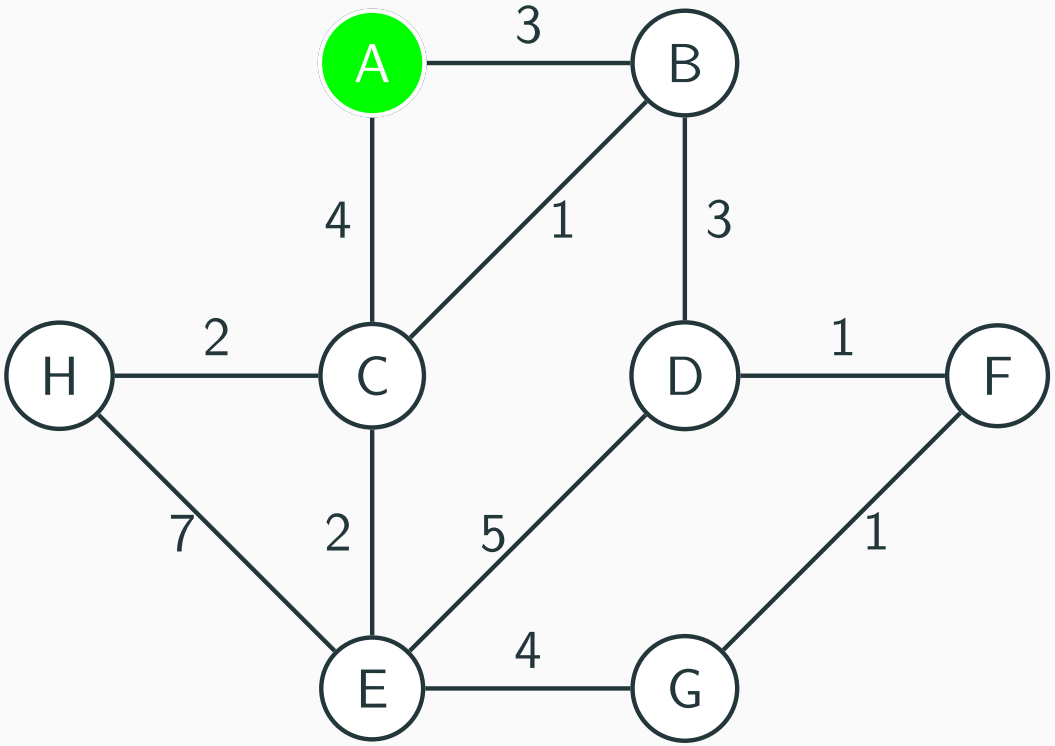
- Visit the unvisited vertex with he smallest known distance from the start vertex.



V	Distance	Parent
A		
B		
C		
D		
E		
F		
G		
H		

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - Start with the source vertex and set the distance to 0

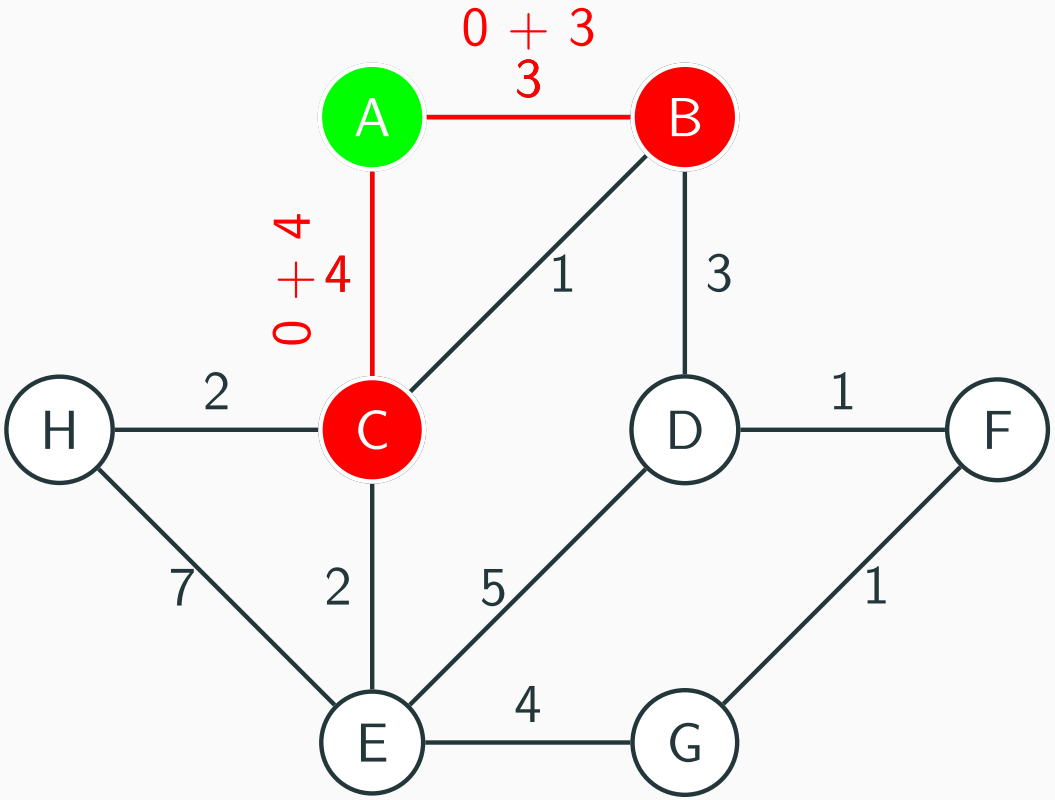


V	Distance	Parent
A	0	-
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Visited = [ ] Unvisited = [A, B, C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

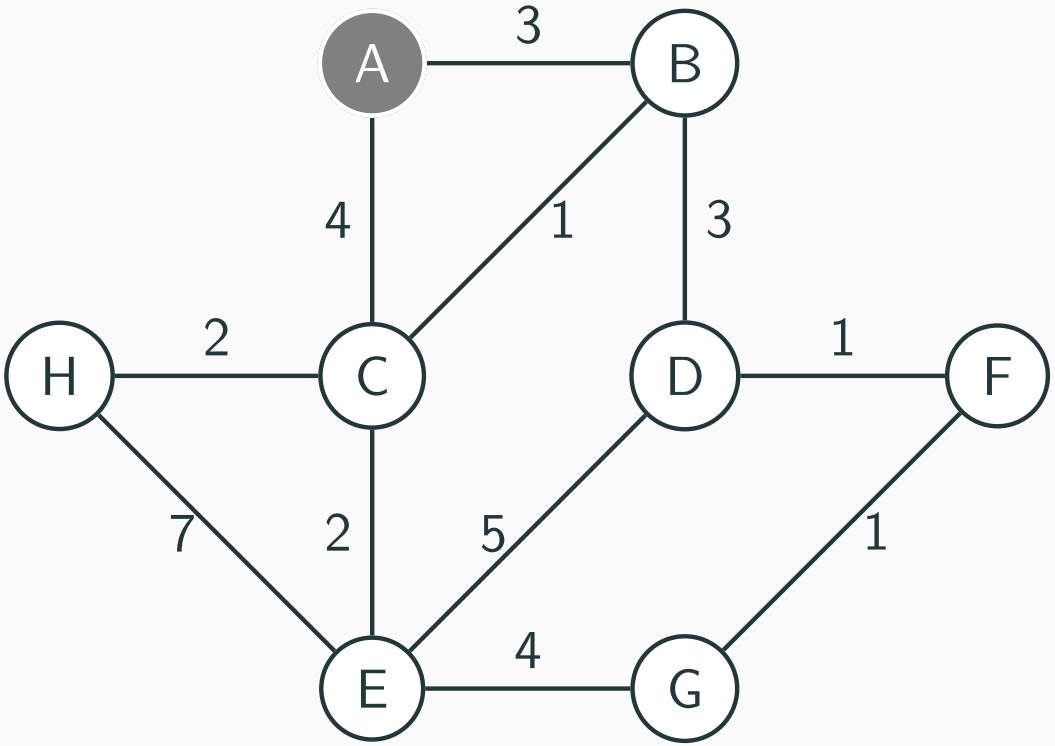


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Visited = [ ] Unvisited = [A, B, C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

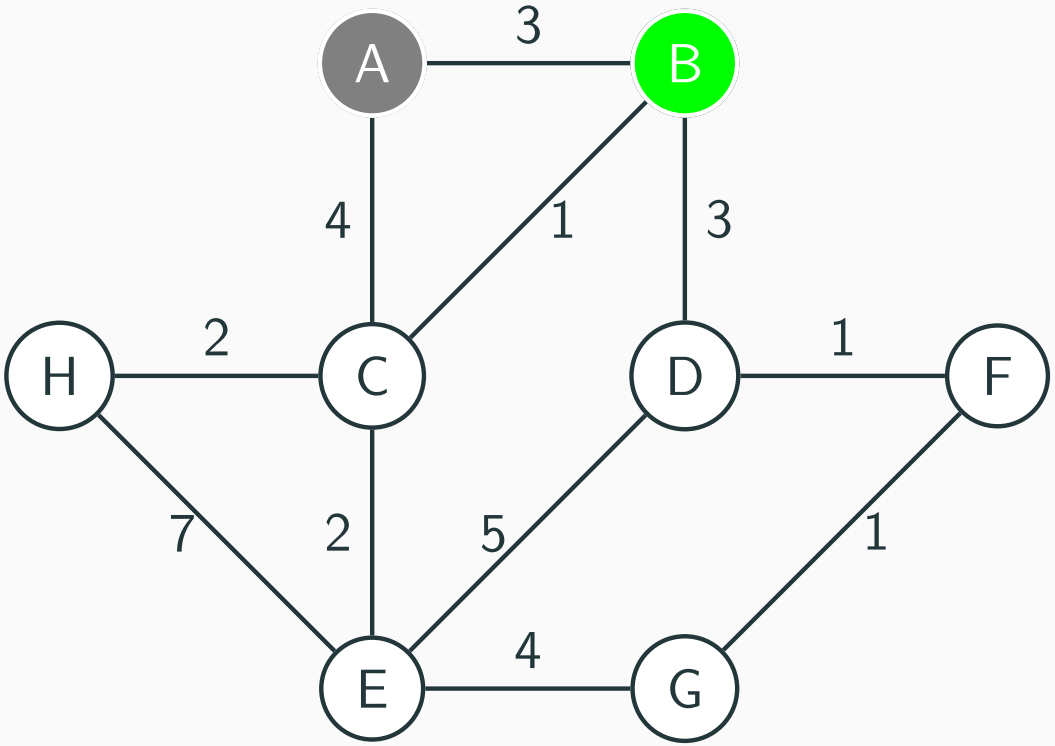


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Visited = [A] Unvisited = [B, C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

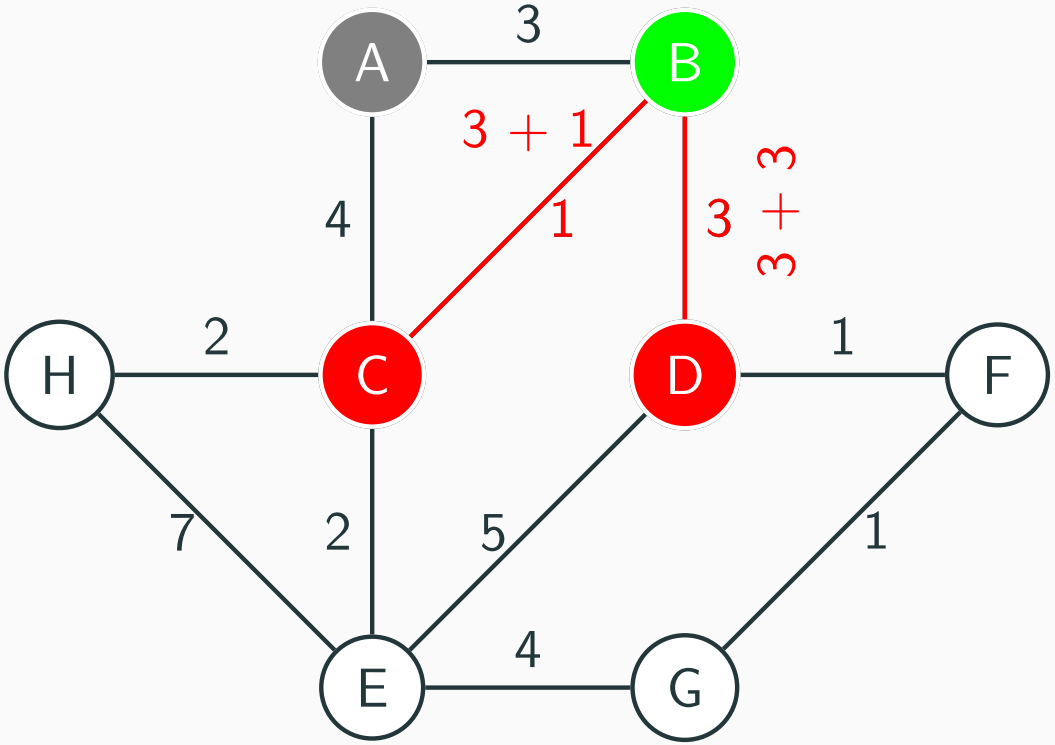


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Visited = [A] Unvisited = [B, C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

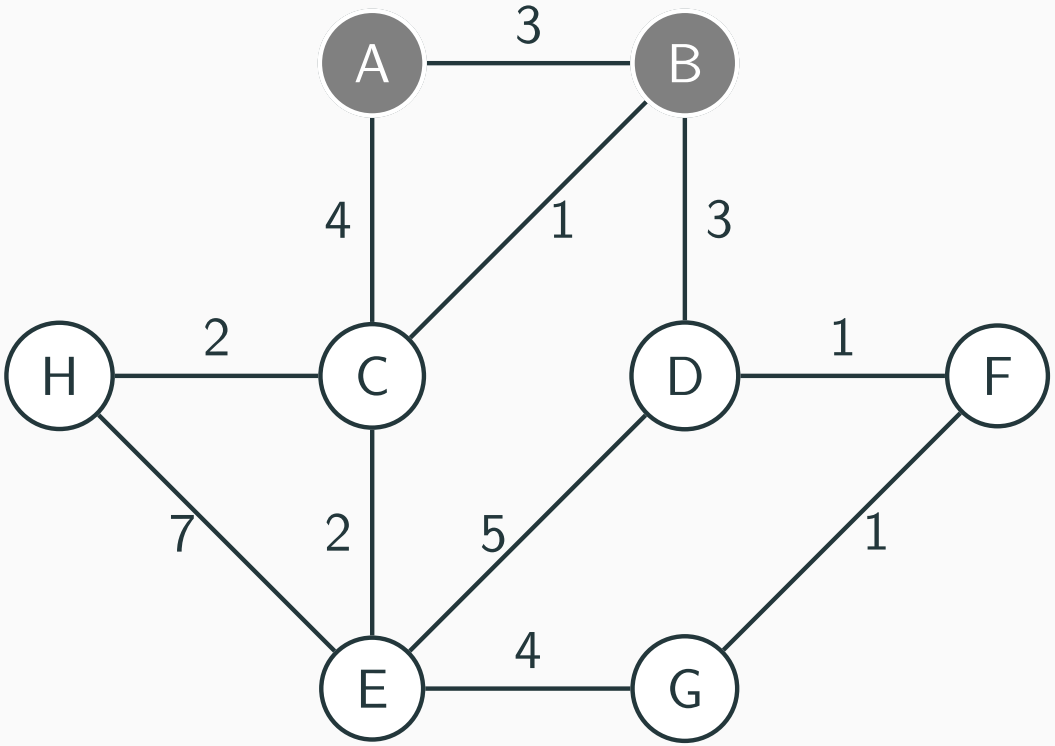


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	∞	
F	∞	
G	∞	
H	∞	

Visited = [A] Unvisited = [B, C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

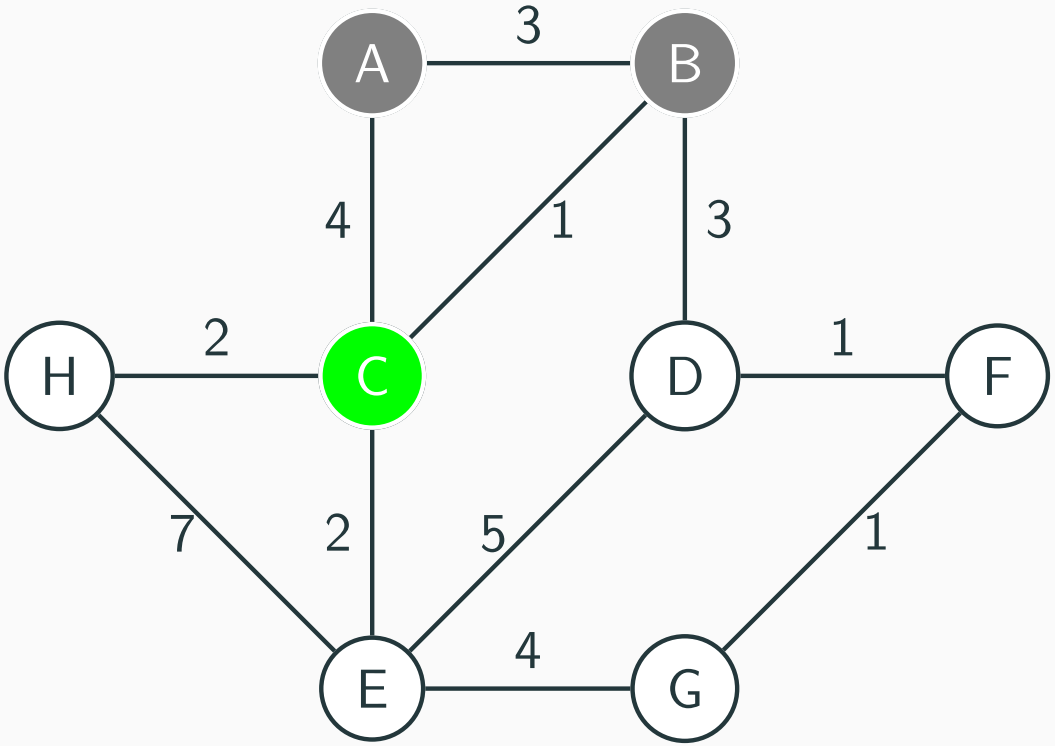


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	∞	
F	∞	
G	∞	
H	∞	

Visited = [A , B] Unvisited = [C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

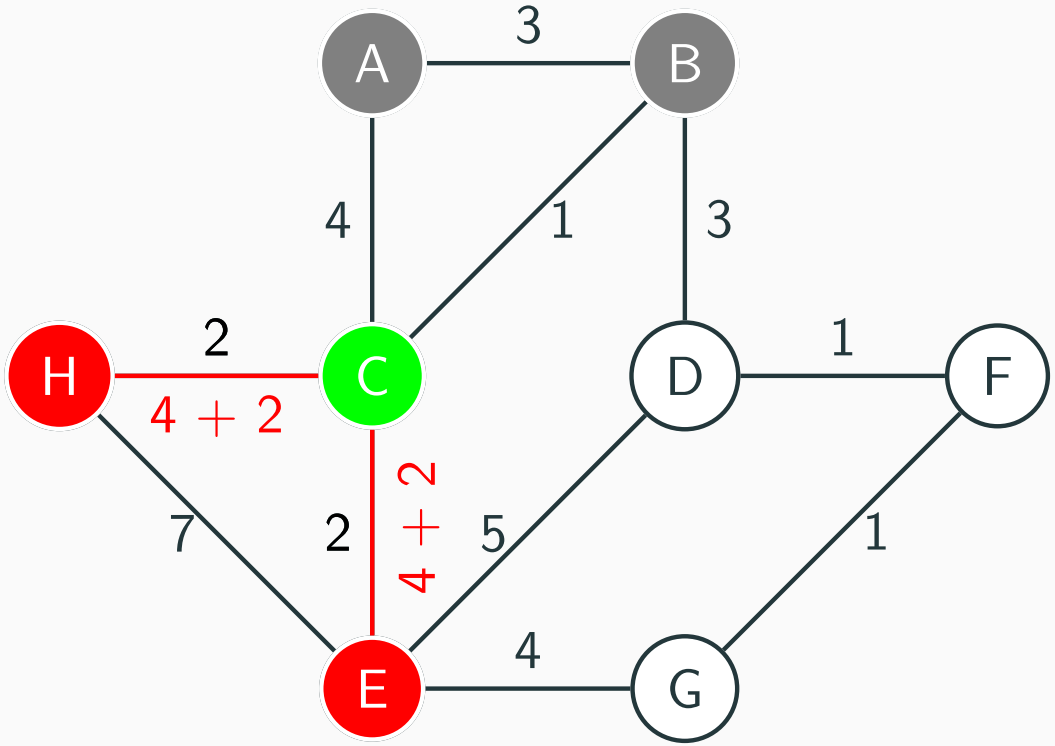


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	∞	
F	∞	
G	∞	
H	∞	

Visited = [A , B] Unvisited = [C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

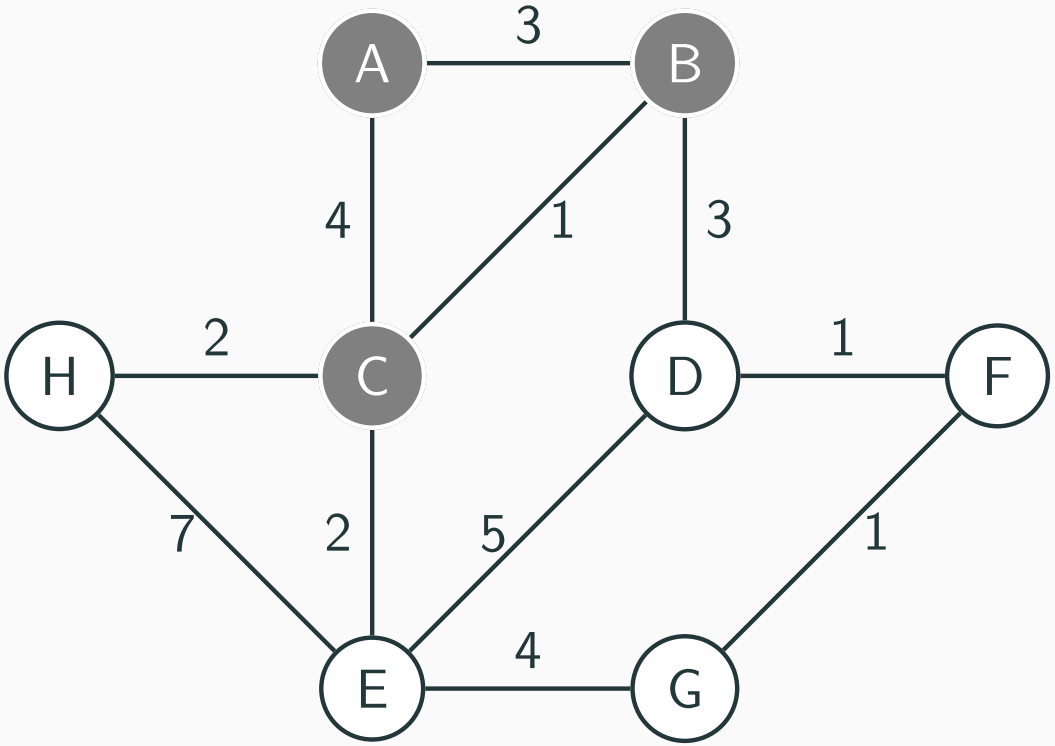


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	∞	
G	∞	
H	6	*C*

Visited = [A , B] Unvisited = [C, D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

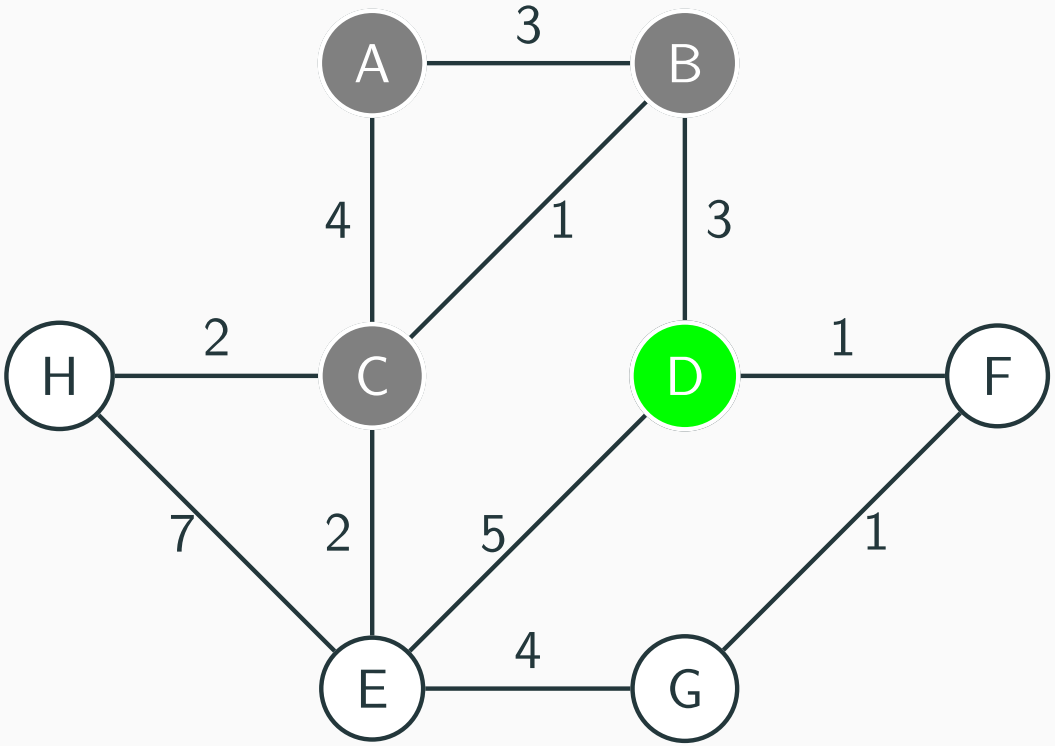


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	∞	
G	∞	
H	6	*C*

Visited = [A , B , C] Unvisited = [D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

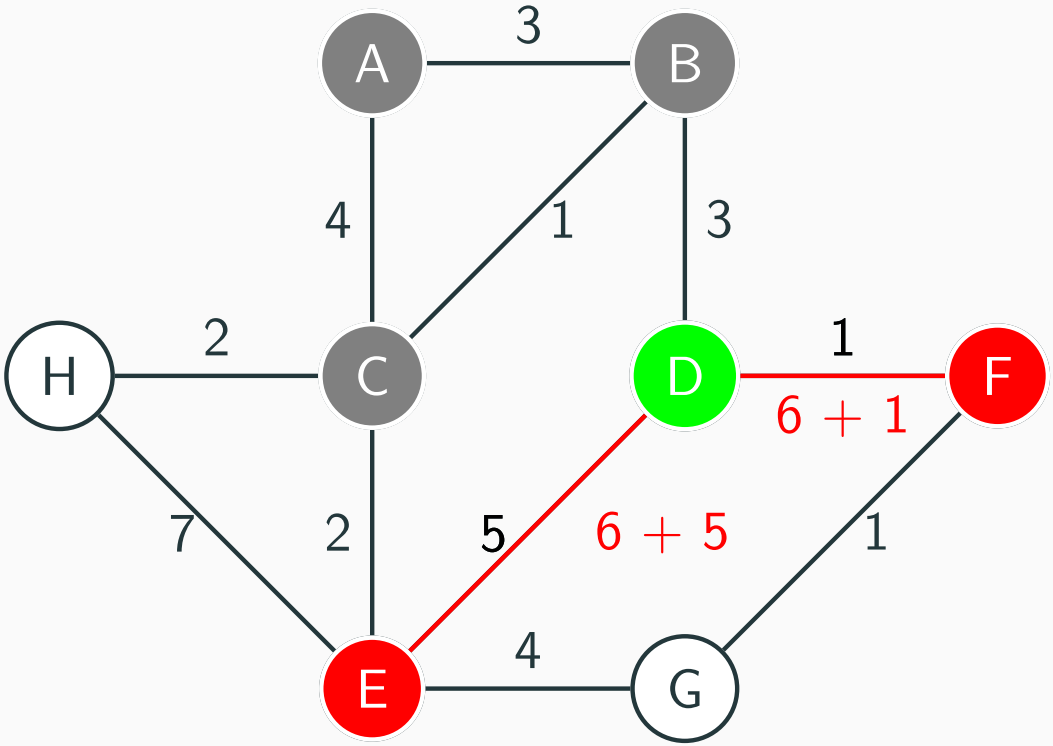


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	∞	
G	∞	
H	6	*C*

Visited = [A , B , C] Unvisited = [D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

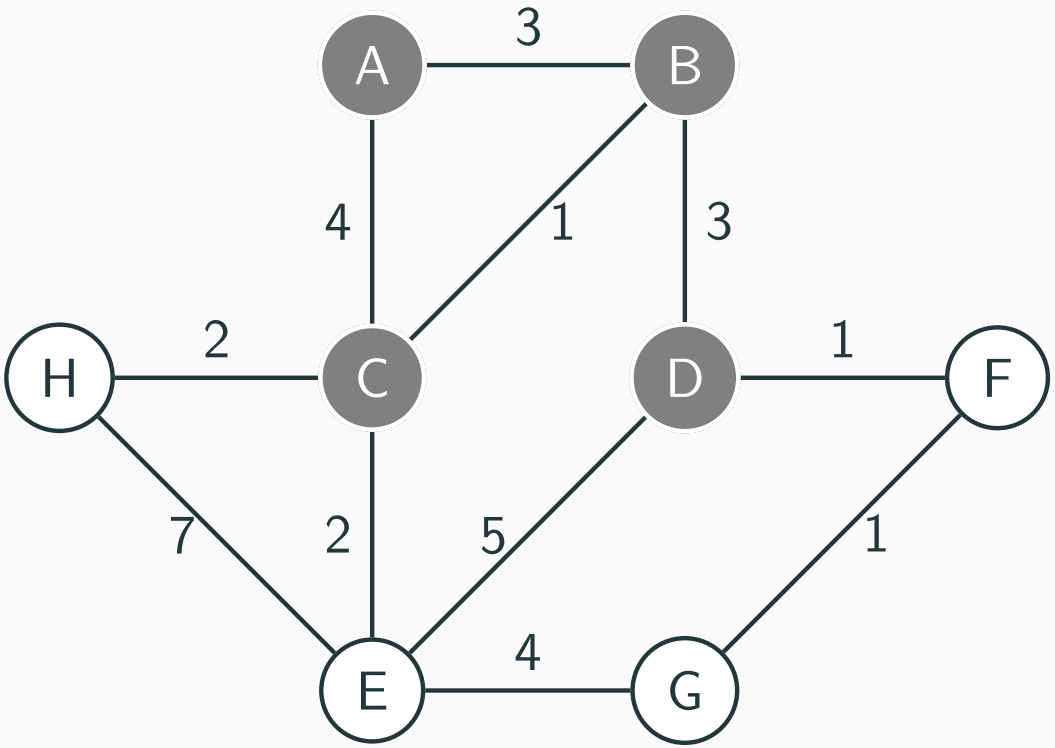


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	∞	
H	6	*C*

Visited = [A , B , C] Unvisited = [D, E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter



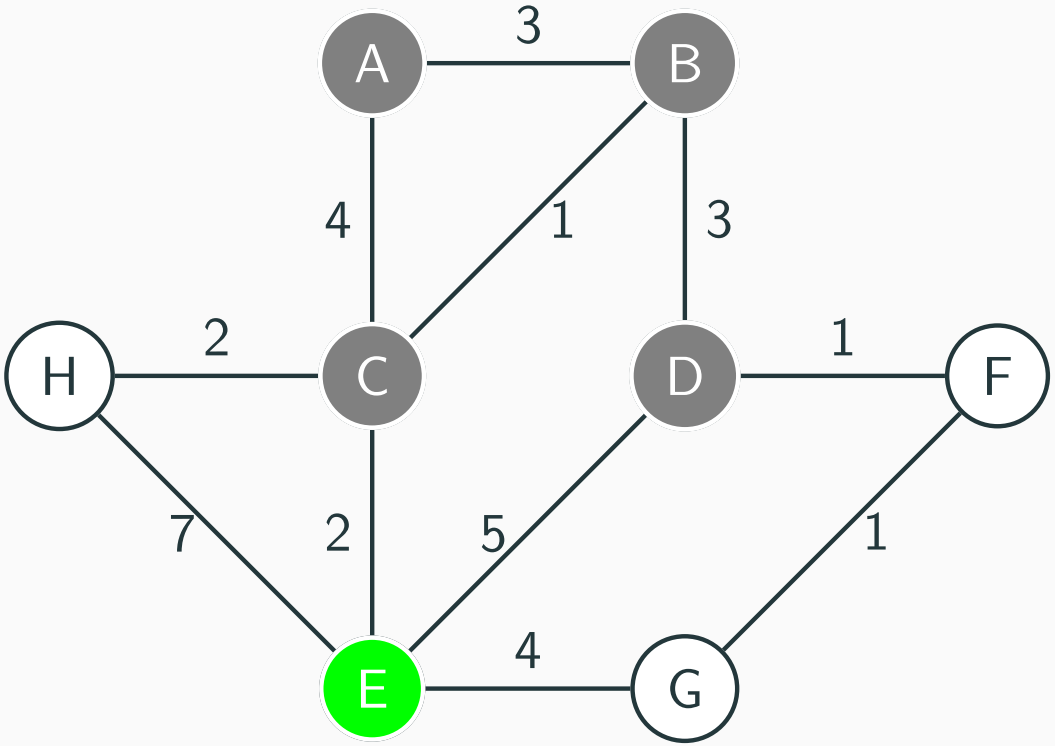
Poll 2

V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	∞	
H	6	*C*

Visited = [A , B , C , D] Unvisited = [E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

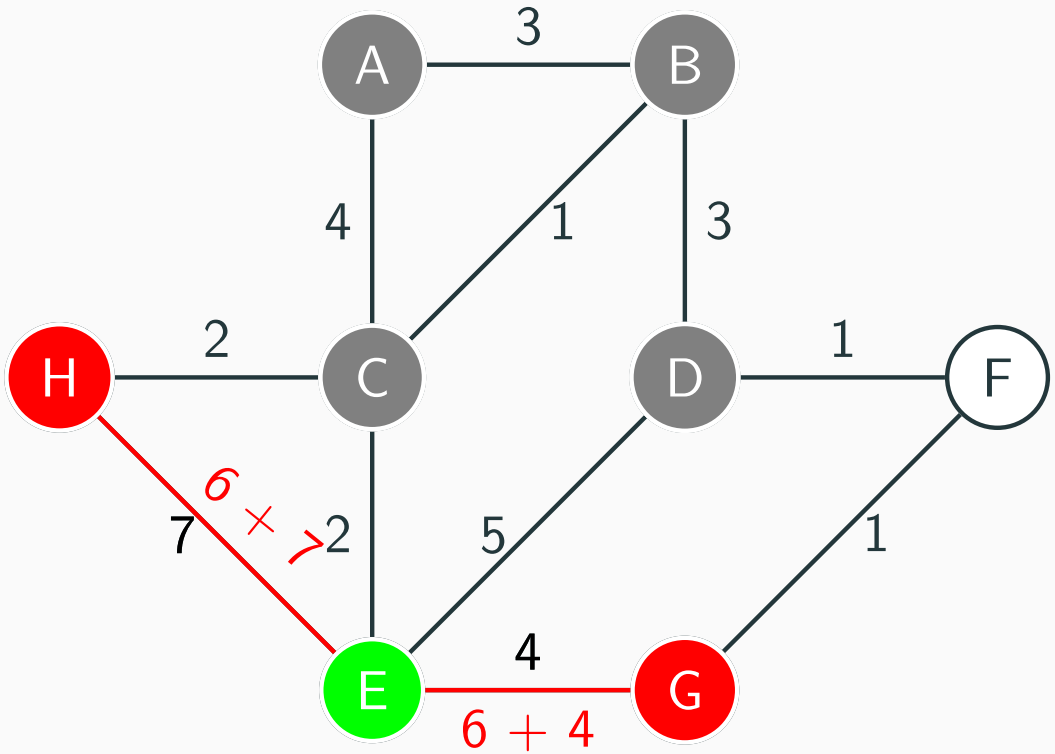


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	∞	
H	6	*C*

Visited = [A , B , C , D] Unvisited = [E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

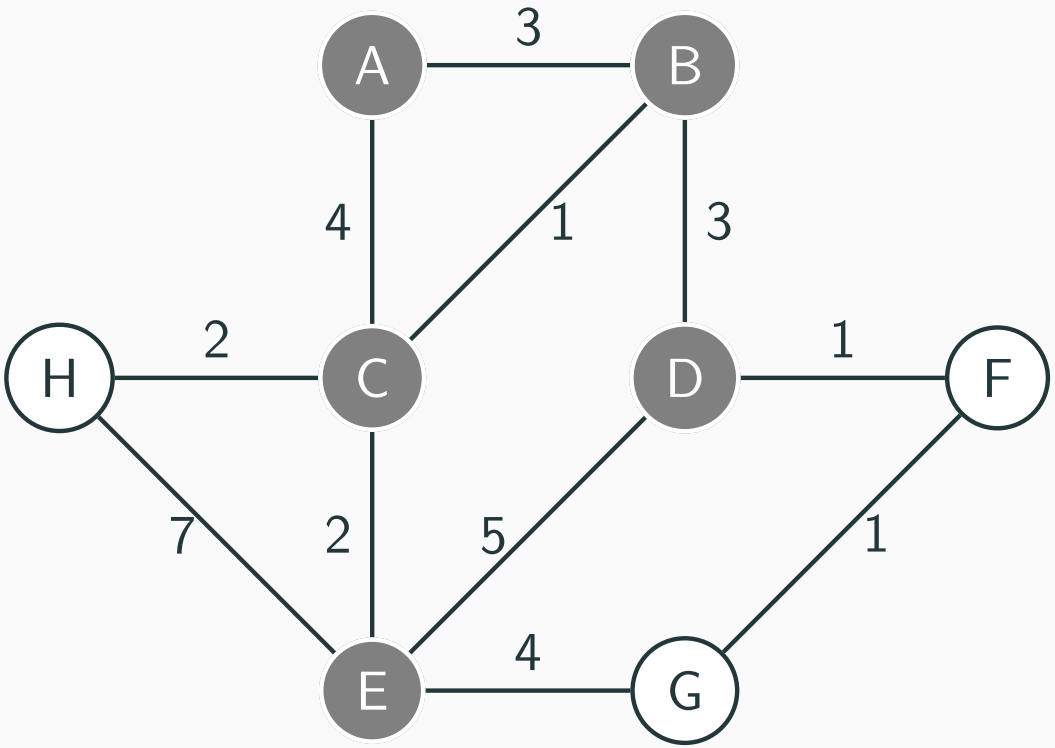


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	10	*E*
H	6	*C*

Visited = [A , B , C , D] Unvisited = [E, F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

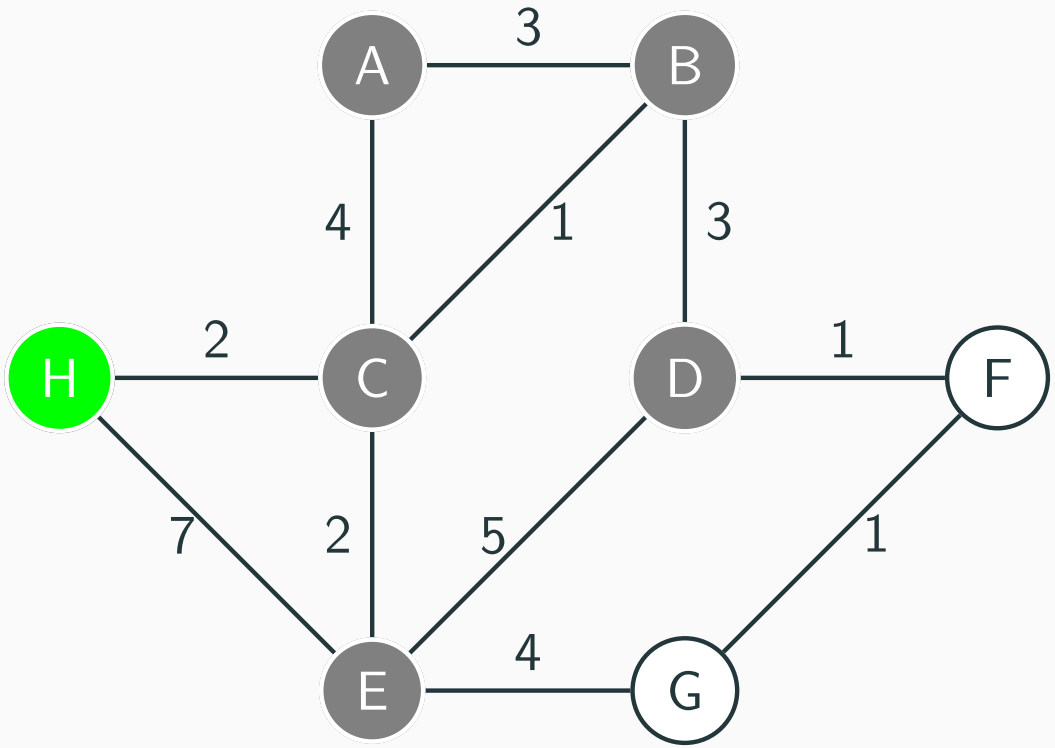


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	10	*E*
H	6	*C*

Visited = [A , B , C , D , E] Unvisited = [F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

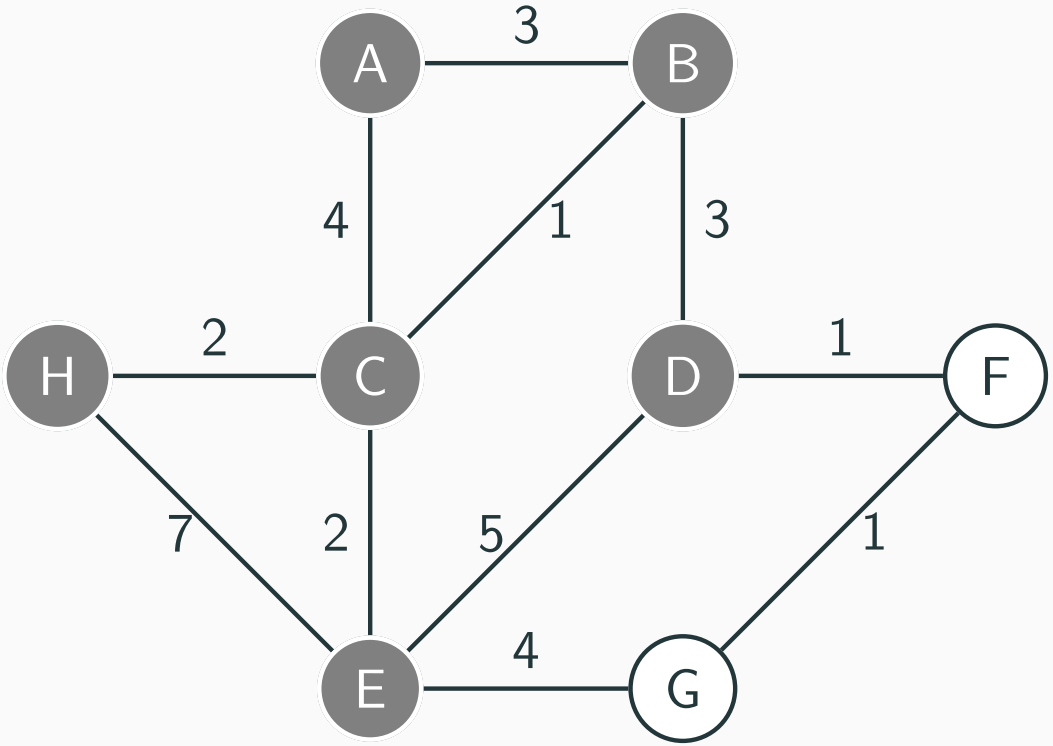


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	10	*E*
H	6	*C*

Visited = [A , B , C , D , E] Unvisited = [F, G, H]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

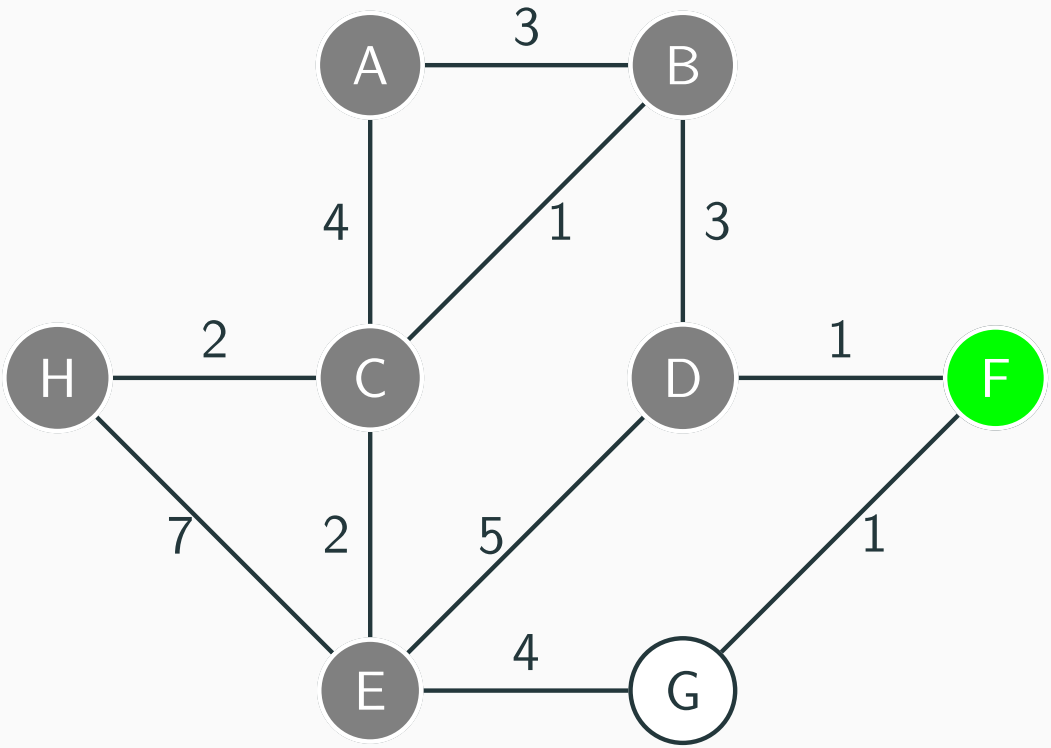


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	10	*E*
H	6	*C*

Visited = [A , B , C , D , E , H] Unvisited = [F, G,]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

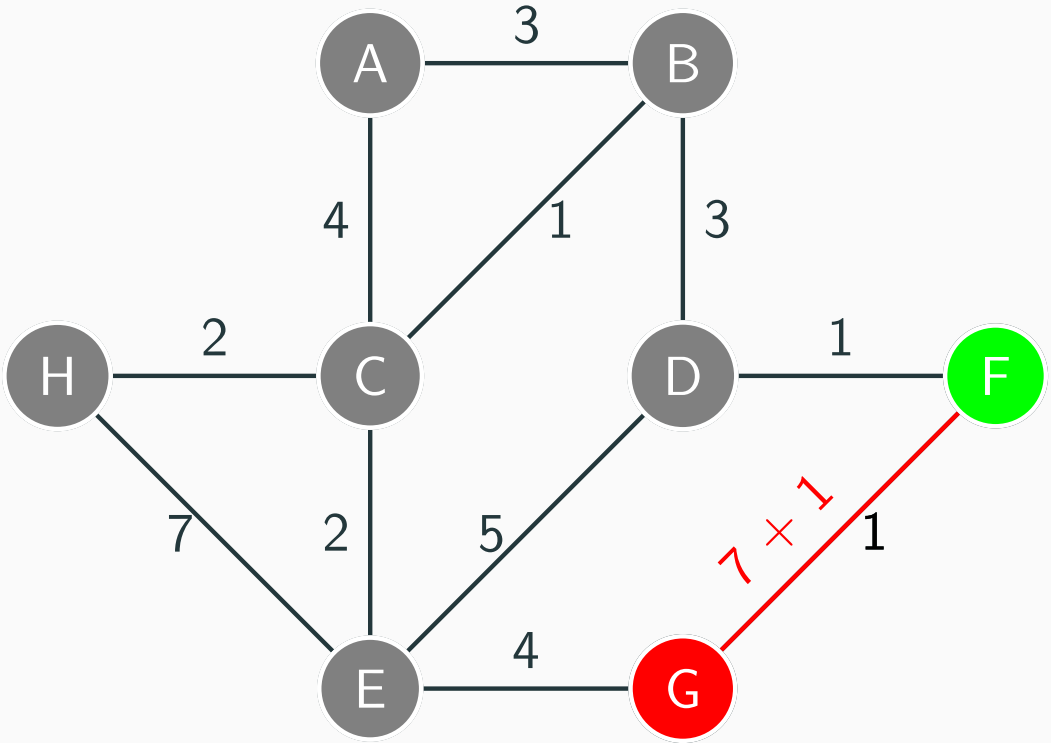


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	10	*E*
H	6	*C*

Visited = [A , B , C , D , E , H] Unvisited = [F, G,]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

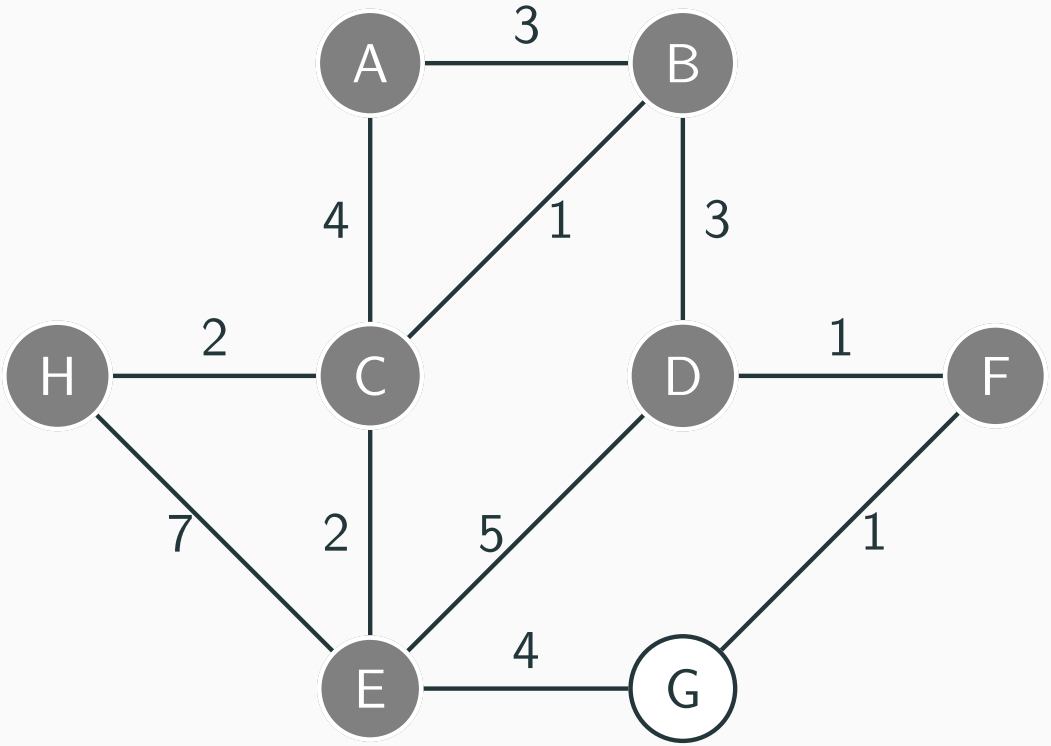


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	8	*F*
H	6	*C*

Visited = [A , B , C , D , E , H] Unvisited = [F, G,]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

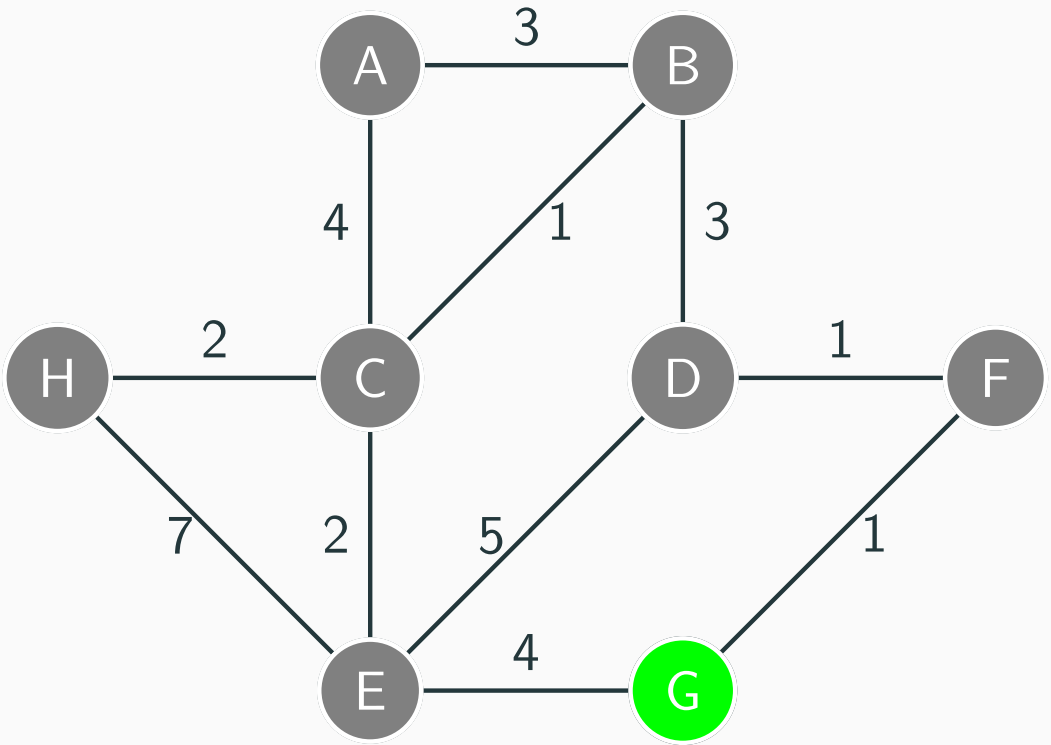


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	8	*F*
H	6	*C*

Visited = [A , B , C , D , E , H , F] Unvisited = [G,]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter

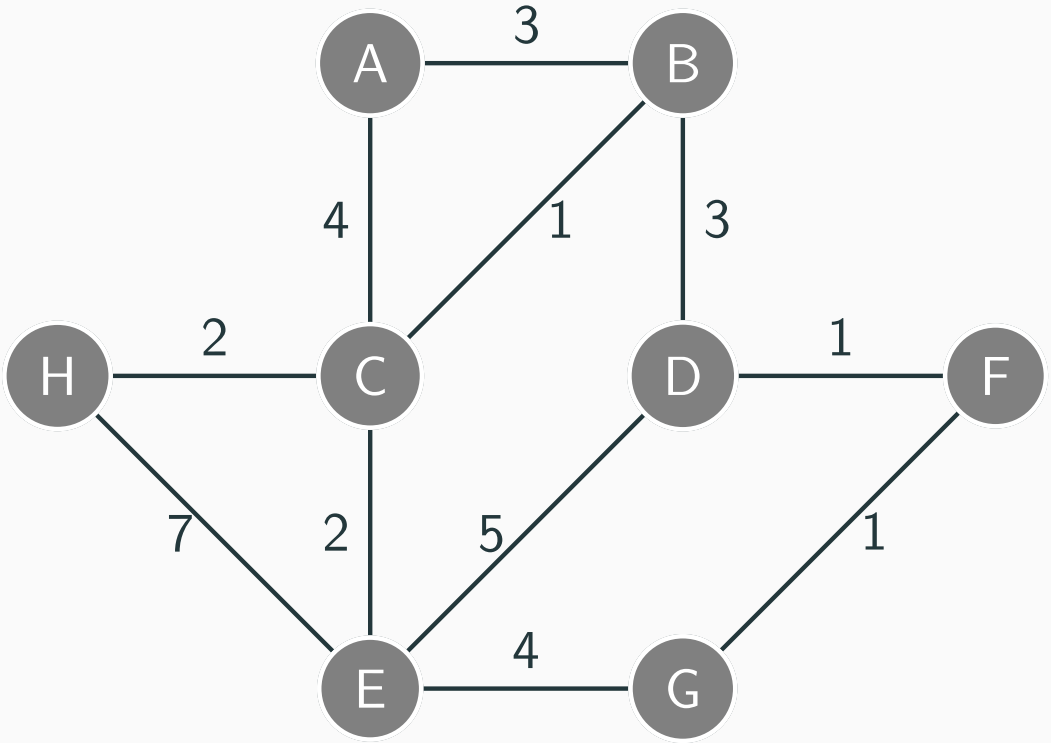


V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	8	*F*
H	6	*C*

Visited = [A , B , C , D , E , H , F] Unvisited = [G,]

Dijkstra Algorithm: Example

- Visit the unvisited vertex with he smallest known distance from the start vertex.
  - For the current vertex calculate the distance for all its unvisited neighbours from the source and update the known distance if the new distance is shorter



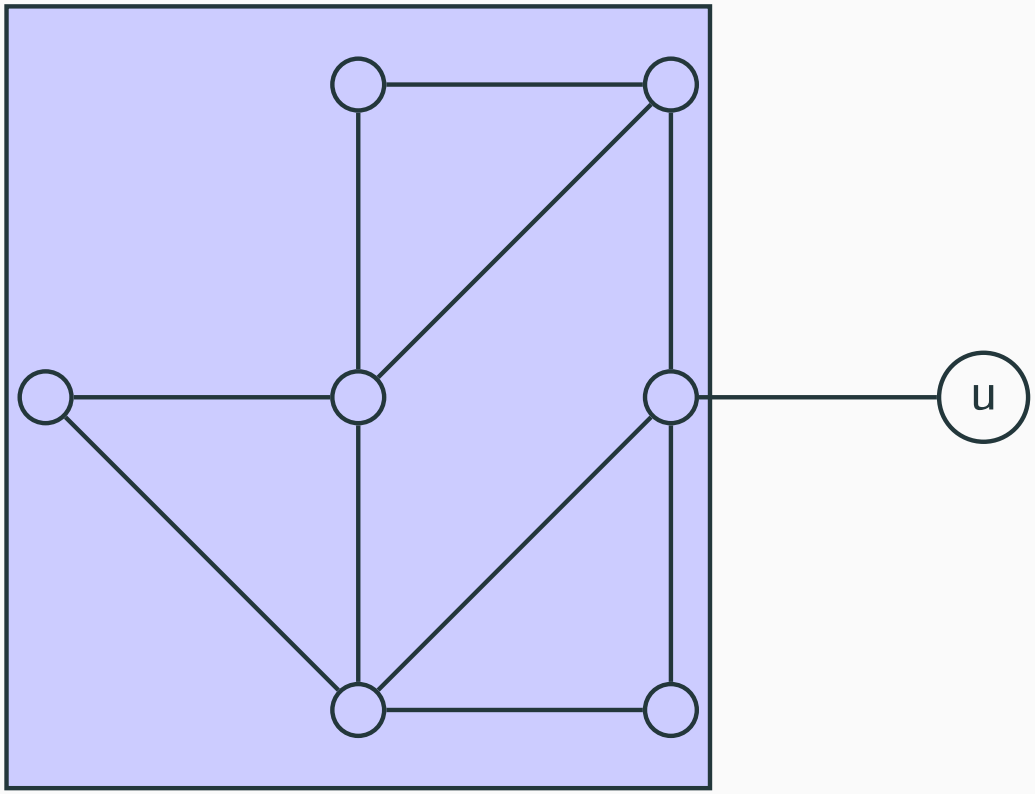
V	Distance	Parent
A	0	-
B	3	*A*
C	4	*A*
D	6	*B*
E	6	*C*
F	7	*D*
G	8	*F*
H	6	*C*

Visited = [A , B , C , D , E , H , F , G] Unvisited = [ ]

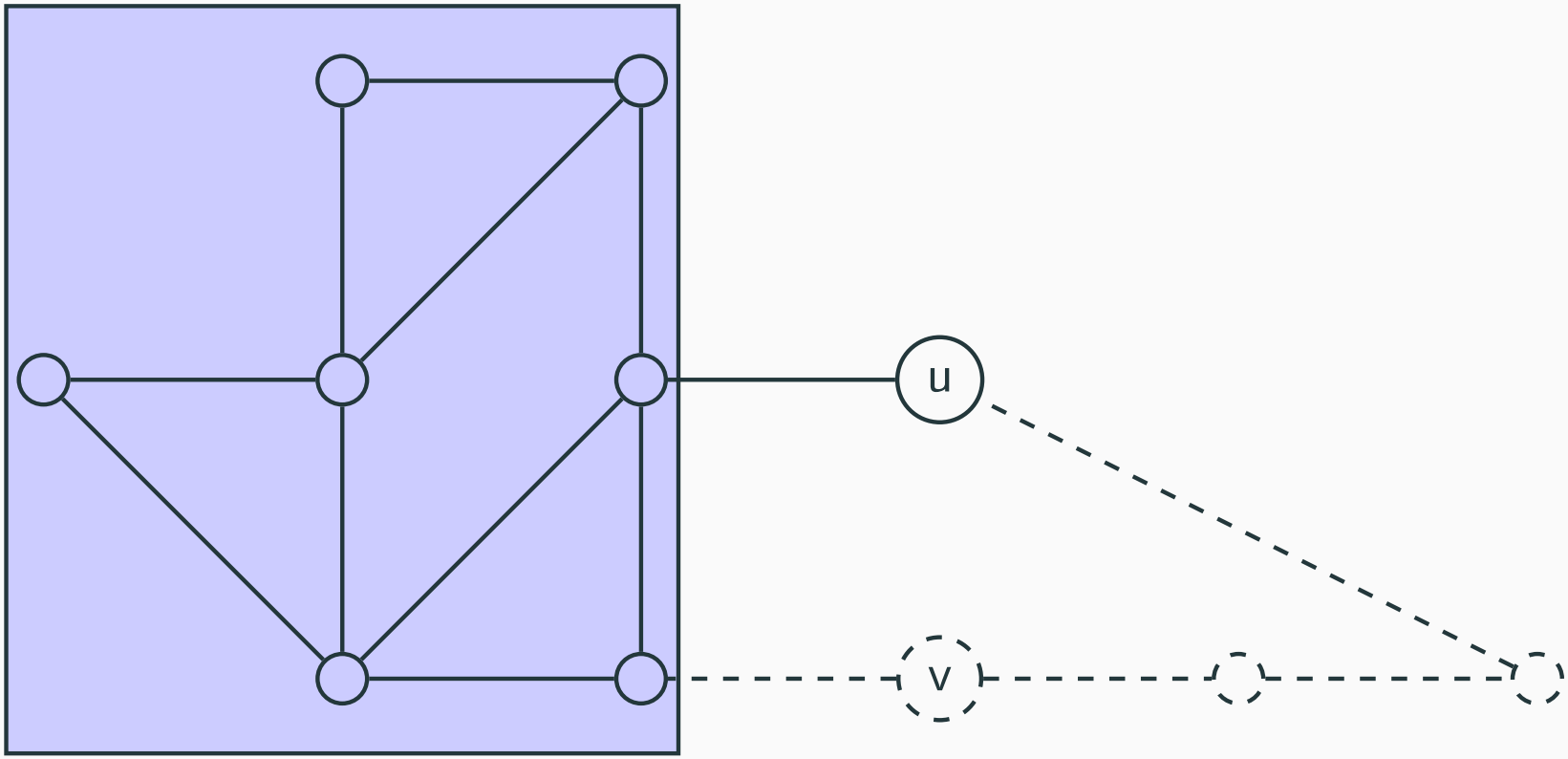
Dijkstra Algorithm: Proof of Correctness

- Assume that Dijkstra's algorithm does not always produce the shortest path.
- Let u be the first vertex for which the algorithm finds an incorrect shortest path.
- Assume the true shortest path to u is through a vertex v, but the algorithm incorrectly determines a different path.
- Since u is the first vertex with an incorrect path, $d[v]$ must be correct when u is processed.
- When u is selected for processing, all vertices with shorter paths (including v) should have already been processed.
- However, since v was not processed before u, $d[v]$ must be greater than $d[u]$, contradicting the assumption that the path through v is shorter.
- Hence, the assumption that Dijkstra's algorithm does not always produce the shortest path is false.
- Therefore, Dijkstra's algorithm correctly finds the shortest path.

Dijkstra Algorithm: Proof of Correctness



Dijkstra Algorithm: Proof of Correctness



Dijkstra Algorithm: Time Complexity

- Dijkstra's algorithm uses a priority queue for efficient distance updates.
- The main operations are:
  - Insertion into the priority queue.
  - Extracting the minimum element from the priority queue.
  - Decreasing the key value in the priority queue.
- Insertion and decrease-key operations take $O(\log V)$ time.
- Extract-min operation takes $O(\log V)$ time.
- Total time complexity with a binary heap is $O((V + E) \log V)$.
- For a complete graph, where $E = O(V^2)$:
- With a binary heap, the time complexity is $O(V^2 \log V)$.

Dijkstra's Algorithm: Real-World Applications

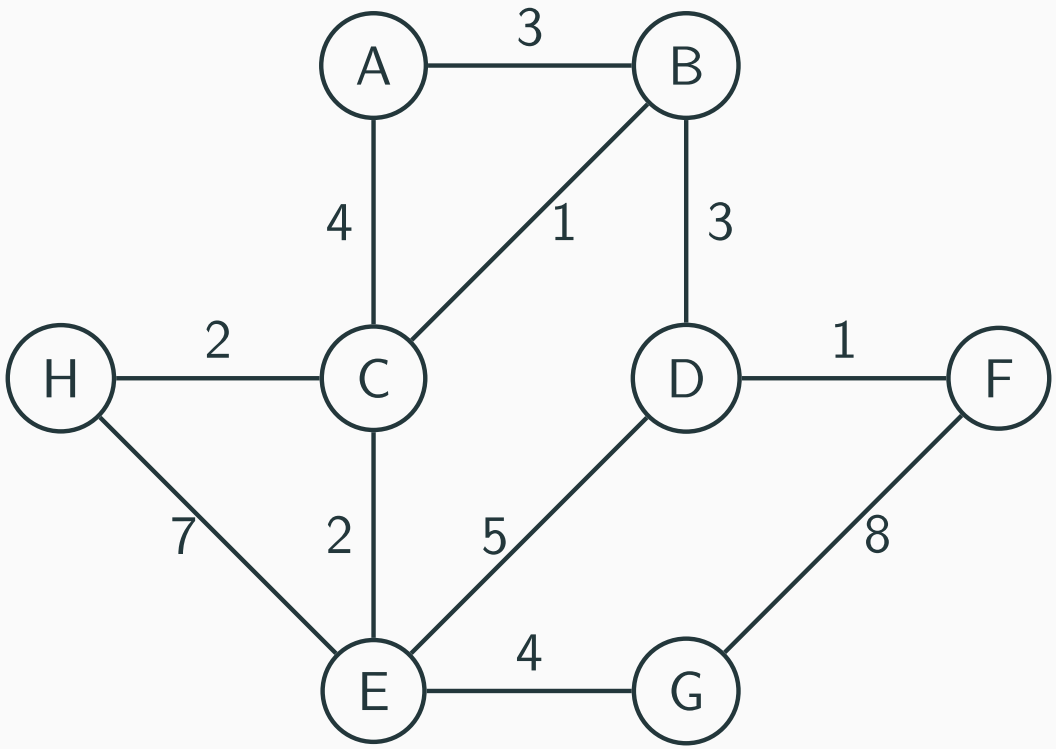
- Used to find the shortest path between two locations, providing efficient routing for vehicles and pedestrians.
- Used in protocols like OSPF (Open Shortest Path First) to determine the most efficient path for data packets to travel across a network.
- Helps in planning and managing traffic flow by identifying the shortest routes and reducing congestion.
- Applied in optimizing delivery routes for logistics companies to ensure timely and cost-effective deliveries.
- Used in AI pathfinding to navigate characters or objects through complex environments in video games.

Shortest Path Problem: Extensions

- **Multi-Source Shortest Paths**:
  - Floyd-Warshall Algorithm: Solves for all pairs shortest paths in $O(V^3)$ time.
- **Negative Weights**:
  - Bellman-Ford Algorithm: Handles negative weights, runs in $O(VE)$ time.
- **Negative Weight Cycles**:
  - Detects negative weight cycles, which can lead to undefined shortest paths.

Minimum Spanning Tree

MST: The Problem



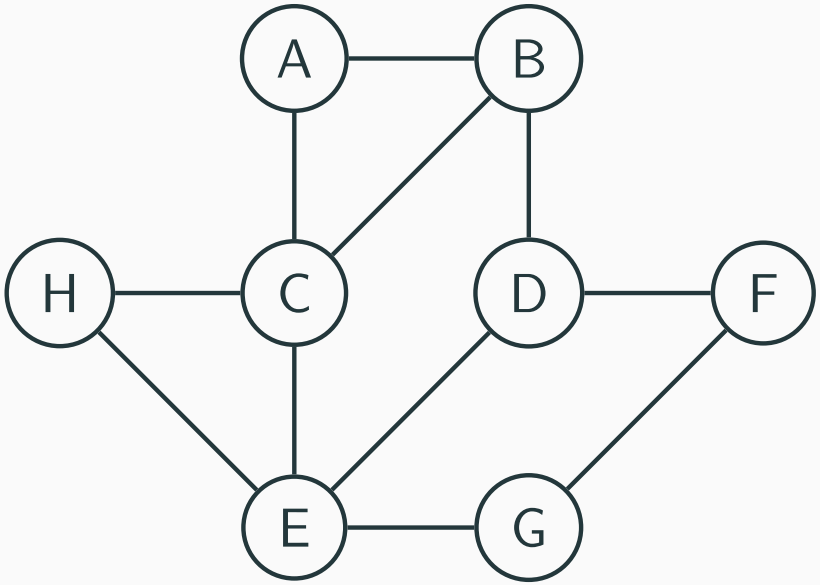
- Find the subset of edges that connects all the vertices with the minimum total weight.
  - E.g., Given a weighted graph, find the minimum spanning tree that connects all nodes.
    - The resulting tree should have the minimum possible total edge weight.

MST: Applications

- **Network Design**:
  - Used to design efficient networks (computer, telecommunication, etc.) with minimal wiring/cabling costs.
- **Urban Planning**:
  - Helps in designing road networks to connect cities with minimum total road length.
- **Clustering Analysis**:
  - Applied in hierarchical clustering methods to determine clusters in data.
- **Circuit Design**:
  - Used to minimize the total length of wires in circuit design.

Spanning Tree: Properties

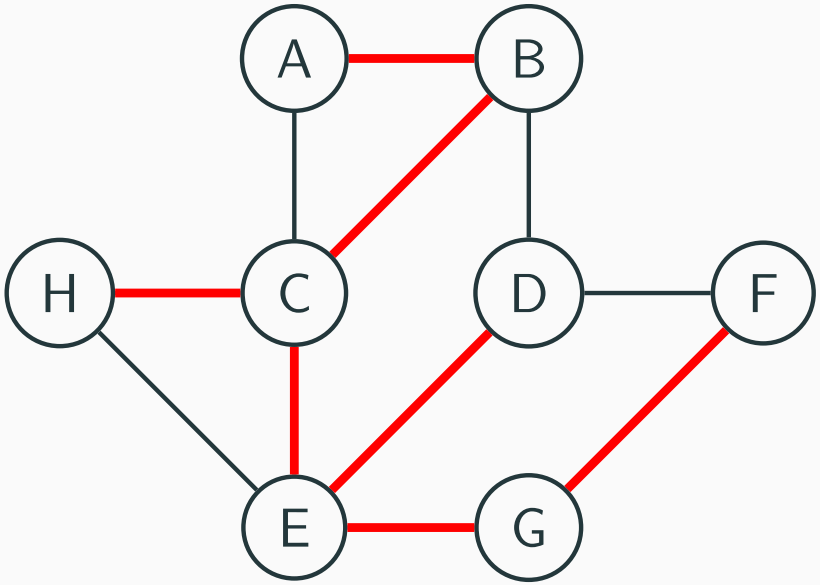
- Given a graph $G = (V, E)$
  - A spanning tree of the graph will be a graph with $|V|$ vertices and $|V| - 1$ edges
  - The number of possible spanning trees is $|E|C(|V| - 1) - |C|$, where C is the number of cycles in the graph



Example Spanning Trees

Spanning Tree: Properties

- Given a graph $G = (V, E)$
  - A spanning tree of the graph will be a graph with $|V|$ vertices and $|V| - 1$ edges
  - The number of possible spanning trees is $|E|C(|V| - 1) - |C|$, where C is the number of cycles in the graph

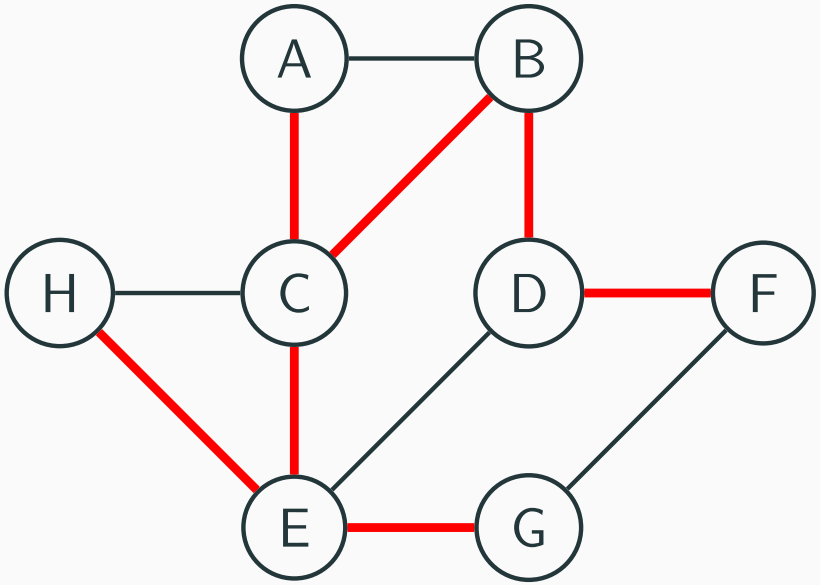


Example Spanning Trees

- (A, B), (B, C), (C, H), (C, E), (E, D), (E, G), (G, F)

Spanning Tree: Properties

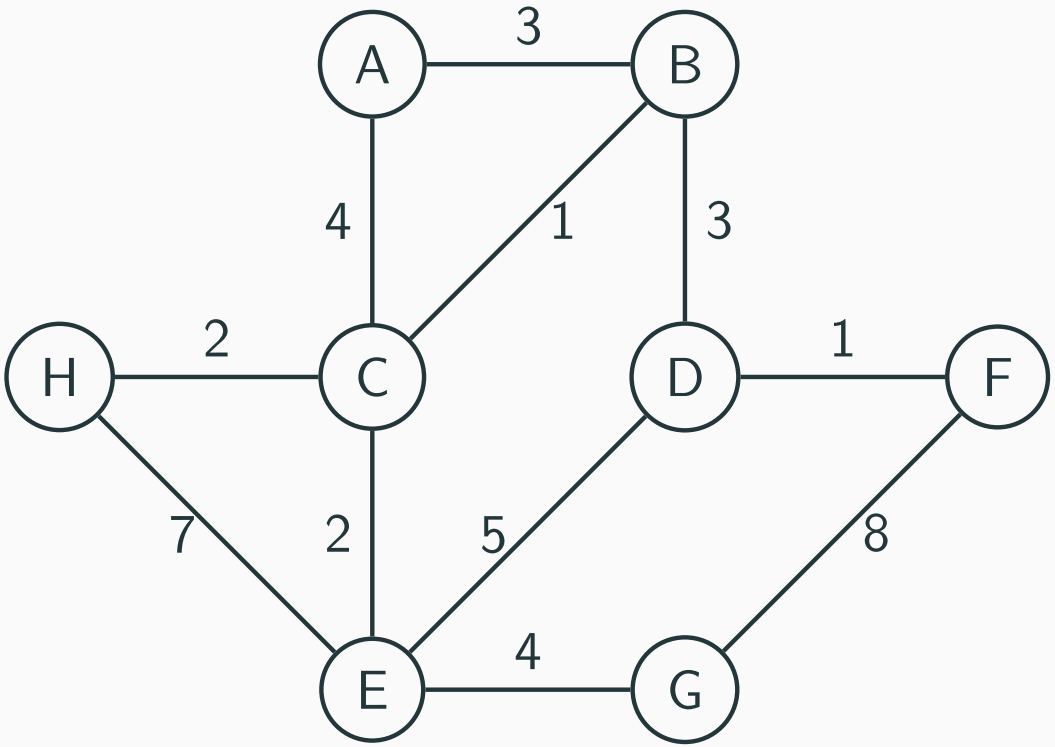
- Given a graph $G = (V, E)$
  - A spanning tree of the graph will be a graph with $|V|$ vertices and $|V| - 1$ edges
  - The number of possible spanning trees is $|E|C(|V| - 1) - |C|$, where C is the number of cycles in the graph



Example Spanning Trees

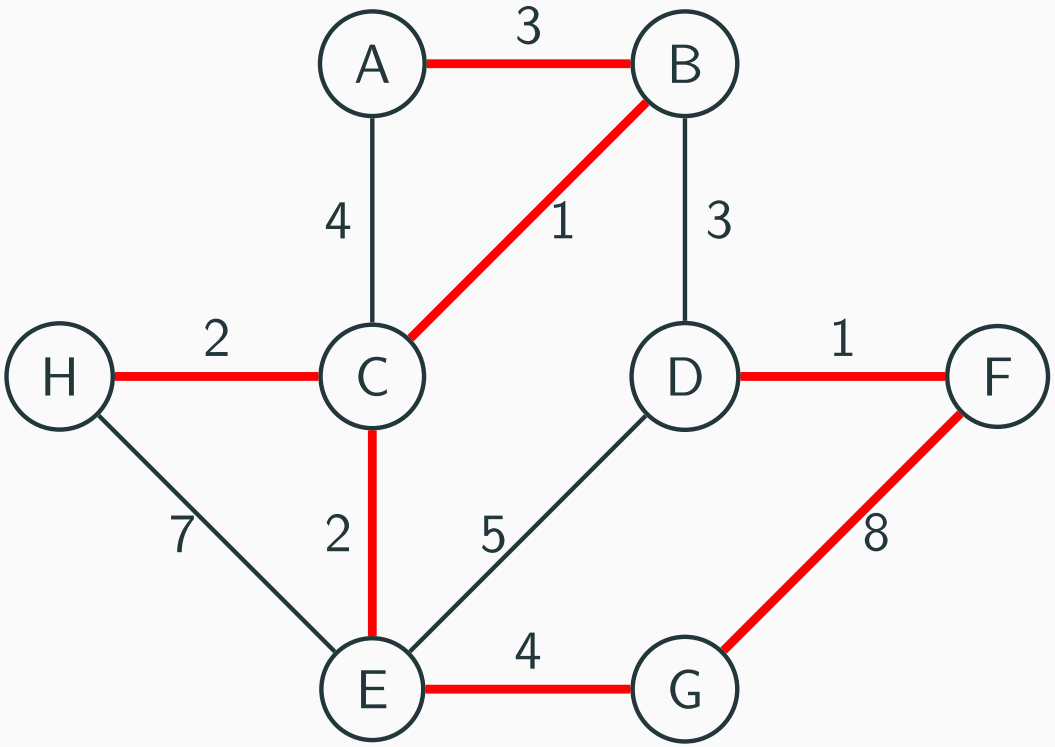
- (A, B), (B, C), (C, H), (C, E), (E, D), (E, G), (G, F)
- (A, C), (C, B), (B, D), (C, E), (E, H), (E, G), (D, F)

MST: Example



- Example: Consider the given graph and following spanning trees.

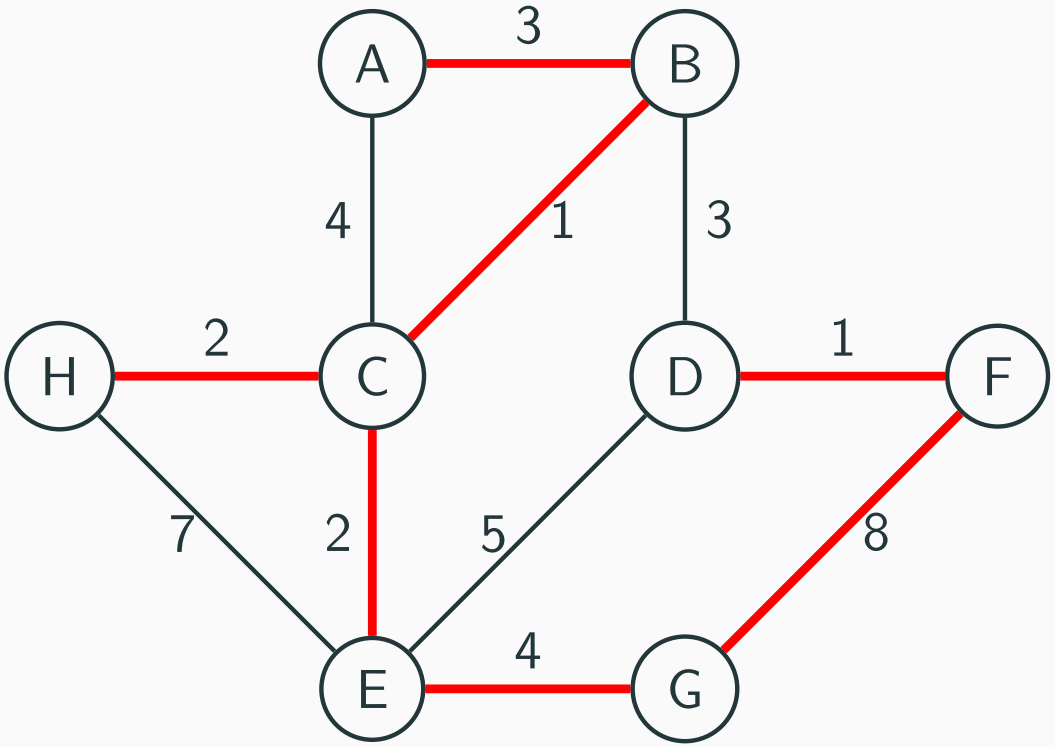
MST: Example



- Example: Consider the given graph and following spanning trees.
  - (A, B) + (B, C) + (C, E) + (D, F) + (E, G) + (C, H) + (F, G)

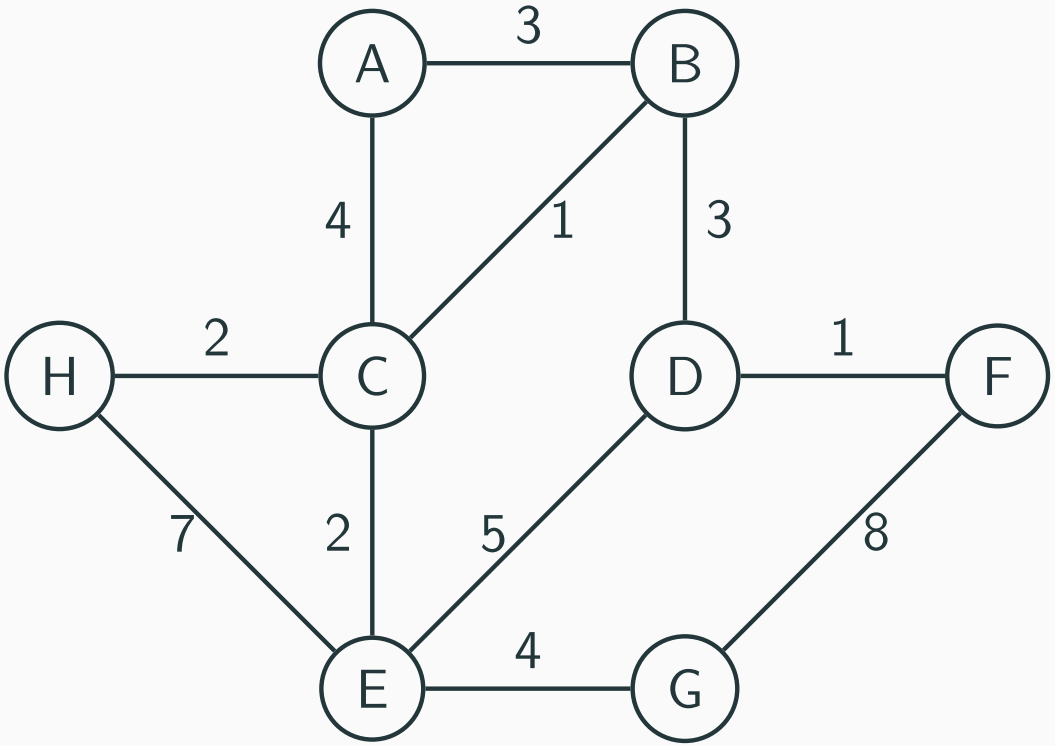
Poll 3

MST: Example



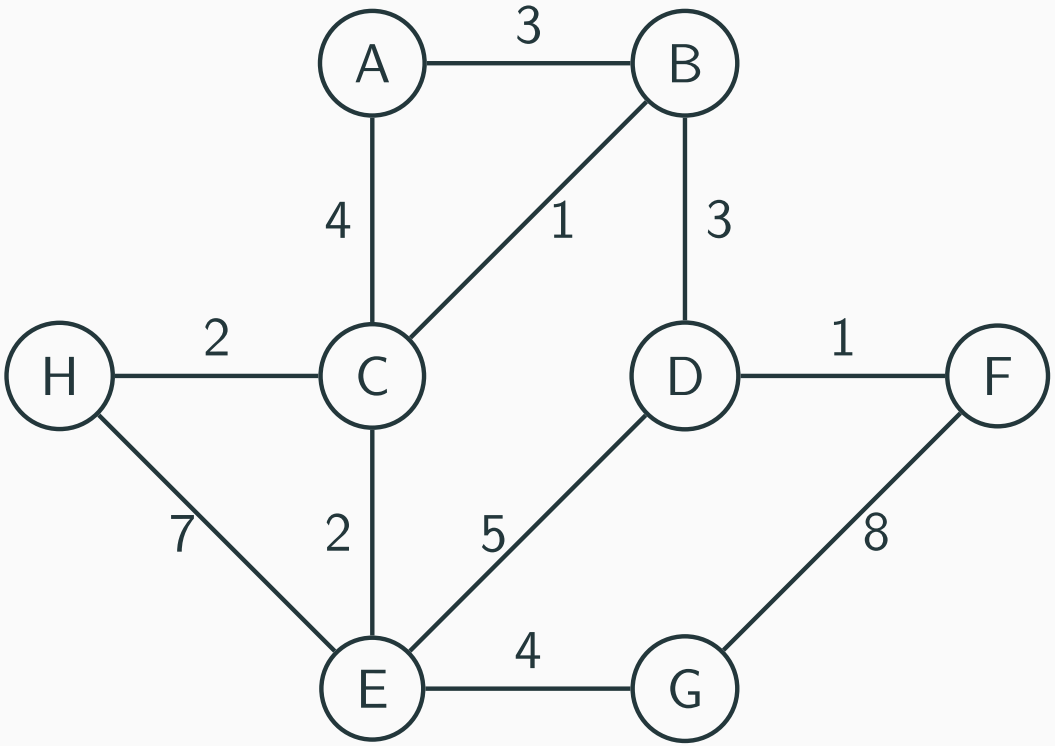
- Example: Consider the given graph and following spanning trees.
  - (A, B) + (B, C) + (C, E) + (D, F) + (E, G) + (C, H) + (F, G)
    - Weight ST: $3 + 1 + 2 + 1 + 4 + 2 + 8 = 21$

MST: Example



- Example: Consider the given graph and following spanning trees.
  - (A, B) + (B, C) + (C, E) + (D, F) + (E, G) + (C, H) + (F, G)
    - Weight ST: $3 + 1 + 2 + 1 + 4 + 2 + 8 = 21$
- What is the Weight of the MST: Poll 4

MST: Example



- Example: Consider the given graph and following spanning trees.
  - (A, B) + (B, C) + (C, E) + (D, F) + (E, G) + (C, H) + (F, G)
    - Weight ST: $3 + 1 + 2 + 1 + 4 + 2 + 8 = 21$
- What is the Weight of the MST: 16

Kruskal's Algorithm: Idea

- **Greedy Choice Property**:
  - At each step, the algorithm makes a locally optimal choice by selecting the smallest weight edge available.
  - This greedy choice leads to a globally optimal solution for the MST.
- **Optimal Substructure**:
  - Any subset of edges that forms an MST for a subgraph of the original graph is part of the MST for the entire graph.
  - This property allows the algorithm to build the MST incrementally.
- Kruskal's Algorithm constructs the MST by adding edges in increasing order of weight.
- At each step, the edge added does not form a cycle with the previously added edges.
- Uses a Disjoint Set data structure (Union-Find) to efficiently manage the connected components.

Union-Find Data Structure

- A data structure that keeps track of a set of elements partitioned into disjoint (non-overlapping) subsets.
- Provides efficient operations to manage and merge these subsets.
- **Key Operations**:
  - **Find**: Determine the subset (or set representative) of an element.
  - **Union**: Merge two subsets into a single subset.
- Before adding an edge, use the Find operation to check if the two vertices are in the same subset.
- If they are in the same subset, adding the edge would form a cycle.
- If they are in different subsets, use the Union operation to merge the subsets, safely adding the edge without forming a cycle.

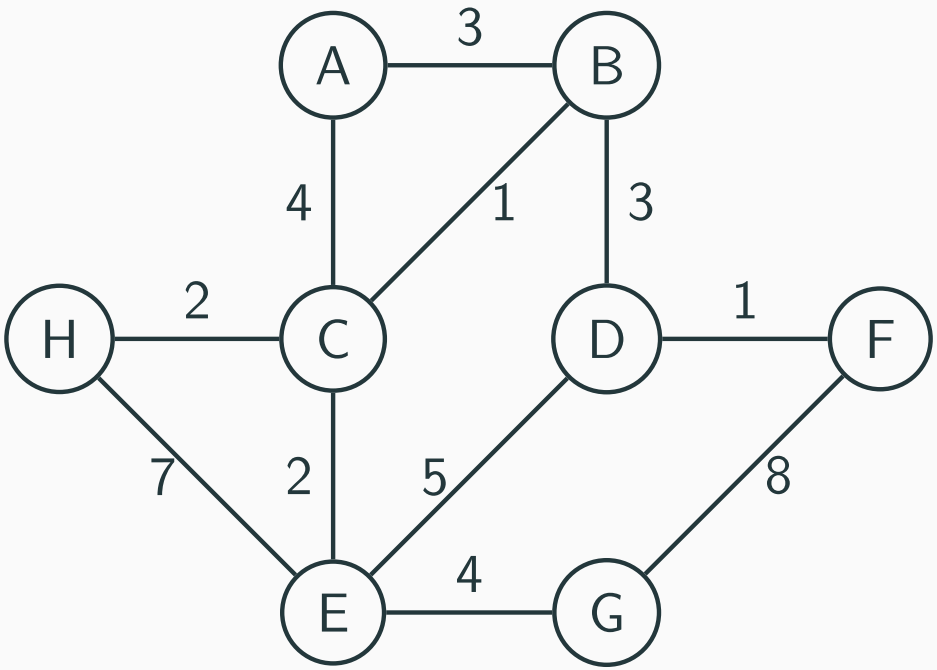
Kruskal's Algorithm: Pseudo-code

Algorithm 9 Kruskal's Algorithm

1: **Input:** Graph $G = (V, E)$
2: **Output:** Minimum Spanning Tree T
3: Sort edges E by weight
4: Initialize $T = \emptyset$
5: **for** each edge (u, v) in E (in increasing order of weight) **do**
6: **if** adding (u, v) to T does not form a cycle **then**
7: Add (u, v) to T
8: **end if**
9: **end for**
10: **return** T

Kruskal's Algorithm: Example

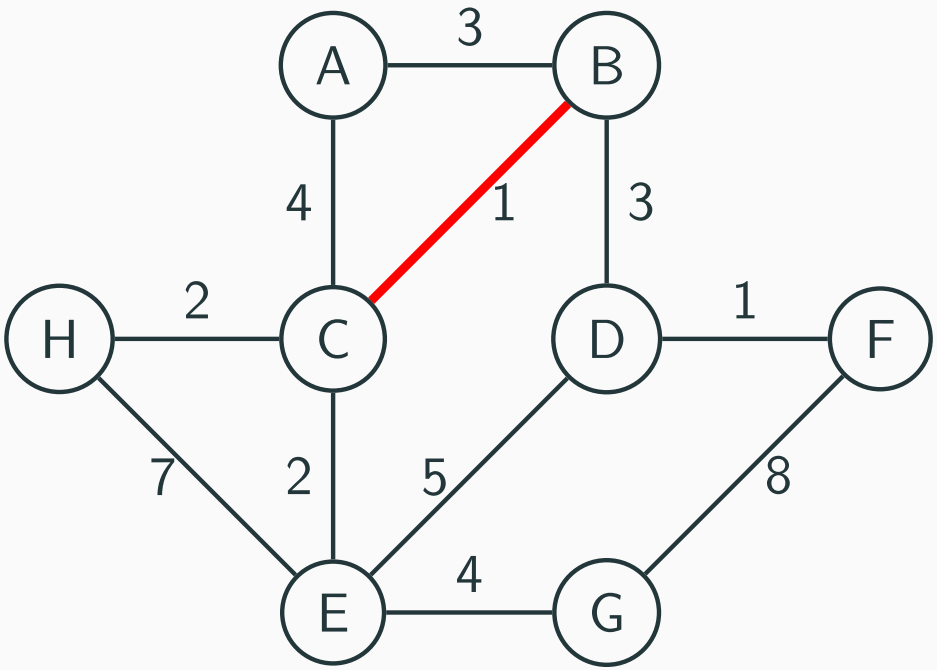
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:

Kruskal's Algorithm: Example

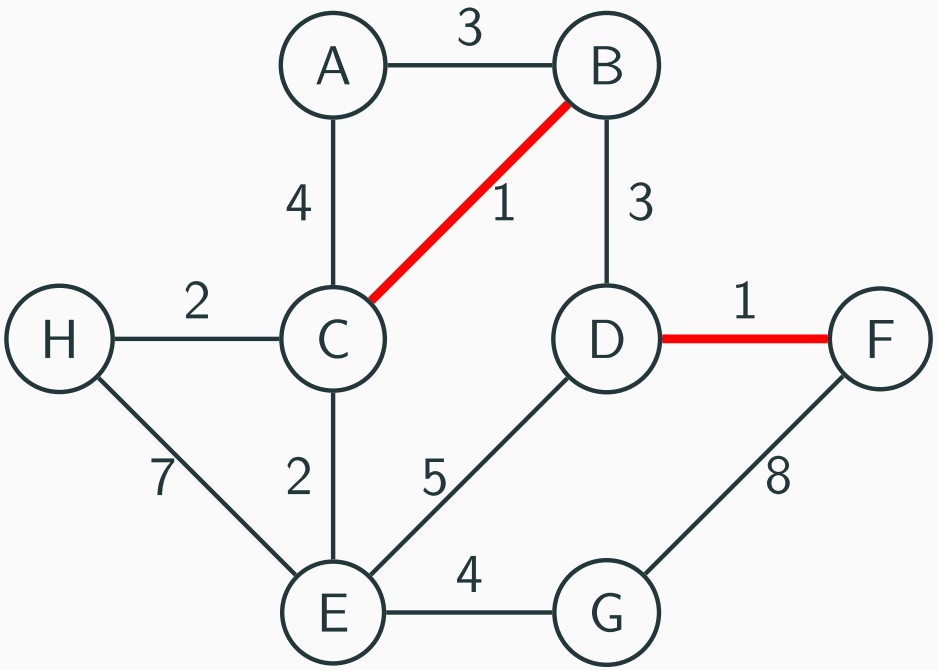
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1

Kruskal's Algorithm: Example

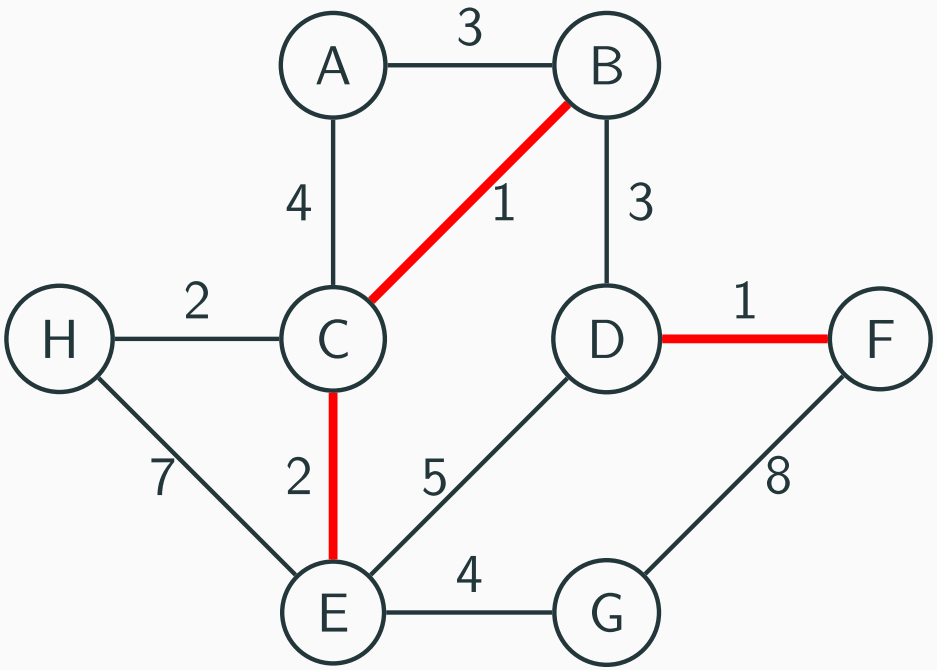
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1

Kruskal's Algorithm: Example

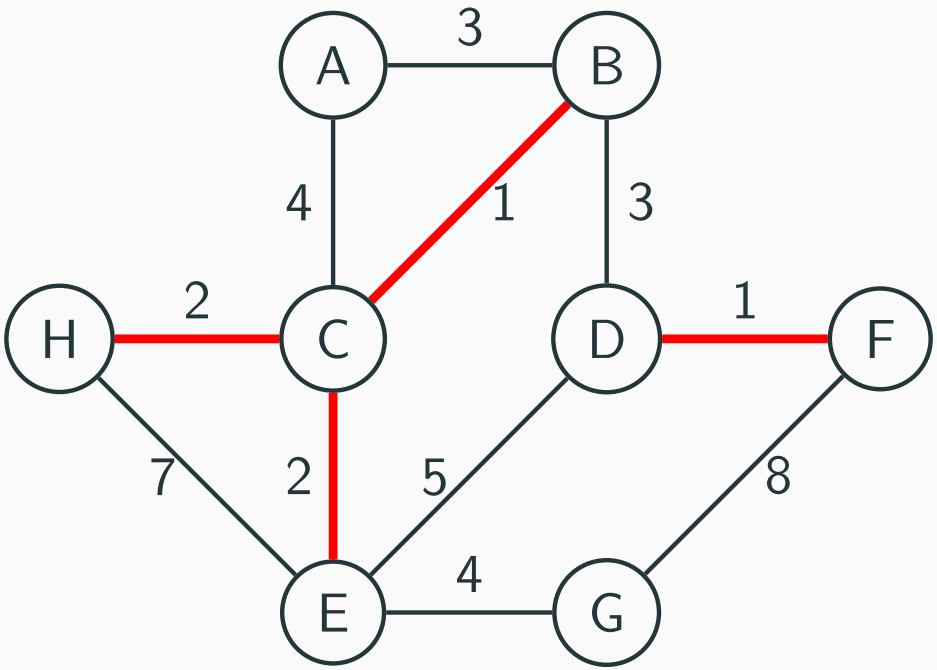
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1
  - (C, E) = 2

Kruskal's Algorithm: Example

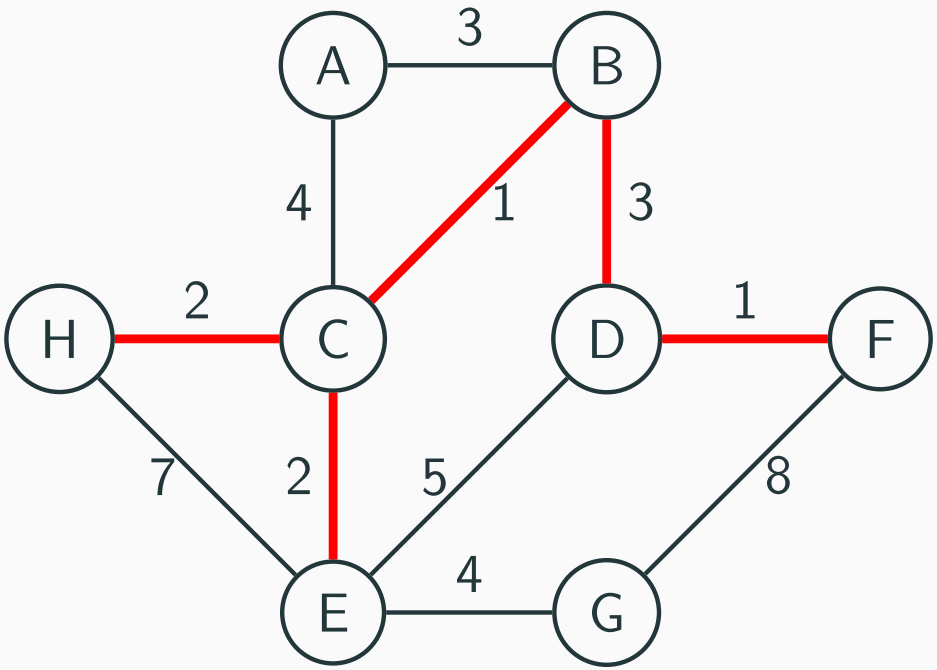
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1
  - (C, E) = 2
  - (C, H) = 2

Kruskal's Algorithm: Example

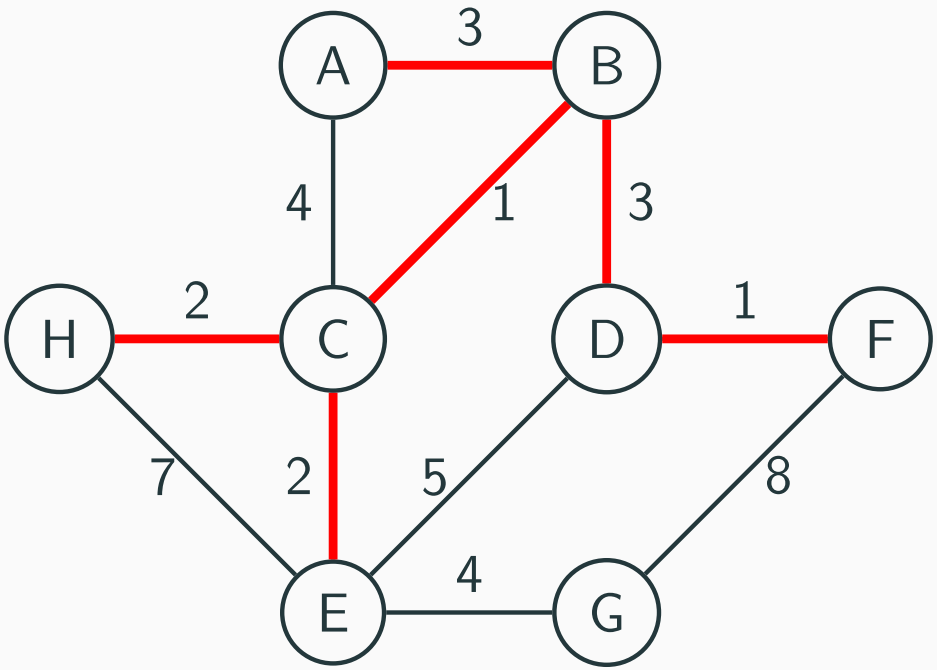
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (B, D) = 3

Kruskal's Algorithm: Example

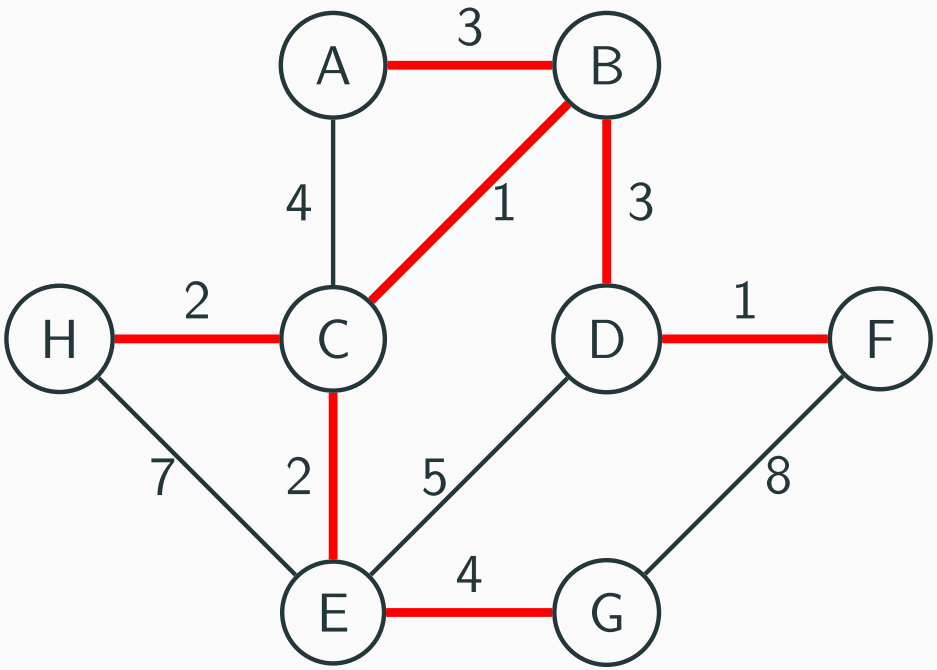
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (B, D) = 3
  - (A, B) = 3

Kruskal's Algorithm: Example

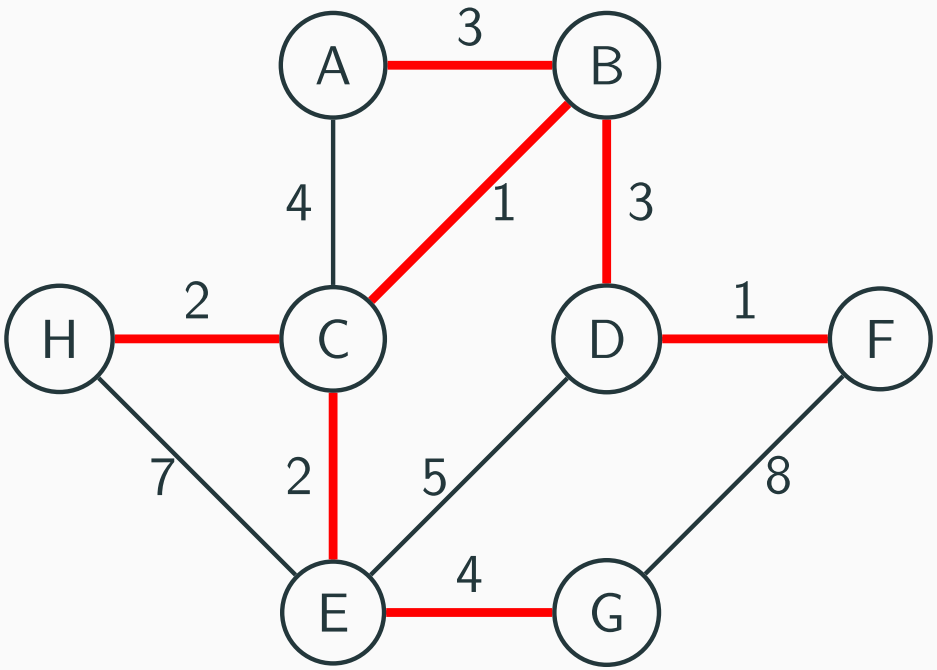
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (B, D) = 3
  - (A, B) = 3
  - (E, G) = 4

Kruskal's Algorithm: Example

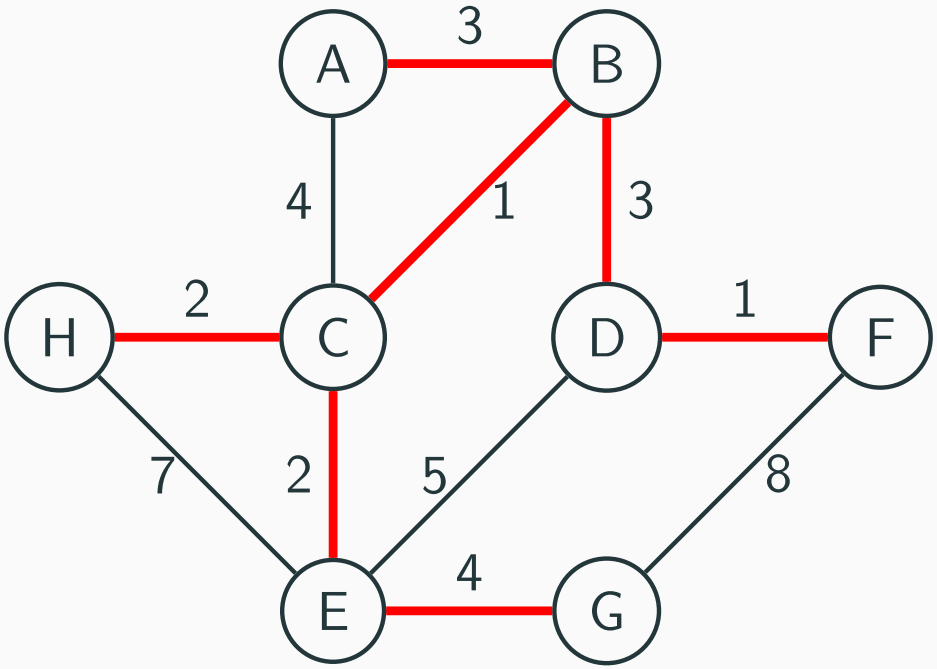
- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (B, D) = 3
  - (A, B) = 3
  - (E, G) = 4

Kruskal's Algorithm: Example

- Start with an empty MST.
  - Add the smallest edge that does not form a cycle.
  - Continue until the MST contains $|V| - 1$ edges.



- Edges added to MST:
  - (B, C) = 1
  - (D, F) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (B, D) = 3
  - (A, B) = 3
  - (E, G) = 4

- MST: (B, C), (D, F), (C, E), (C, H), (B, D) ,(A, B), (E, G)
  - Weight $= 1 + 1 + 2 + 2 + 3 + 3 + 4 = 16$

Kruskal's Algorithm: Correctness

- Assume that Kruskal's Algorithm does not produce a Minimum Spanning Tree (MST).
- Let T be the MST produced by Kruskal's Algorithm, and let T^* be the true MST.
- Assume that the total weight of T is greater than the total weight of T^*.
- **Edge Addition:**
  - Kruskal's Algorithm adds edges in increasing order of weight.
  - During the construction, if an edge e is added to T but not to T^*, replacing any edge in T^* with e would result in a spanning tree with equal or lesser weight.

Kruskal's Algorithm: Correctness

- **Cycle Formation:**
  - The Union-Find data structure ensures that adding e does not form a cycle.
  - Therefore, every edge added maintains the acyclic property of T.
- **Contradiction:**
  - Since T and T^* are both spanning trees and T is constructed by adding edges in increasing order of weight, it cannot have a greater total weight than T^*.
  - Hence, the assumption that Kruskal's Algorithm does not produce an MST is false.

Kruskal's Algorithm: Time Complexity

- Sorting the edges takes $O(E \log E)$ time.
- Each Union and Find operation takes $O(\log V)$ time.
- Overall time complexity is $O(E \log E + E \log V)$, which simplifies to $O(E \log V)$ for connected graphs.

Prim's Algorithm: Idea

- **Greedy Choice Property**:
  - At each step, the algorithm makes a locally optimal choice by selecting the smallest weight edge that connects a vertex in the MST to a vertex outside the MST.
  - This greedy choice ensures that the MST grows incrementally by always adding the minimum possible edge.
- **Optimal Substructure**:
  - Any subset of edges that forms an MST for a subgraph of the original graph is part of the MST for the entire graph.
  - This property allows the algorithm to build the MST incrementally, ensuring optimality at each step.
- Prim's Algorithm constructs the MST by starting from an arbitrary vertex and growing the MST one edge at a time.
- At each step, the algorithm selects the smallest weight edge that connects a vertex in the MST to a vertex outside the MST.
- Uses a priority queue (often implemented with a binary heap or Fibonacci heap) to efficiently select the next edge to add.

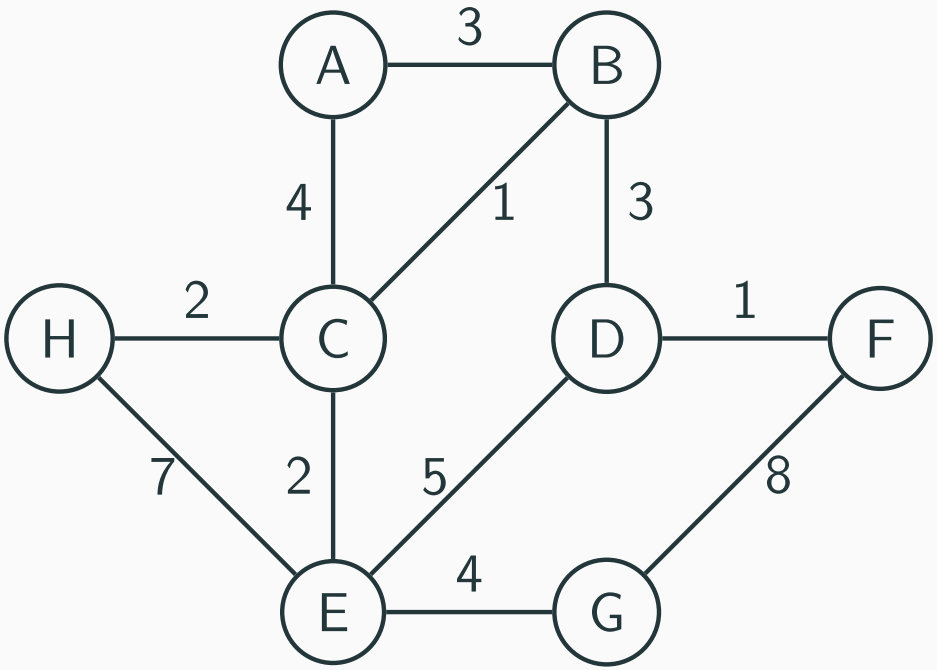
Prim's Algorithm: Pseudocode

Algorithm 10 Prim's Algorithm

1: **Input:** Graph $G = (V, E)$, starting vertex s
2: **Output:** Minimum Spanning Tree T
3: Initialize distances $d[v] \leftarrow \infty$ for all $v \in V$ except $d[s] \leftarrow 0$
4: Initialize an empty priority queue Q
5: Insert s into Q with priority 0
6: **while** Q is not empty **do**
7: Extract vertex u with the smallest distance $d[u]$ from Q
8: **for** each neighbor v of u **do**
9: **if** v is not in the MST and $w(u, v) < d[v]$ **then**
10: $d[v] \leftarrow w(u, v)$
11: Insert v into Q with priority $d[v]$
12: **end if**
13: **end for**
14: **end while**

Prim's Algorithm: Example

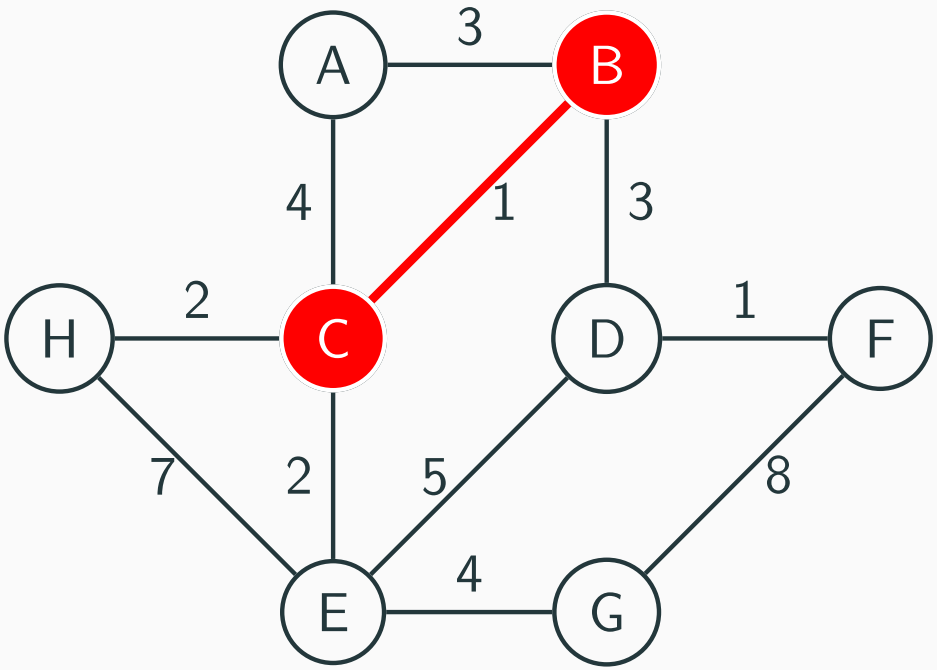
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:

Prim's Algorithm: Example

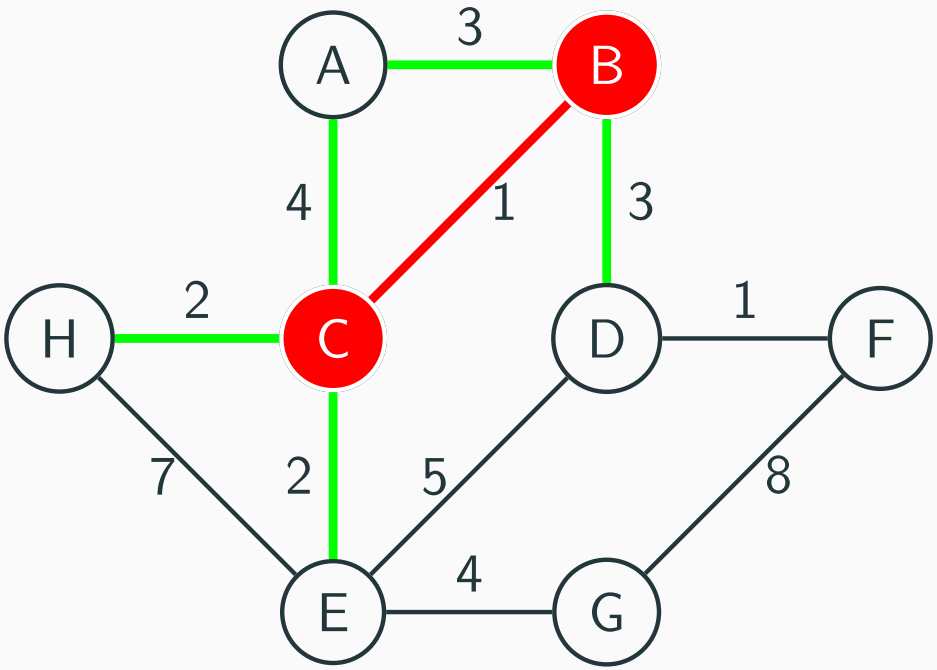
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1

Prim's Algorithm: Example

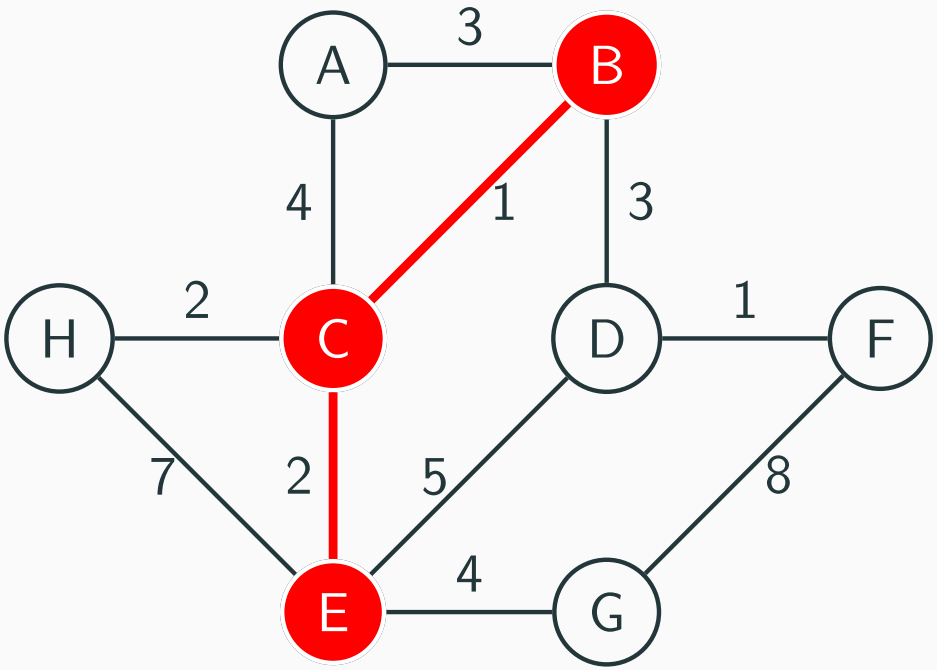
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1

Prim's Algorithm: Example

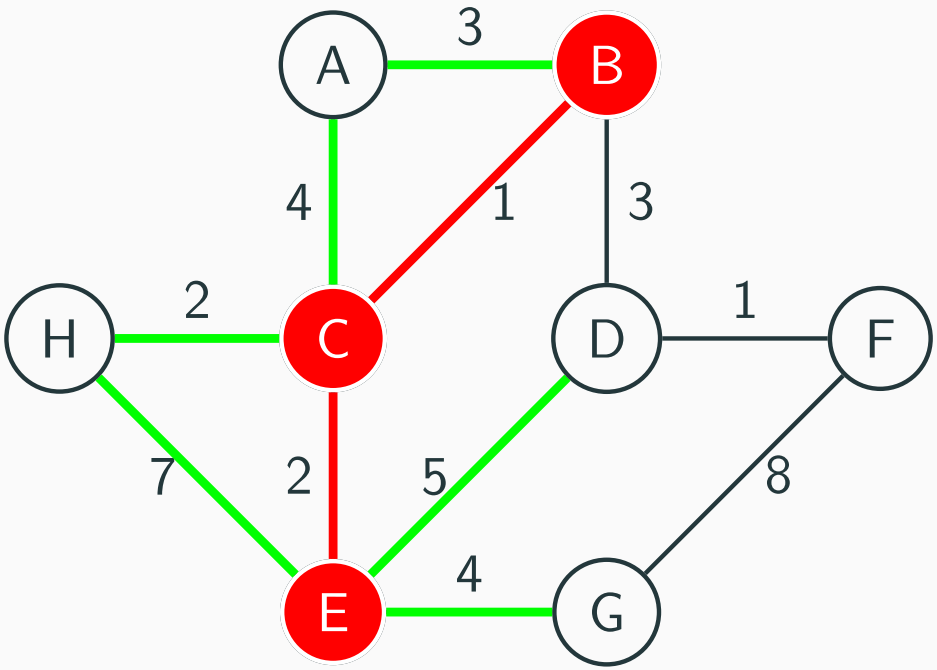
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2

Prim's Algorithm: Example

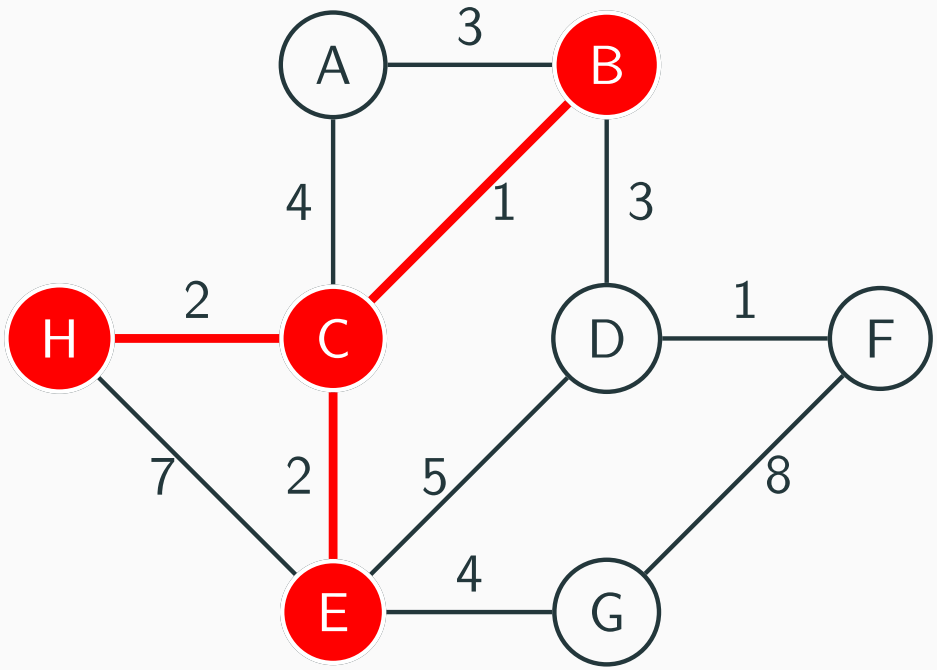
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2

Prim's Algorithm: Example

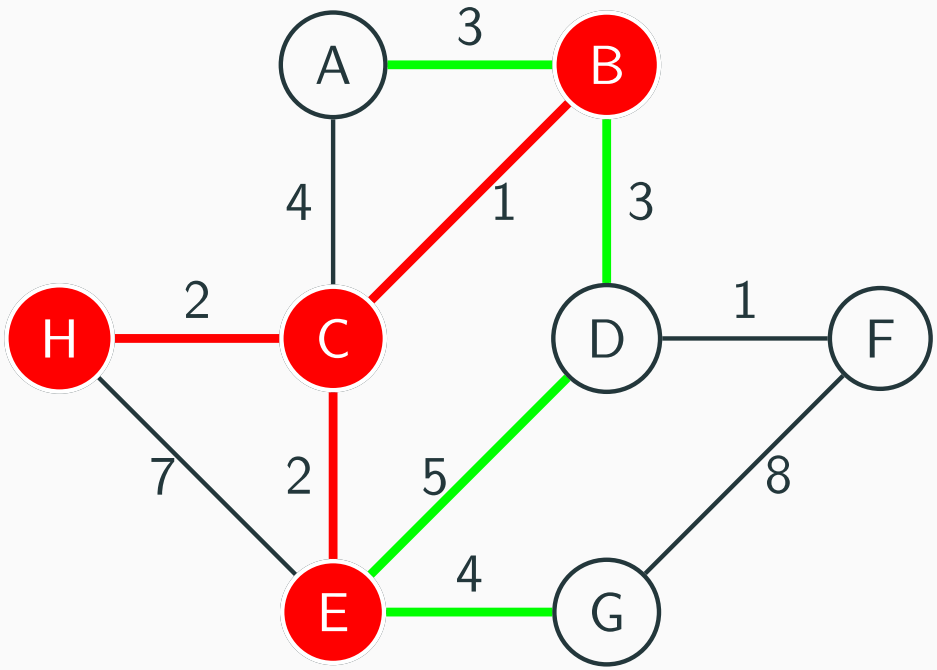
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2

Prim's Algorithm: Example

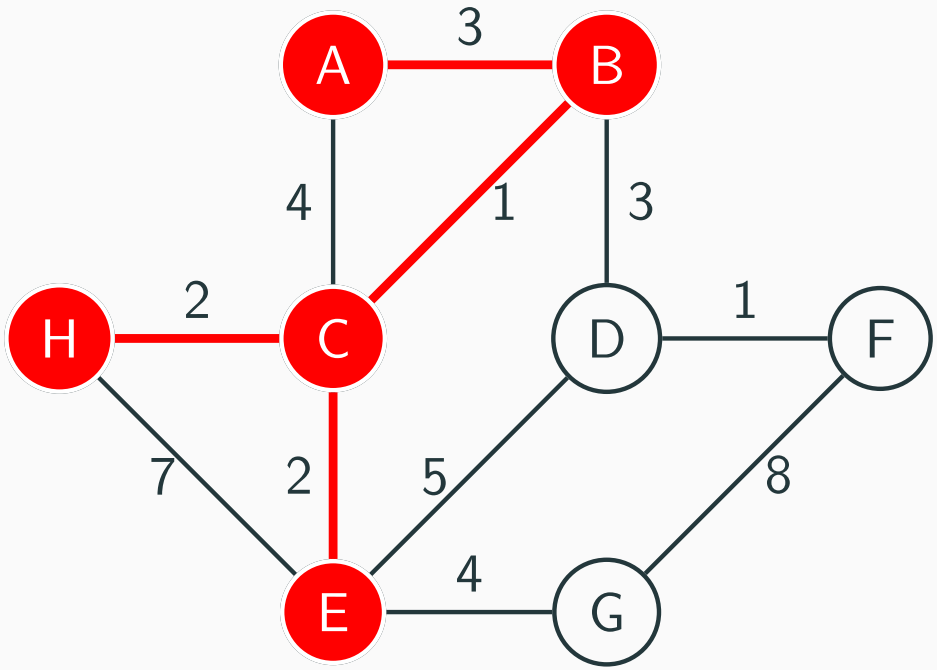
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2

Prim's Algorithm: Example

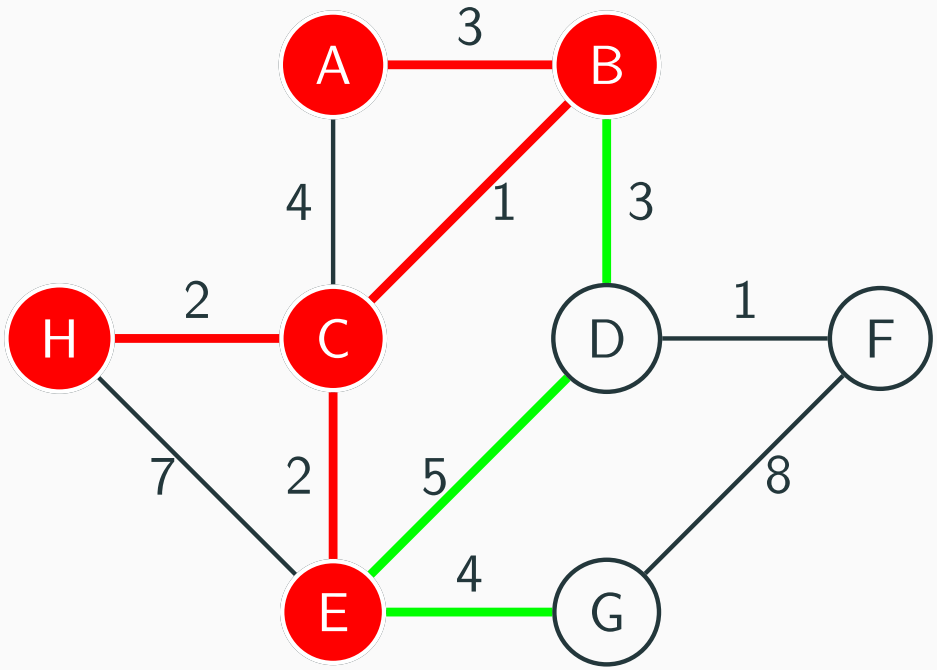
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (A, B) = 3

Prim's Algorithm: Example

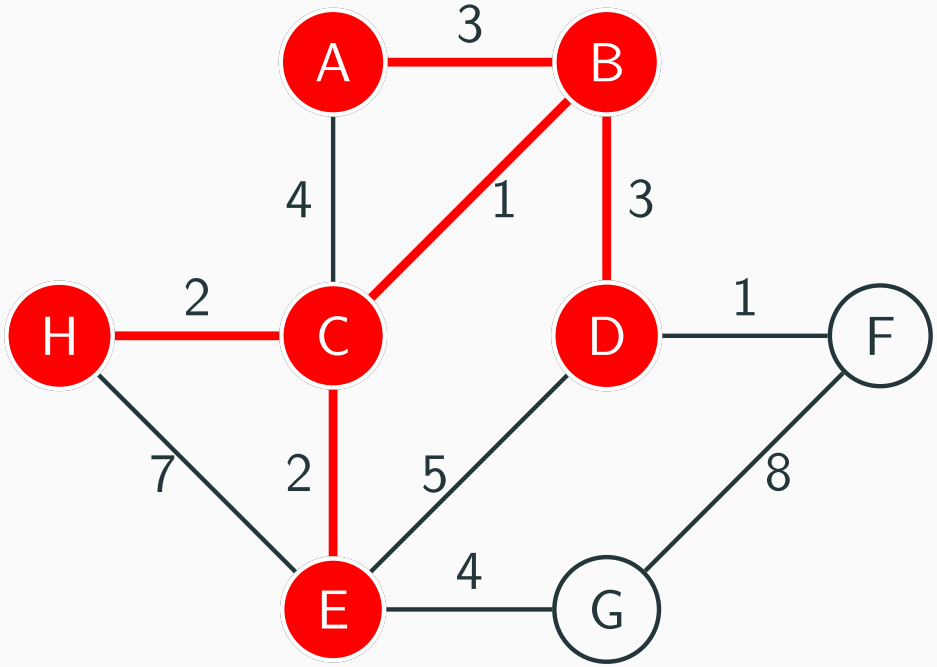
- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (A, B) = 3

Prim's Algorithm: Example

- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.

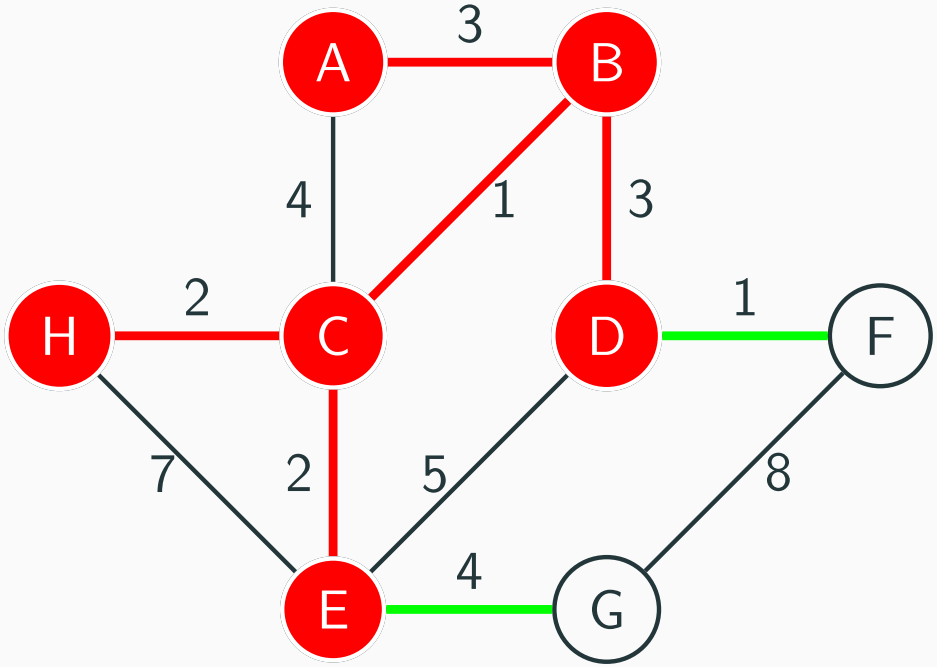


- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (A, B) = 3
  - (B, D) = 3

- MST: (B, C), (C, E), (C,H), (A, B), (B, D), (D, F), (E, G)
  - Weight = $1 + 2 + 2 + 3 + 3 + 1 + 4 = 16$

Prim's Algorithm: Example

- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.

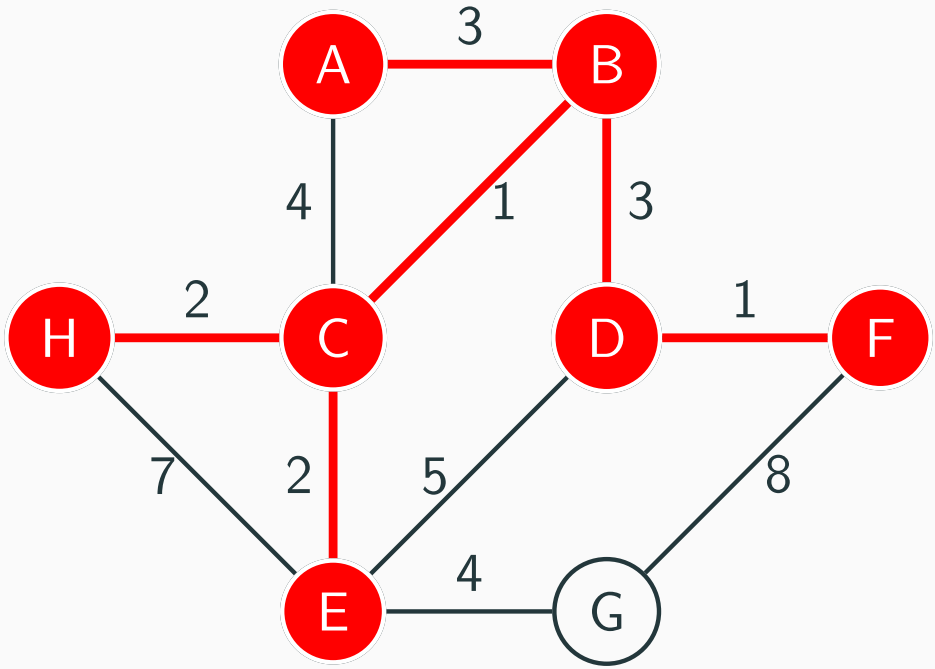


- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (A, B) = 3
  - (B, D) = 3

- MST: (B, C), (C, E), (C,H), (A, B), (B, D), (D, F), (E, G)
  - Weight = $1 + 2 + 2 + 3 + 3 + 1 + 4 = 16$

Prim's Algorithm: Example

- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.

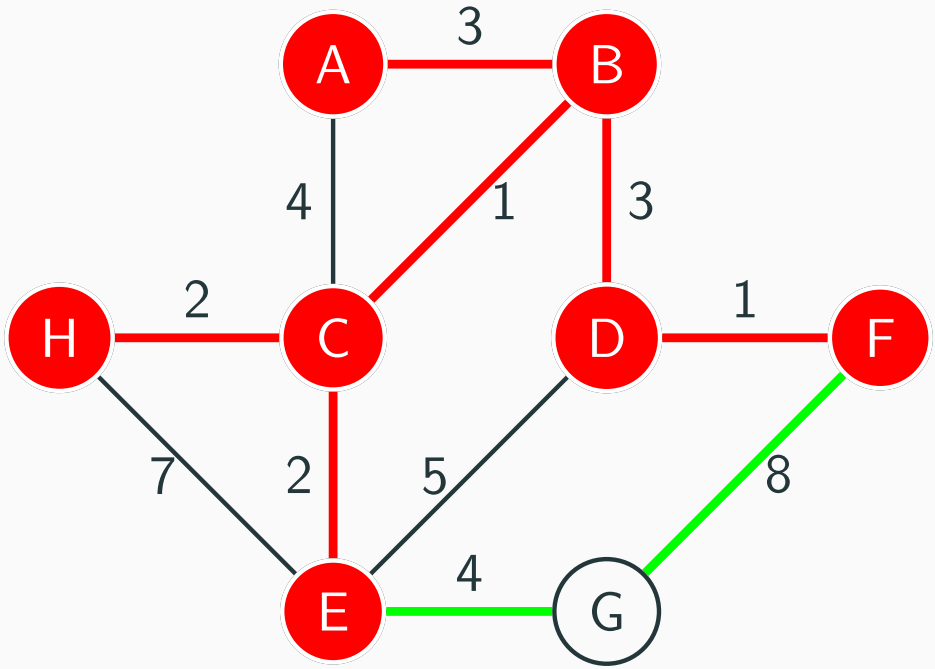


- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (A, B) = 3
  - (B, D) = 3
  - (D, F) = 1

- MST: (B, C), (C, E), (C,H), (A, B), (B, D), (D, F), (E, G)
  - Weight = $1 + 2 + 2 + 3 + 3 + 1 + 4 = 16$

Prim's Algorithm: Example

- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.

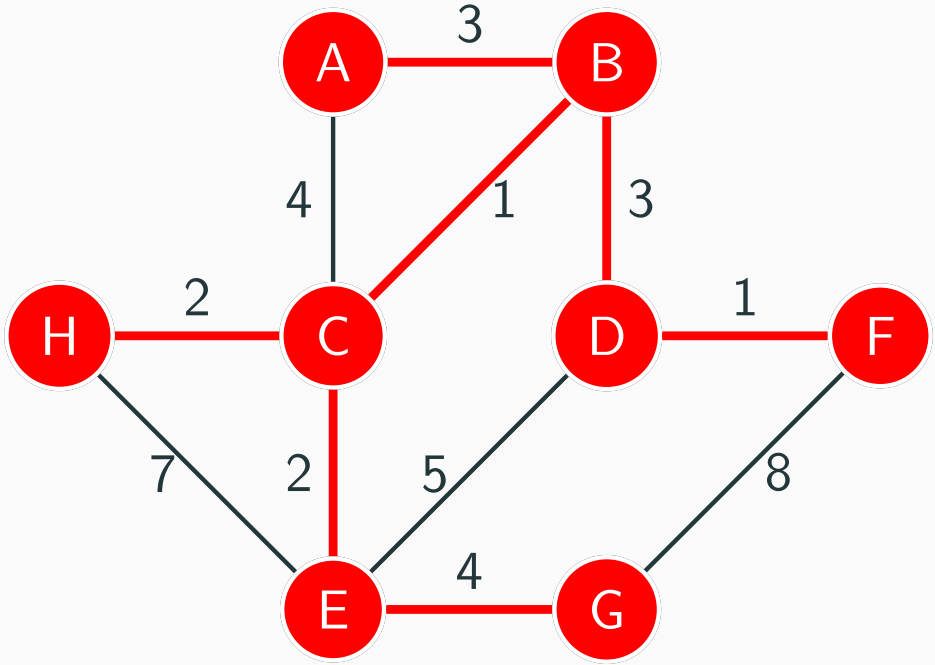


- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (A, B) = 3
  - (B, D) = 3
  - (D, F) = 1

- MST: (B, C), (C, E), (C,H), (A, B), (B, D), (D, F), (E, G)
  - Weight = $1 + 2 + 2 + 3 + 3 + 1 + 4 = 16$

Prim's Algorithm: Example

- Start with an arbitrary vertex.
  - Add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
  - Continue until all vertices are included in the MST.



- Edges added to MST:
  - (B, C) = 1
  - (C, E) = 2
  - (C, H) = 2
  - (A, B) = 3
  - (B, D) = 3
  - (D, F) = 1
  - (E, G) = 4

- MST: (B, C), (C, E), (C,H), (A, B), (B, D), (D, F), (E, G)
  - Weight = 1 + 2 + 2 + 3 + 3 + 1 + 4 = 16

Prim's Algorithm: Correctness

- Assume that Prim's Algorithm does not produce a Minimum Spanning Tree (MST).
- Let T be the MST produced by Prim's Algorithm, and let T^* be the true MST.
- Assume that the total weight of T is greater than the total weight of T^*.
- **Edge Addition:**
  - Prim's Algorithm adds edges in increasing order of their weight, starting from an arbitrary vertex.
  - At each step, it selects the smallest edge that connects a vertex in T to a vertex outside T.
  - If an edge e is added to T but not to T^*, replacing any edge in T^* with e would result in a spanning tree with equal or lesser weight.

Prim's Algorithm: Correctness

- **Cycle Avoidance:**
  - Prim's Algorithm ensures that each added edge connects a vertex inside the MST to one outside, maintaining the acyclic property of T.
  - Since T starts with a single vertex and grows by adding one edge at a time, no cycles can form.
- **Contradiction:**
  - Since T and T^* are both spanning trees and T is constructed by adding edges in increasing order of weight, it cannot have a greater total weight than T^*.
  - Hence, the assumption that Prim's Algorithm does not produce an MST is false.
- **Conclusion:**
  - Prim's Algorithm correctly produces an MST.

Prim's Algorithm: Time Complexity

- Using a binary heap, the time complexity is $O((V + E) \log V)$.
- Using a Fibonacci heap, the time complexity is $O(E + V \log V)$.
- For dense graphs, the time complexity is dominated by the number of edges.

Comparison of Kruskal's and Prim's Algorithms

- **Kruskal's Algorithm**:
  - Constructs the MST by adding edges in increasing order of weight.
  - At each step, the edge added does not form a cycle with the previously added edges.
  - Uses a Disjoint Set data structure (Union-Find) to manage connected components.
  - Suitable for sparse graphs.
  - Time Complexity: $O(E \log E)$ (or $O(E \log V)$ if implemented with efficient union-find operations).

Comparison of Kruskal's and Prim's Algorithms

- **Prim's Algorithm**:
  - Constructs the MST by starting from an arbitrary vertex and growing the MST one edge at a time.
  - At each step, selects the smallest weight edge that connects a vertex in the MST to a vertex outside the MST.
  - Uses a priority queue (binary heap or Fibonacci heap) for efficient edge selection.
  - Suitable for dense graphs.
  - Time Complexity: $O((V + E) \log V)$ with a binary heap, $O(E + V \log V)$ with a Fibonacci heap.

Comparison of Kruskal's and Prim's Algorithms

- **Similarities**:
  - Both algorithms use a greedy approach to construct the MST.
  - Both algorithms ensure the MST has the minimum total weight.
  - Both algorithms maintain the properties of acyclic and connectivity.
- **Differences**:
  - **Approach**:
    - Kruskal's: Edge-based, adds edges in increasing weight.
    - Prim's: Vertex-based, grows MST from a starting vertex.
  - **Data Structures**:
    - Kruskal's: Uses Union-Find for cycle detection.
    - Prim's: Uses a priority queue for selecting the next edge.
  - **Efficiency**:
    - Kruskal's: More efficient for sparse graphs.
    - Prim's: More efficient for dense graphs.

Conclusion

Greedy Algorithms: Summary

- A greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece.
- Greedy algorithms are best suited for problems where:
  - The problem exhibits the greedy choice property.
  - The problem has optimal substructure.
  - A clear local optimum can be identified that leads to a global optimum.
  - Example problems include finding the minimum spanning tree, shortest paths in graphs, and constructing optimal codes.

Today's Session: Summary

- We discussed scheduling problems
  - Interval Scheduling
  - Interval Partitioning
  - Minimizing Lateness
- Fractional knapsack
- We presented greedy algorithms to the problems
  - We showed these algorithms result in correct output
  - We showed the greedy properties of each
  - We showed the running time of these algorithms

Today's Session:Summary

- Greedy algorithms for graph
  - Single Source shortest path with Dijkstra Algorithm
  - Minimum Spanning Tree
    - Kruskal's algorithm
    - Prim's algorithms
- Example applications of these problems/algorithms

Resource

- Shortest Path Problem
  - General Idea: https://www.youtube.com/watch?v=Aa2sqUhIn-E
  - Dijkstra's Algorithm:
    - https://www.youtube.com/watch?v=pVfj6mxhdMw
    - https://www.youtube.com/watch?v=2E7MmKv0Y24
    - https://www.youtube.com/watch?v=HXhJIDB6EcM
- MST:
  - https://www.youtube.com/watch?v=Yldkh0aOEcg
  - https://www.youtube.com/watch?v=tKwnms5iRBU