# Algorithms

## Divide and Conquer

Dr. Mudassir Shabbir

LUMS University

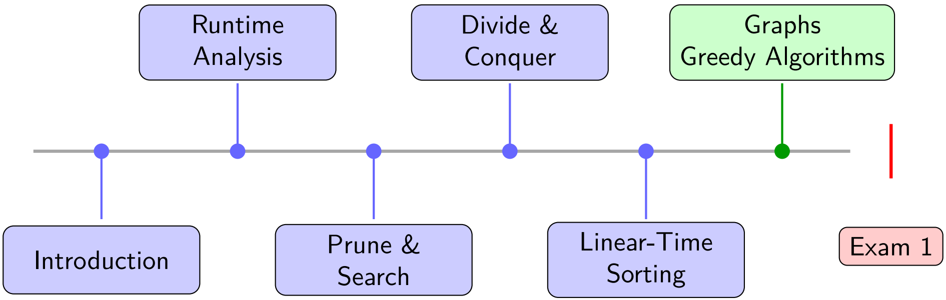
February 16, 2026

# Announcements

- Midterm Exam/Long Quiz 1 on Sun 02/22, 2026 noon - 1:45p.
- Homework 2 (no-submission practice problems) is available - no submission required.

**My Office Hours: Mon/Wed 12-1 PM.**

# Course Recap & What's Next



# Recap: How to Write an Algorithm in the Exam

- *Input:* Describe the input format and what the algorithm receives.
- *Output:* Describe the output format and what the algorithm should produce.
- **Algorithm:** Write your algorithm in **plain English**, bullet points, or pseudocode.
- **Correctness:** Explain why your algorithm is correct, with a proof or argument.
- **Runtime Analysis:** Analyze the runtime of your algorithm.

# Greedy Algorithms

# What is a Greedy Algorithm?

**Definition**

A **greedy algorithm** is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit according to some locally optimal criterion.

**Key characteristics:**

- Makes a sequence of choices $c_1, c_2, \ldots, c_k$
- Each choice $c_i$ is *locally optimal* given choices $c_1, \ldots, c_{i-1}$
- Never reconsiders previous choices (no backtracking)
- Hope: local optimality $\Rightarrow$ global optimality

Question: When does this work?

# Greedy vs. Other Paradigms

| Paradigm | Strategy |
|---|---|
| **Greedy** | Make locally optimal choice at each step; never backtrack |
| **Divide & Conquer** | Break into independent subproblems; combine solutions |
| **Backtracking** | Try possibilities; undo when dead end reached |
| **Dynamic Programming** | Solve overlapping subproblems; build optimal solution from optimal substructure |

**Greedy advantages:**

- Often simple and efficient

**Greedy challenges:**

- Many problems where greedy fails

# Greedy Choice Property

### Definition (Greedy Choice Property)

A problem exhibits the **greedy choice property** if a globally optimal solution can be arrived at by making locally optimal (greedy) choices.

**To prove greedy correctness, we typically show:**

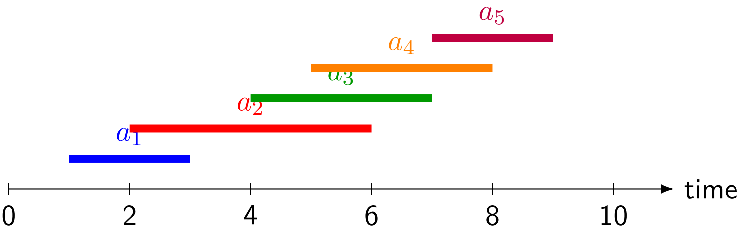
1. **Greedy Choice Property:** There exists an optimal solution that includes the greedy choice
2. **Optimal Substructure:** After making the greedy choice, the remaining problem is a smaller instance of the same problem

### Proof Strategy

Use *exchange argument*: Start with any optimal solution $OPT$. If it doesn't include the greedy choice, show we can modify $OPT$ to include it without worsening the objective.

# Example: Activity Selection Problem

**Problem:** Given $n$ activities with start times $s_i$ and finish times $f_i$, select maximum number of non-overlapping activities.




**Greedy strategy:** Select activity with earliest finish time that doesn't conflict with previously selected activities.

**Why earliest finish time?** Leaves the most room for subsequent activities.

# Activity Selection: Correctness Proof

### Theorem

*The greedy algorithm that selects activities by earliest finish time produces an optimal solution to the activity selection problem.*

### Proof sketch.

Let $a_1$ be the activity with earliest finish time. We show there exists an optimal solution containing $a_1$.

Let $OPT = \{b_1, b_2, \ldots, b_k\}$ be any optimal solution with activities ordered by finish time.

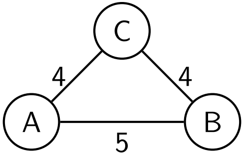
**Case 1:** If $b_1 = a_1$, done.
**Case 2:** If $b_1 \neq a_1$, then $f(a_1) \leq f(b_1)$ (by our choice).
We can replace $b_1$ with $a_1$ to get $OPT' = \{a_1, b_2, \ldots, b_k\}$.
Since $f(a_1) \leq f(b_1) < s(b_2)$, activities in $OPT'$ don't overlap.
Thus $|OPT'| = |OPT|$, so $OPT'$ is also optimal and contains $a_1$. $\square$ $\square$

## When Greedy Fails: Counterexamples

**Greedy doesn't always work!**

**Example 1: Longest Path**

- Problem: Find longest simple path in a graph
- Greedy: Always take the longest available edge
- Result: Can get stuck in local maximum




Greedy chooses edge $(A, B)$ with weight 5, but optimal path is $A \to C \to B$ with length 8.

**Example 2: 0-1 Knapsack**

# General Framework: Proving Greedy Correctness

**Step-by-step approach:**

1. **Define the problem formally**
   - Input, output, objective function
2. **Specify the greedy choice**
   - What criterion determines the "best" next choice?
3. **Prove the greedy choice property**
   - Exchange argument: show any optimal solution can be modified to include the greedy choice
4. **Prove optimal substructure**
   - After the greedy choice, remaining problem is smaller instance
   - Optimal solution to original $=$ greedy choice $+$ optimal solution to subproblem

# Common Problems Solved by Greedy

1. **Activity Selection** (earliest finish time)
2. **Fractional Knapsack** (value/weight ratio)
3. **Huffman Coding** (merge lowest frequency pairs)
4. **Minimum Spanning Tree**
   - Kruskal's algorithm (sort edges by weight)
   - Prim's algorithm (grow tree from starting vertex)
5. **Dijkstra's Shortest Path** (minimum distance vertex)
6. **Interval Scheduling**
7. **Job Sequencing with Deadlines**

**Next Up: Kruskal's Algorithm**

We'll apply these greedy principles to find minimum spanning trees, proving correctness via the *cut property*.

# Why Minimum Spanning Trees? (Historical Motivation)

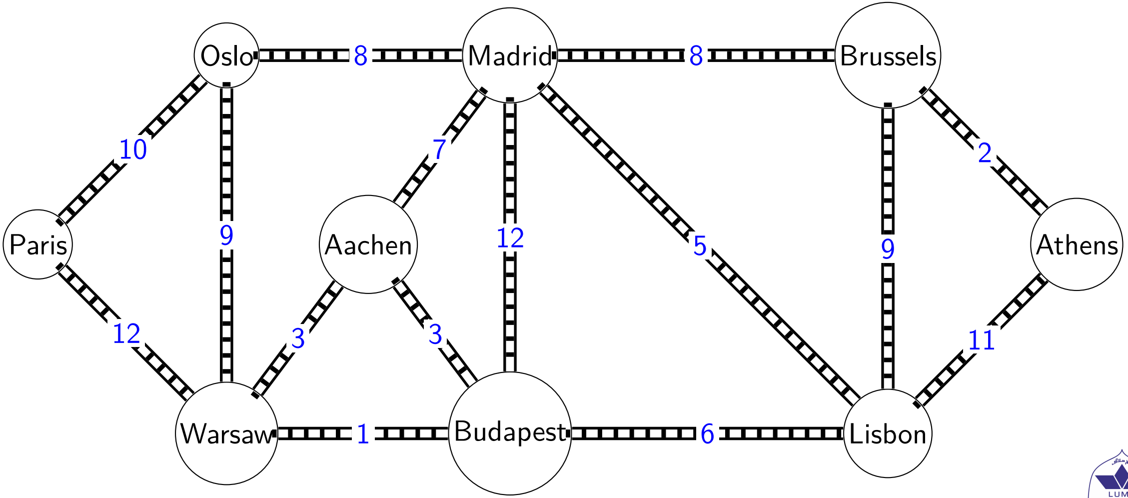
- During World War II, the Allies controlled railway infrastructure across Europe.
- Maintaining rail lines was expensive: fuel, guards, repairs, signaling.
- They needed a network that:
  - keeps all cities connected,
  - minimizes total maintenance cost.
- Question: **Which tracks should remain open?**

# Why Minimum Spanning Trees? (Historical Motivation)

- During World War II, the Allies controlled railway infrastructure across Europe.
- Maintaining rail lines was expensive: fuel, guards, repairs, signaling.
- They needed a network that:
    - keeps all cities connected,
    - minimizes total maintenance cost.
- Question: **Which tracks should remain open?**

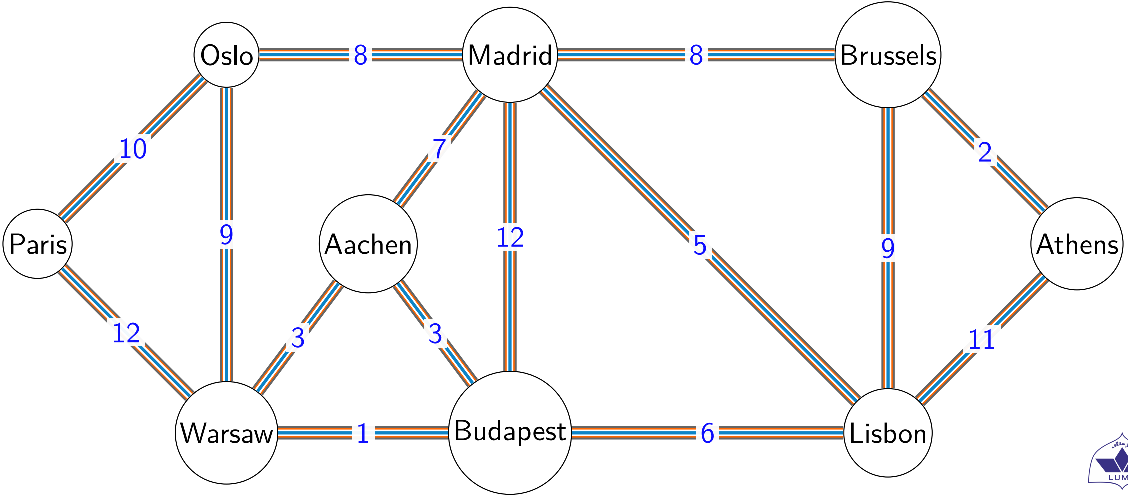
*This is exactly the Minimum Spanning Tree problem.*

# European rail network

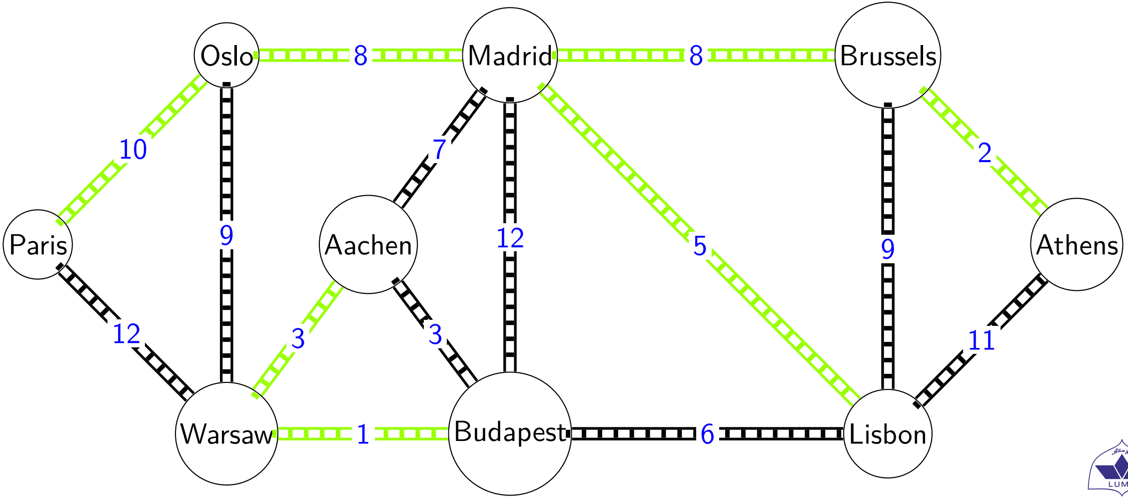



**Goal:** Keep cities connected while minimizing total cost.

# A Modern Version: Fiber Network Design



# Example Minimum Spanning Tree



# Minimum Spanning Tree (MST) Problem

- Input: Weighted graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}^+$.
- Output: Subset of edges $T \subseteq E$ such that:
  - $T$ connects all vertices (spanning),
  - $T$ contains no cycles (tree),
  - $T$ has minimum total weight: $\sum_{e \in T} w(e)$.

# How would we compute this?

- Idea: Try all subsets of edges.
- Keep only those that:
  - connect all vertices,
  - contain no cycle.
- Choose the cheapest among them.

# How would we compute this?

- Idea: Try all subsets of edges.
- Keep only those that:
  - connect all vertices,
  - contain no cycle.
- Choose the cheapest among them.

**Question:** How many subsets of edges are there?

# How would we compute this?

- Idea: Try all subsets of edges.
- Keep only those that:
  - connect all vertices,
  - contain no cycle.
- Choose the cheapest among them.

**Question:** How many subsets of edges are there?

$$2^{|E|}$$

# How would we compute this?

- Idea: Try all subsets of edges.
- Keep only those that:
  - connect all vertices,
  - contain no cycle.
- Choose the cheapest among them.

**Question:** How many subsets of edges are there?

$$2^{|E|}$$

**Conclusion:** Brute force is exponential — infeasible.

# Towards an Efficient Algorithm

- We want to build the tree gradually.
- We must repeatedly answer:
  - Do two vertices belong to the same component?
  - Can we add this edge safely?

## Towards an Efficient Algorithm

- We want to build the tree gradually.
- We must repeatedly answer:
  - Do two vertices belong to the same component?
  - Can we add this edge safely?
- We need a data structure for tracking components.

# Towards an Efficient Algorithm

- We want to build the tree gradually.
- We must repeatedly answer:
  - Do two vertices belong to the same component?
  - Can we add this edge safely?
- We need a data structure for tracking components.

**Union–Find (Disjoint Set Union)**

# Union–Find Operations

We maintain a partition of vertices into components.

- **Find(x)** — returns representative of x's component
- **Union(x,y)** — merges two components

# Union–Find Operations

We maintain a partition of vertices into components.

- **Find(x)** — returns representative of x's component
- **Union(x,y)** — merges two components

With path compression + union by rank:

$$\text{Amortized time per operation} = O(\alpha(n))$$

# Union–Find Operations

We maintain a partition of vertices into components.

- **Find(x)** — returns representative of x's component
- **Union(x,y)** — merges two components

With path compression + union by rank:

$$\text{Amortized time per operation } = O(\alpha(n))$$

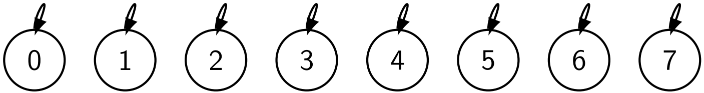
(Inverse Ackermann — practically constant)

# What is Union Find?

- Also known as **Disjoint Set Union (DSU)**
- Data structure that keeps track of elements partitioned into disjoint sets
- Two main operations:
  - `Find(x)`: Determine which set element $x$ belongs to
  - `Union(x, y)`: Merge the sets containing $x$ and $y$
- Applications: Kruskal's MST algorithm, dynamic connectivity, image segmentation

# Initial State: Disjoint Sets

Initially, each element is in its own set (parent of itself).

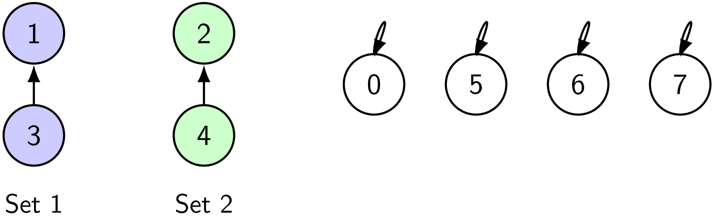



Elements: 0, 1, 2, 3, 4, 5, 6, 7

`parent[i] = i` for all elements

# Union Operation: Merge Sets

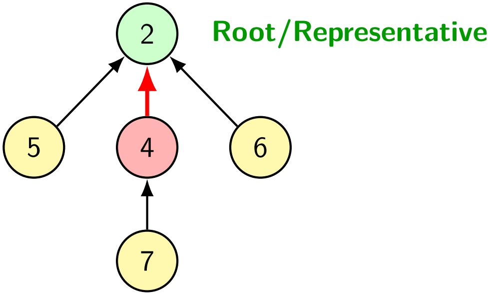
After `Union(1, 3)` and `Union(2, 4)`:




Each tree represents a disjoint set. The root is the representative.

# Find Operation: Path to Root

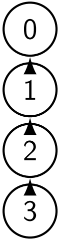
`Find(4)` traces parent pointers to find the root (representative):




`Find(4)` returns 2

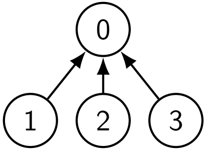
# Path Compression Optimization

**Before Path Compression**




Height $= 3$

**After Path Compression**




Height $= 1$

During `Find`, make all nodes on the path point directly to root!

# Basic Implementation

### Find with Path Compression

```
def find(x):
    if parent[x] != x:
        parent[x] = find(parent[x])  # Path compression
    return parent[x]
```

### Union

```
def union(x, y):
    root_x = find(x)
    root_y = find(y)
    if root_x != root_y:
        parent[root_x] = root_y
```

# Time Complexity of Union-Find

- **Without optimizations:** $O(n)$ per operation in worst case
- **With Union by Rank:** $O(\log n)$
- **With path compression + union by rank/size:**
  - Amortized $O(\alpha(n))$ where $\alpha$ is the inverse Ackermann function
  - For all practical purposes: $\alpha(n) \leq 4$

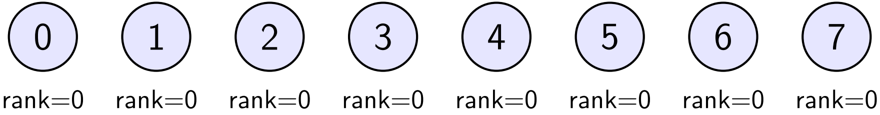
**Union by Rank**

Always attach the smaller tree under the root of the larger tree to keep trees flat.

# Step 0: Initial State

**Operation:** Initialization

**Elements:** 0, 1, 2, 3, 4, 5, 6, 7

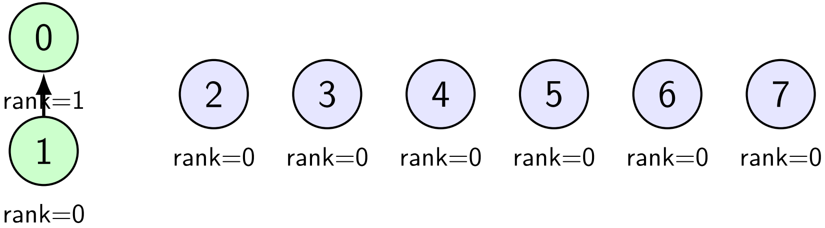



```
parent = [0, 1, 2, 3, 4, 5, 6, 7]
 rank = [0, 0, 0, 0, 0, 0, 0, 0]
```

# Step 1: Union(0, 1)

**Operation:** `Union(0, 1)`

**Action:** Both have rank 0, attach 1 under 0, increment rank of 0

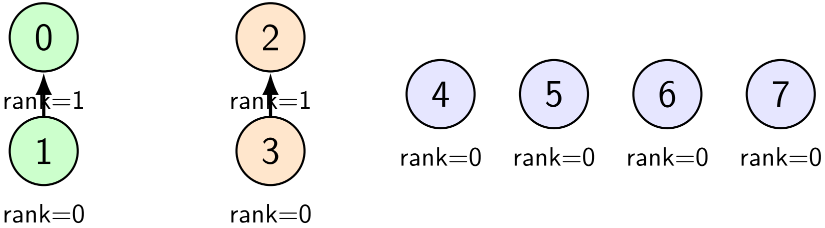



```
parent = [0, 0, 2, 3, 4, 5, 6, 7]
rank = [1, 0, 0, 0, 0, 0, 0, 0]
```

# Step 2: Union(2, 3)

**Operation:** `Union(2, 3)`

**Action:** Both have rank 0, attach 3 under 2, increment rank of 2

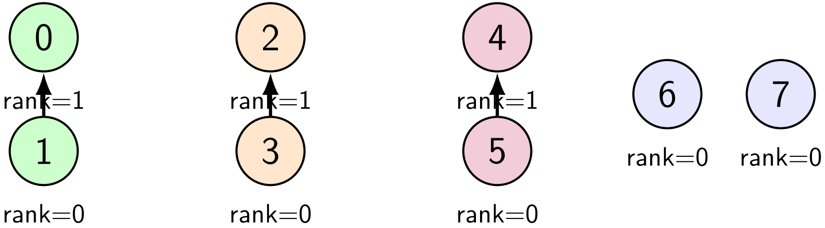



```
parent = [0, 0, 2, 2, 4, 5, 6, 7]
rank = [1, 0, 1, 0, 0, 0, 0, 0]
```

# Step 3: Union(4, 5)

**Operation:** `Union(4, 5)`

**Action:** Both have rank 0, attach 5 under 4, increment rank of 4

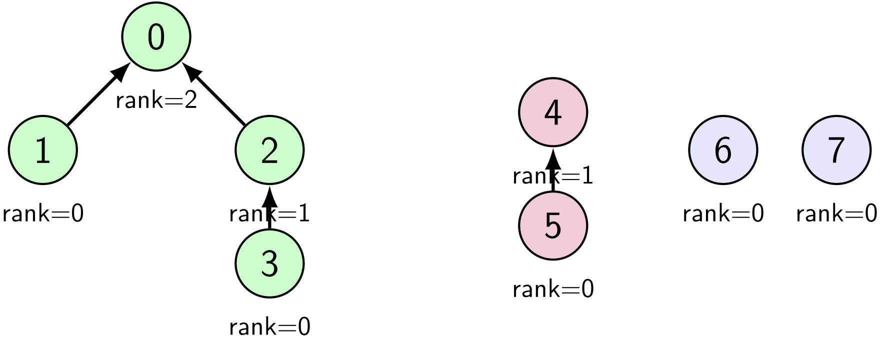



```
parent = [0, 0, 2, 2, 4, 4, 6, 7]
rank = [1, 0, 1, 0, 1, 0, 0, 0]
```

# Step 4: Union(0, 2)

**Operation:** `Union(0, 2)`

**Action:** Both have rank 1, attach 2 under 0, increment rank of 0

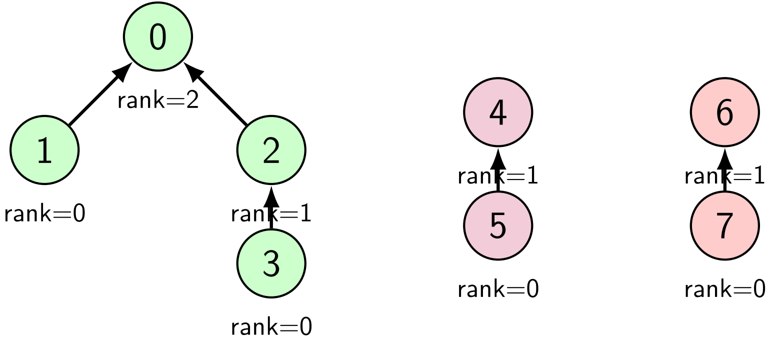



```
parent = [0, 0, 0, 2, 4, 4, 6, 7]
rank = [2, 0, 1, 0, 1, 0, 0, 0]
```

# Step 5: Union(6, 7)

**Operation:** `Union(6, 7)`

**Action:** Both have rank 0, attach 7 under 6, increment rank of 6



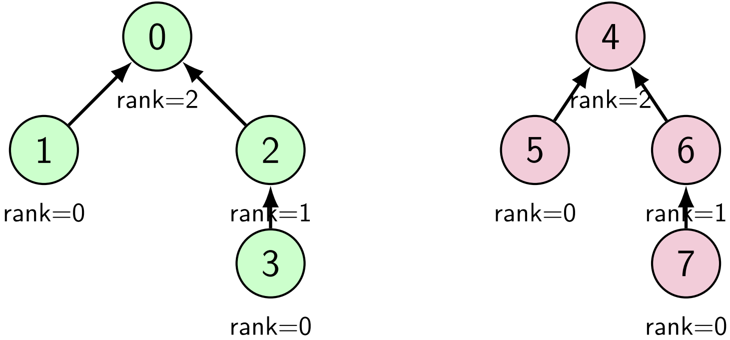

`parent = [0, 0, 0, 2, 4, 4, 6, 6]`

`rank = [2, 0, 1, 0, 1, 0, 1, 0]`

# Step 6: Union(4, 6)

**Operation:** `Union(4, 6)`

**Action:** Both have rank 1, attach 6 under 4, increment rank of 4

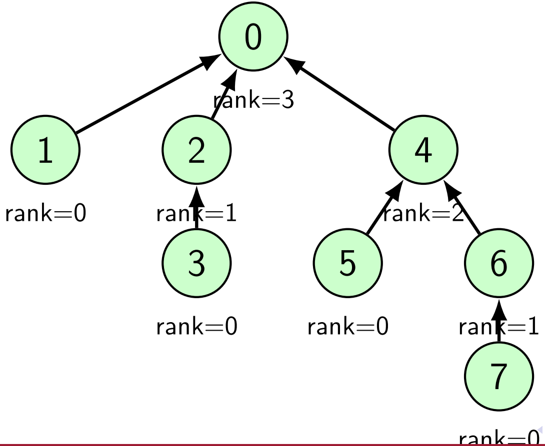



```
parent = [0, 0, 0, 2, 4, 4, 4, 6]
  rank = [2, 0, 1, 0, 2, 0, 1, 0]
```

## Step 7: Union(0, 4)

**Operation:** `Union(0, 4)`

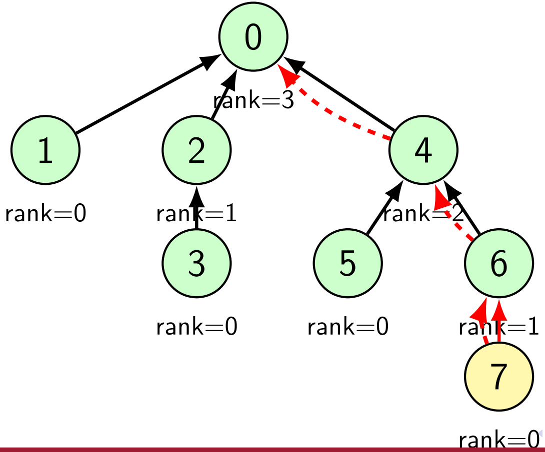
**Action:** Both have rank 2, attach 4 under 0, increment rank of 0



## Step 8: Find(7) with Path Compression

**Operation:** `Find(7)` - Returns 0

**Result:** All elements now in one connected set!



# Kruskal's Algorithm

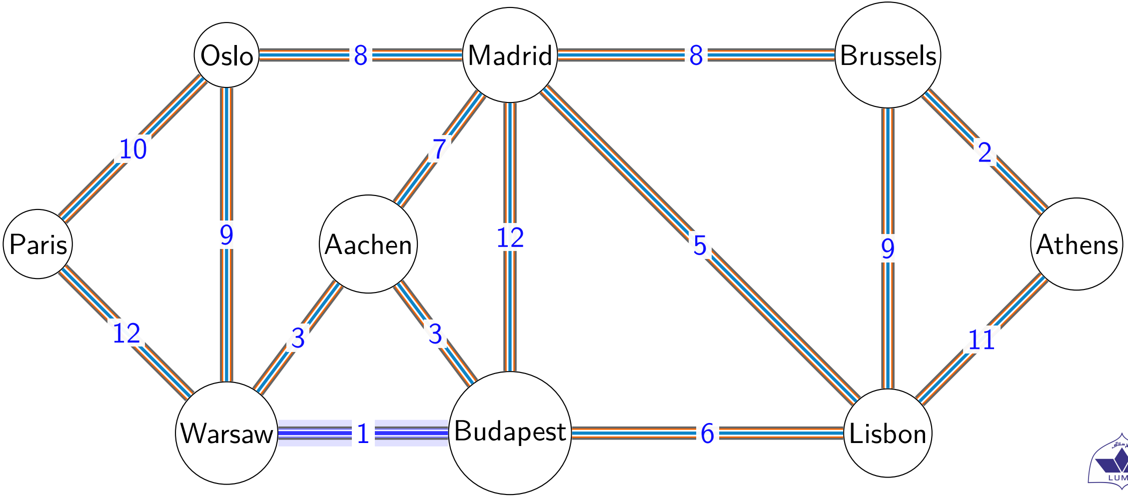
1. Sort edges by increasing weight
2. Start with empty tree
3. Process edges in order:
   - If edge connects different components $\rightarrow$ add it
   - Otherwise skip it
4. Stop after selecting $n - 1$ edges

# Kruskal's Algorithm

1. Sort edges by increasing weight
2. Start with empty tree
3. Process edges in order:
   - If edge connects different components $\rightarrow$ add it
   - Otherwise skip it
4. Stop after selecting $n-1$ edges

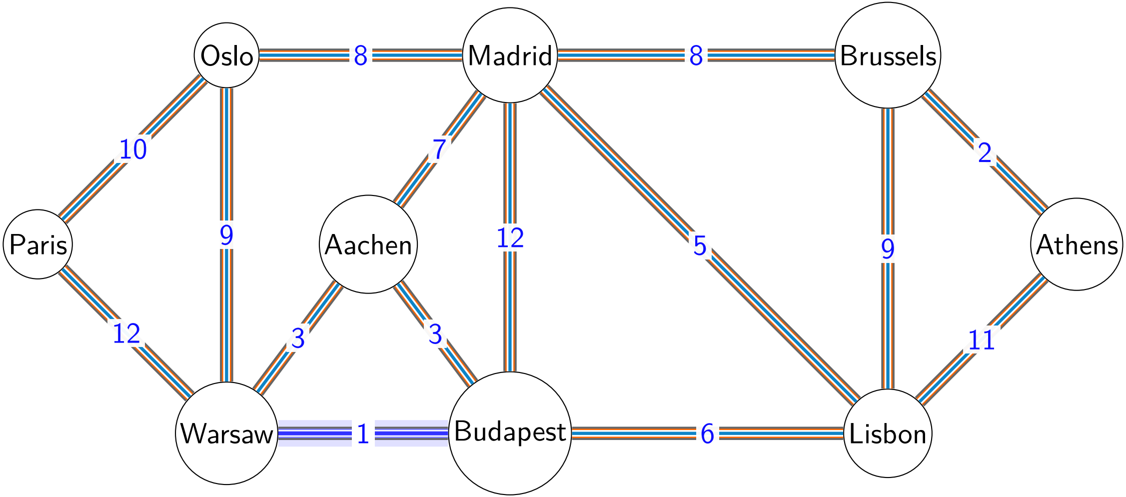
Greedy + Union–Find

# Kruskal in Action



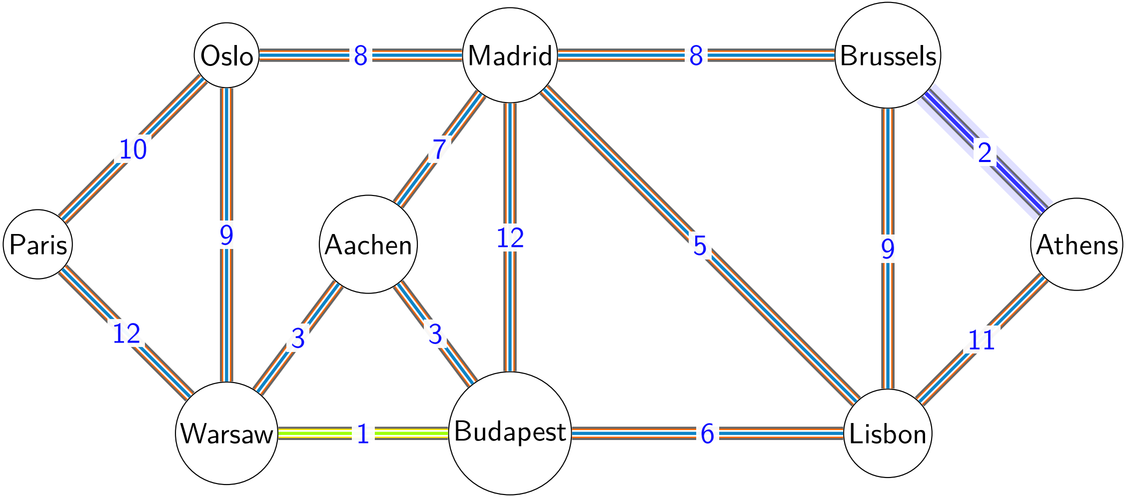
## Testing edge CF (weight 1)

Accept - No cycle created



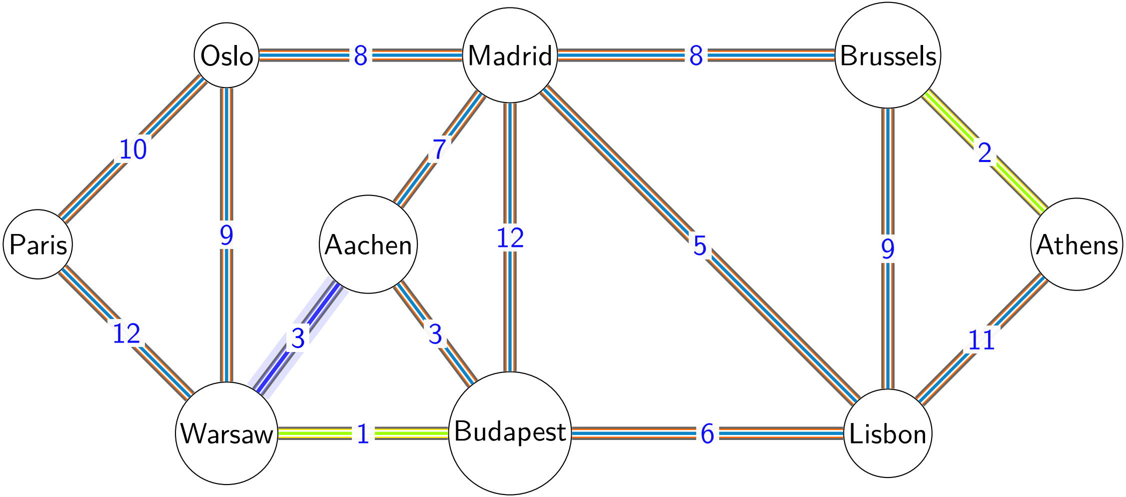
## Testing edge GI (weight 2)

Accept - No cycle created



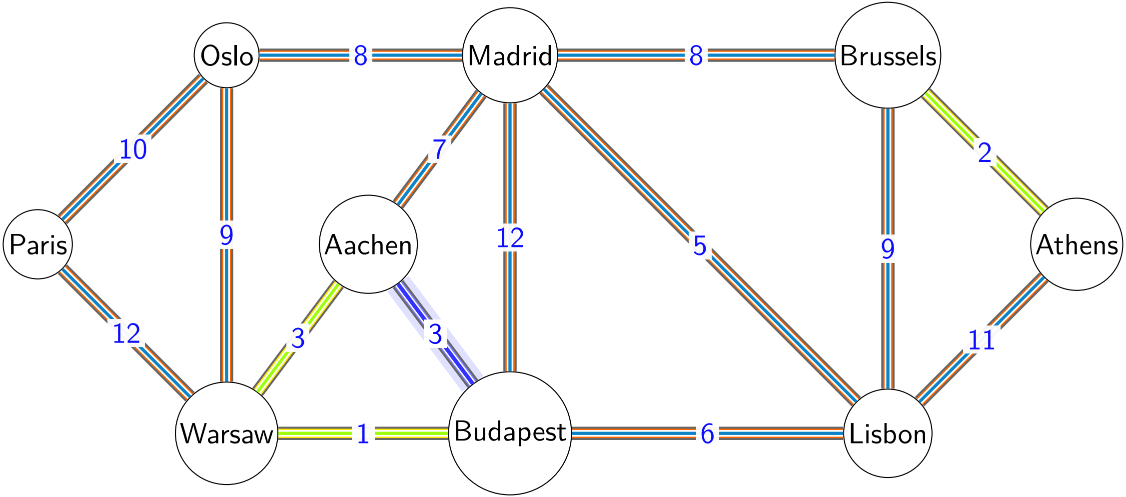
## Testing edge CE (weight 3)

Accept - No cycle created



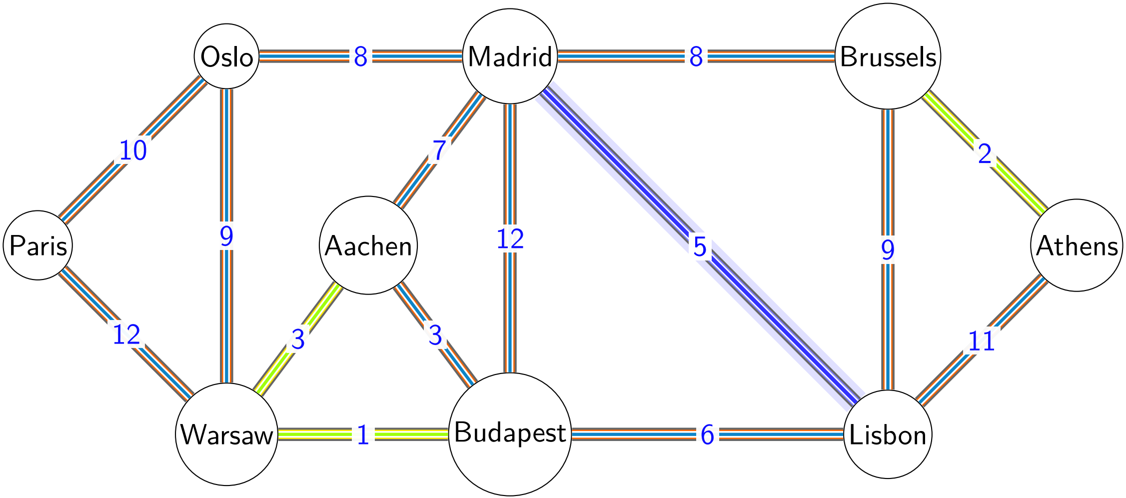
## Testing edge EF (weight 3)

Reject - Creates cycle: C-E-F-C



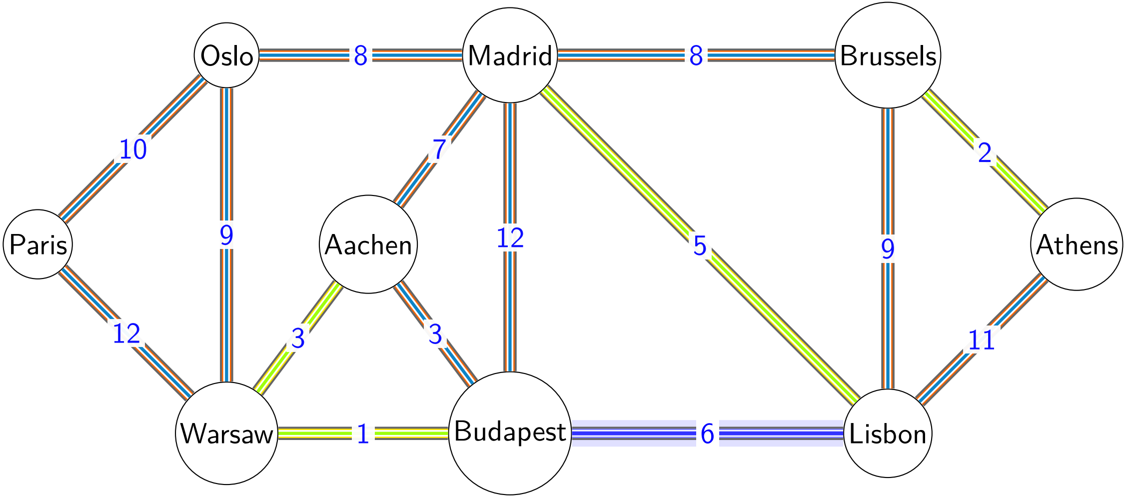
# Testing edge DH (weight 5)

Accept - No cycle created



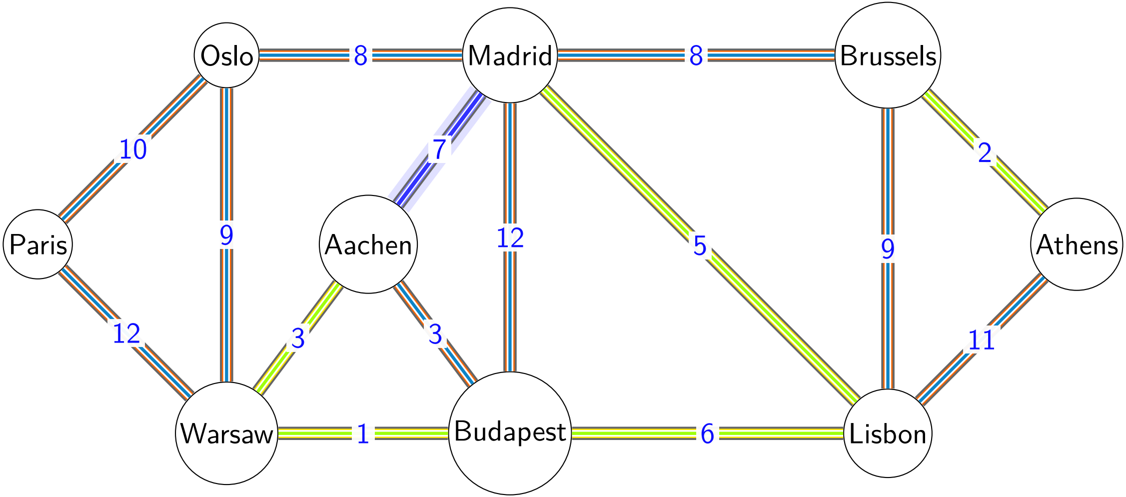
# Testing edge FH (weight 6)

Accept - Connects two components



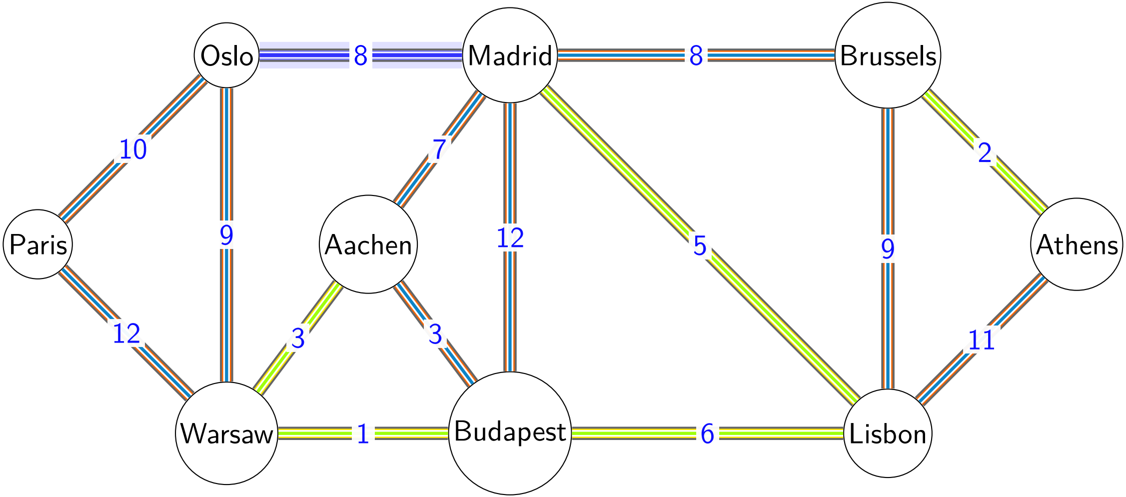
## Testing edge DE (weight 7)

Reject - Creates cycle: D-H-F-C-E-D



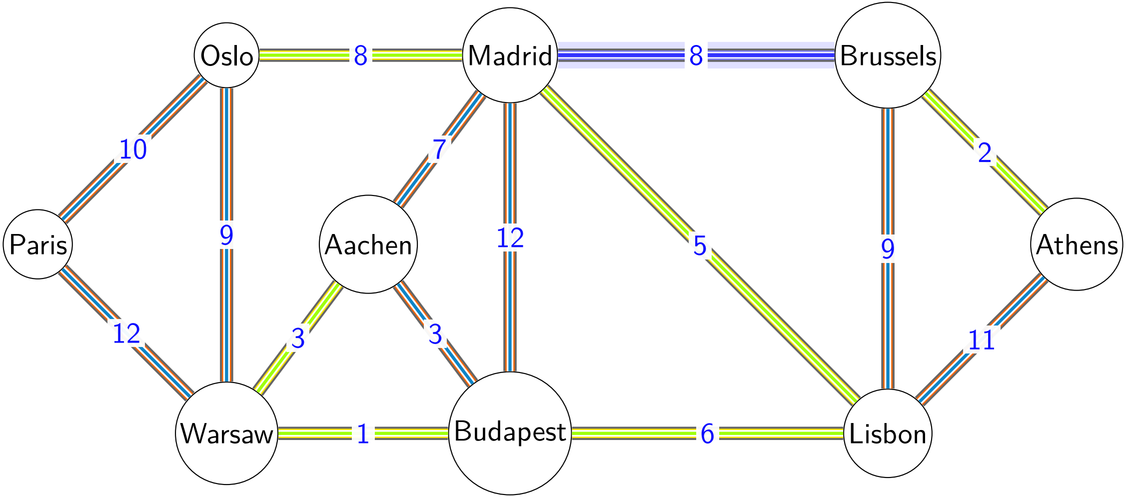
# Testing edge BD (weight 8)

Accept - No cycle created



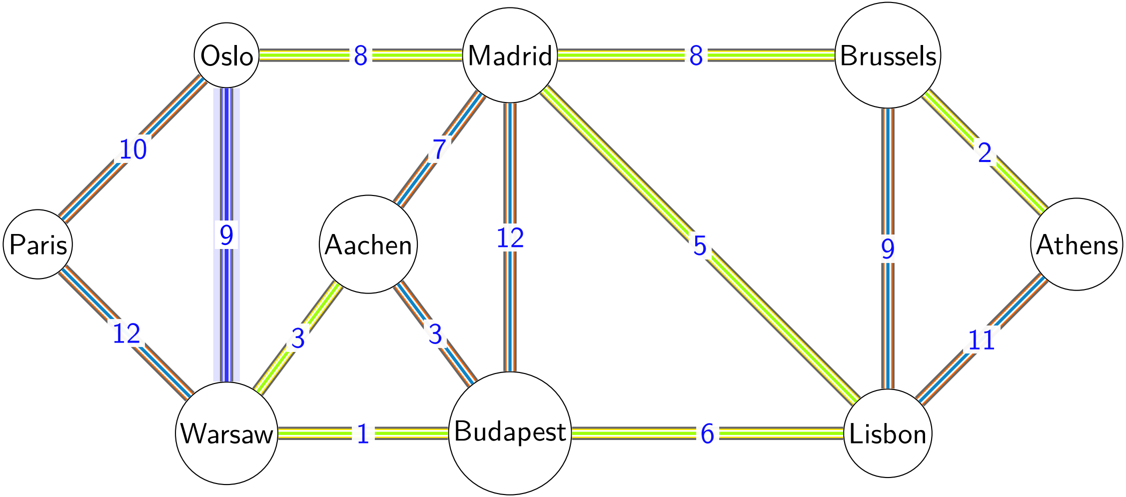
# Testing edge DG (weight 8)

Accept - Connects components



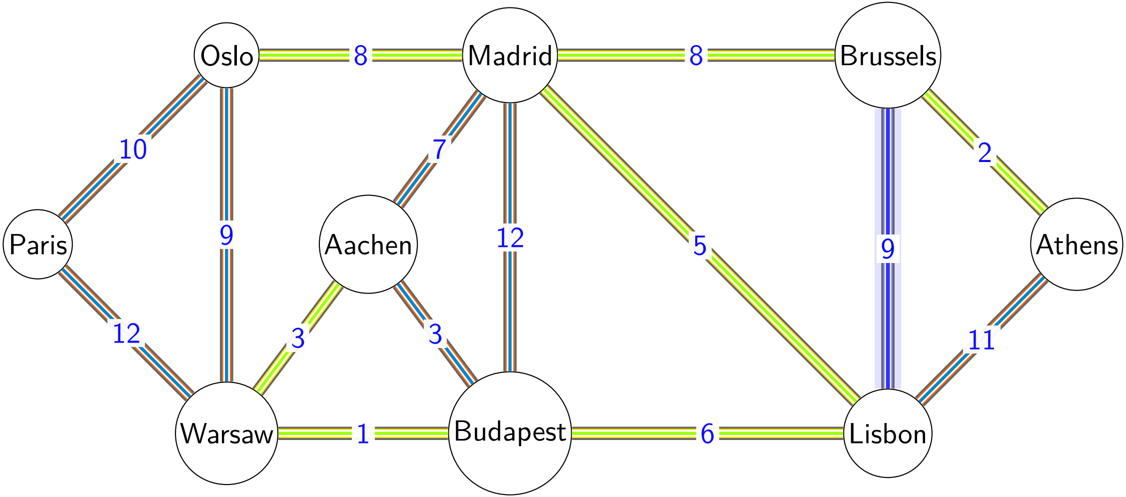
## Testing edge BC (weight 9)

Reject - Creates cycle



## Testing edge GH (weight 9)

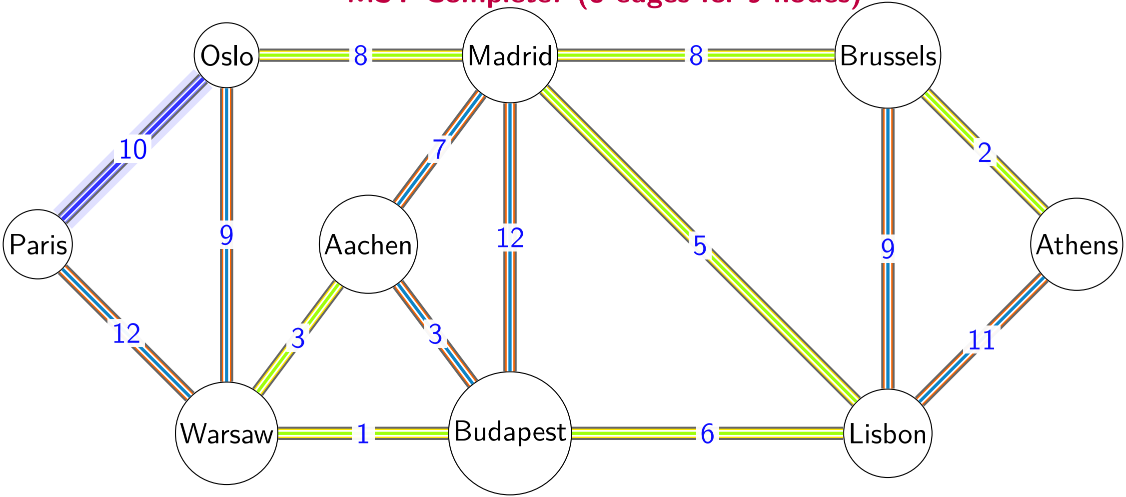
Reject - Creates cycle



## Testing edge AB (weight 10)

Accept - No cycle created

**MST Complete! (8 edges for 9 nodes)**



## Minimum Spanning Tree

**Total Weight: 1+2+3+5+6+8+8+10 = 43**