Lecture 7

Graphs III – Trees

Karl Bringmann, Dominic Zimmer

Saarland University

June, 2019

max planck institut
informatik

Midterm-Contest

- Date: 20.06.2020 at 10:00
  Meet in the Lecture Zoom at 10:00 sharp!
- Install & test the VM as **soon as possible ($\leq$ *Thursday*)!**
  - Installation instructions:
    `https://cms.sic.saarland/cp20/2/VM`
  - Test if you can boot & connect to the VPN
  - Test if you can submit a solution for a problem
  - Last chance to contact us in case of problems:
    Office Hour at Friday, 10:00
  - Before the contest: Delete all files you created (except your cheatsheet)
- Submit your Cheat-Sheet until **Thursday 23:59**
  - 100.000 characters, `.txt` file
  - Submission on your personal status page

Overview

- Graphs I: Traversals and Shortest Paths
- Graphs II: DFS Applications and Friends
- Graphs III: Trees
- Graphs IV: Flow Problems

Special Graphs

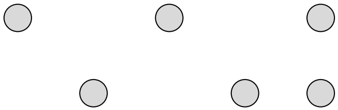
City Planning

After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?

Special Graphs

City Planning

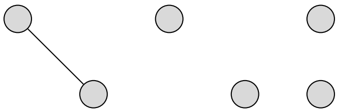
After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



Special Graphs

City Planning

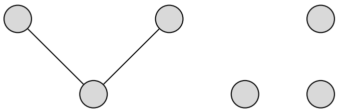
After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



Special Graphs

City Planning

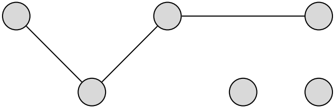
After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



Special Graphs

City Planning

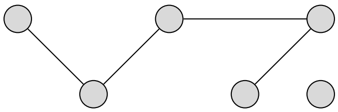
After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



Special Graphs

City Planning

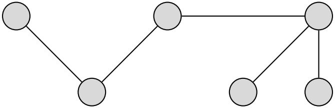
After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



Special Graphs

City Planning

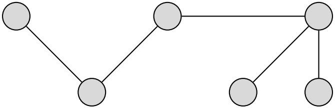
After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



Special Graphs

City Planning

After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



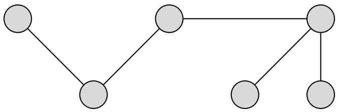
Insight

Between two vertices v and w in this graph, there exists exactly one unique path.

Special Graphs

City Planning

After constructing a city where every citizen could visit everyone else, you do some statistics. Your traffic authorities count that there are precisely $V - 1$ roads in your city.
How many unique paths are there?



Insight

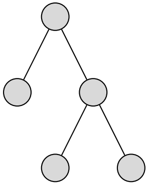
Between two vertices v and w in this graph, there exists exactly one unique path.

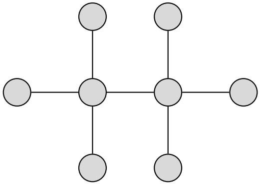
In total, there are $\binom{V}{2} = \frac{V \cdot (V-1)}{2}$ unique paths

Trees

Definition: Tree

A *Tree* is an acyclic, connected, undirected graph.



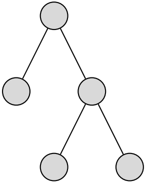


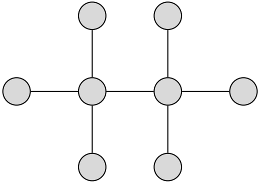
Trees

Definition: Tree

Let G be undirected. The following statements are equivalent

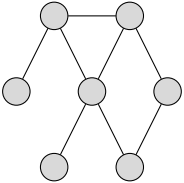
- G is called a tree
- G is acyclic and connected
- Between any two vertices of G, there is exactly one path
- G is acyclic and $E = V - 1$
- G is connected and $E = V - 1$
- G is minimally connected
- G is maximally acyclic

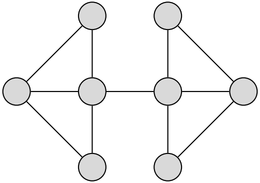




Definition: Spanning Tree

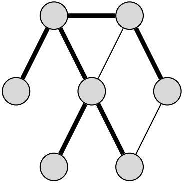
A subgraph S of G is called a *Spanning Tree*, if S is a tree and it contains every vertex of G.

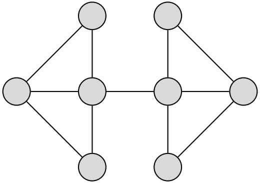




Definition: Spanning Tree

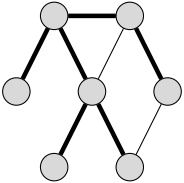
A subgraph S of G is called a *Spanning Tree*, if S is a tree and it contains every vertex of G.

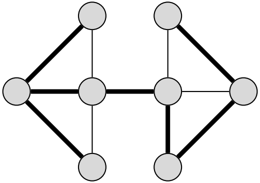




Definition: Spanning Tree

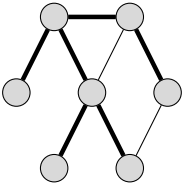
A subgraph S of G is called a *Spanning Tree*, if S is a tree and it contains every vertex of G.

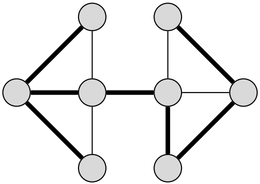




Definition: Spanning Tree

A subgraph S of G is called a *Spanning Tree*, if S is a tree and it contains every vertex of G.





So far, we have seen

- DFS/BFS Spanning Tree
- Shortest Path Spanning Tree

Lights Out

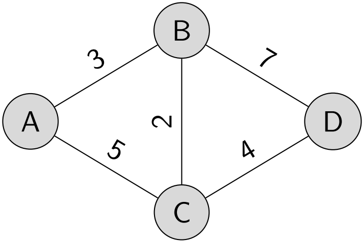
The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

Trees

Problem: Lights Out

Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

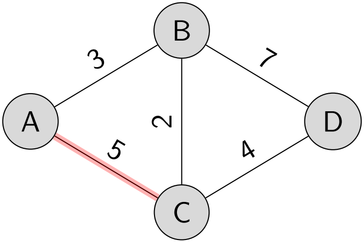


Trees

Problem: Lights Out

Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

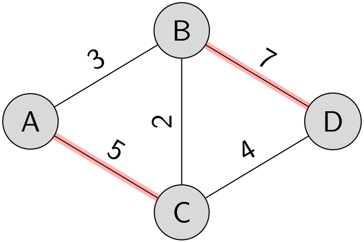


Trees

Problem: Lights Out

Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?



Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

Insight

Trees

Problem: Lights Out

Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

Insight

- We are given an undirected, weighted graph G

max planck institut informatik

Trees

Problem: Lights Out

Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

Insight

- We are given an undirected, weighted graph G
- A spanning tree of G uses the minimum number of edges to connect G

Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

Insight

- We are given an undirected, weighted graph G
- A spanning tree of G uses the minimum number of edges to connect G
- Among all spanning trees, find one of minimal total edge weight

Minimum Spanning Trees

Minimum Spanning Trees

A *Minimum Spanning Tree* (MST) is a spanning tree of a weighted graph, such that the total sum of its edge weights is minimal.

Minimum Spanning Trees

Minimum Spanning Trees

A *Minimum Spanning Tree* (MST) is a spanning tree of a weighted graph, such that the total sum of its edge weights is minimal.

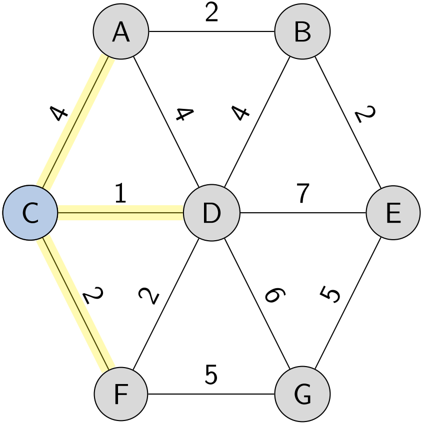
Algorithm Idea

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight
  - If the edge is a back edge, discard it
  - Otherwise, add it to the MST

Minimum Spanning Trees

Algorithm Idea

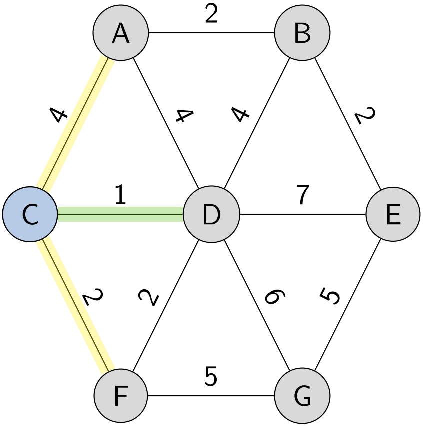
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

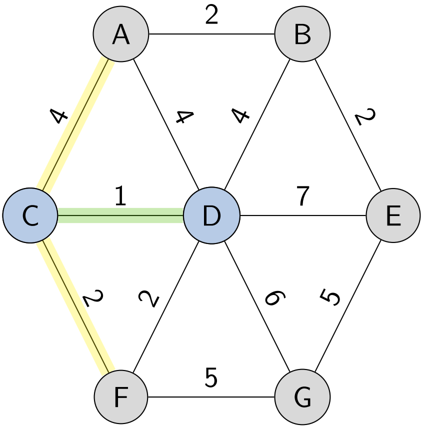
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

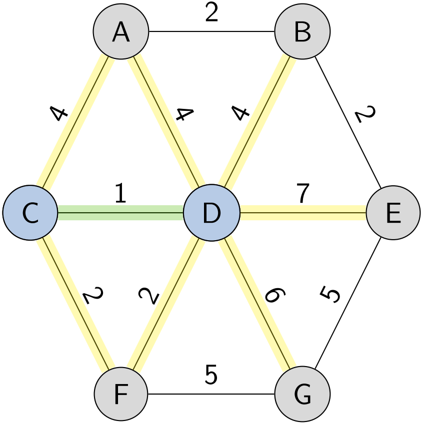
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

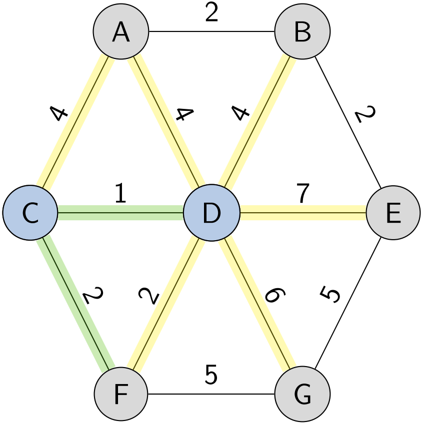
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

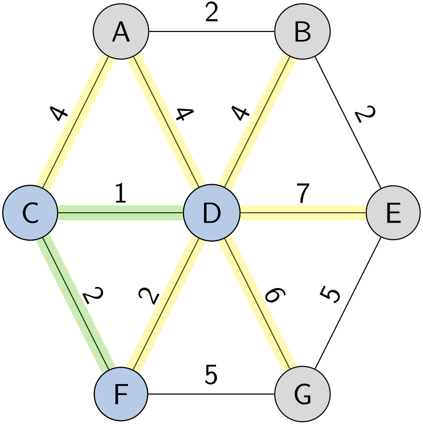
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

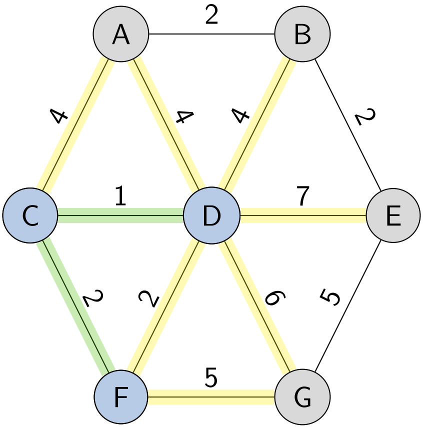
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

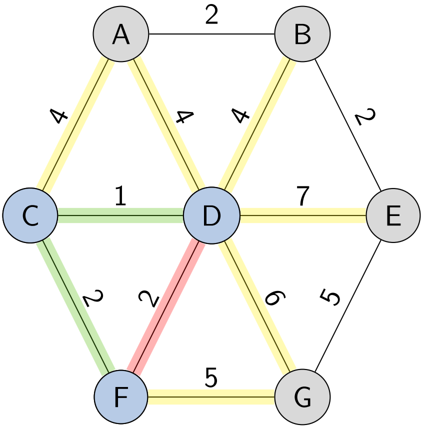
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

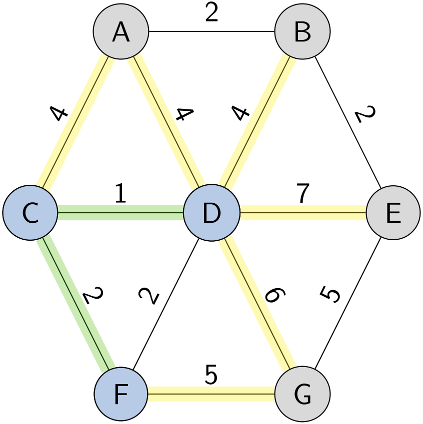
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

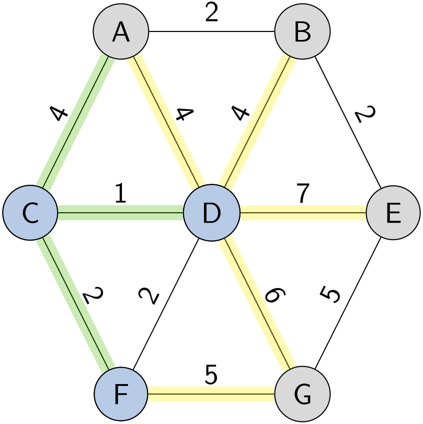
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
    - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

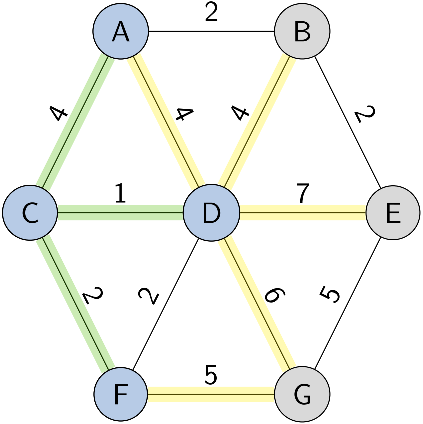
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

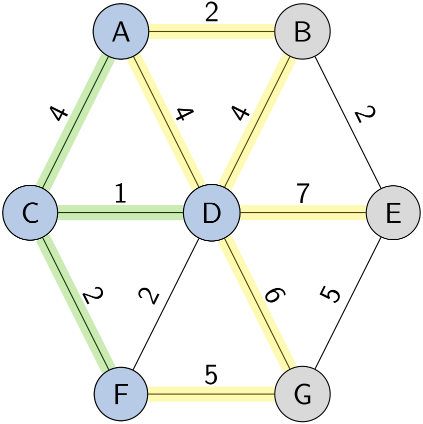
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

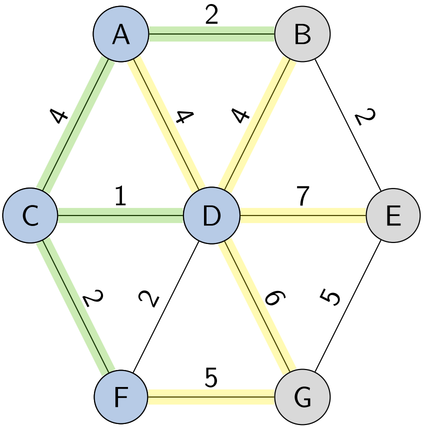
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
    - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

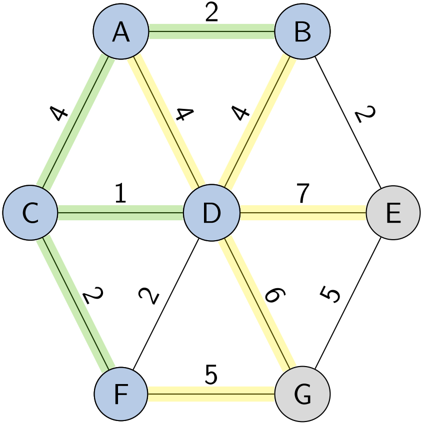
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

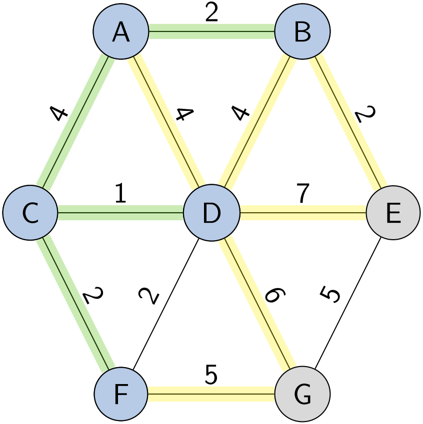
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight

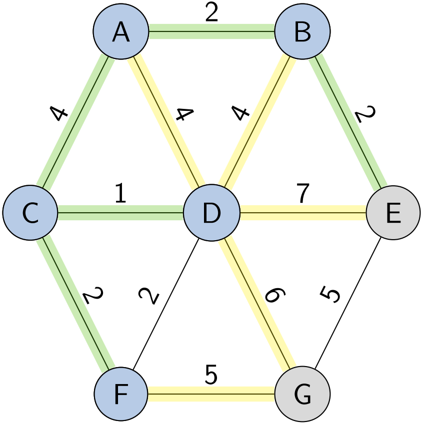


max planck institut
informatik

Minimum Spanning Trees

Algorithm Idea

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight

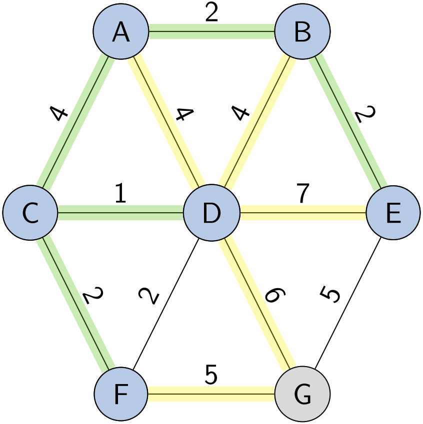


max planck institut
informatik

Minimum Spanning Trees

Algorithm Idea

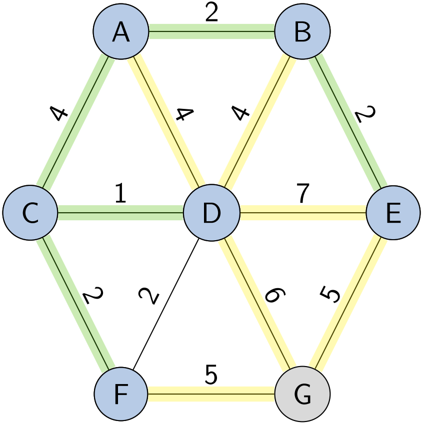
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

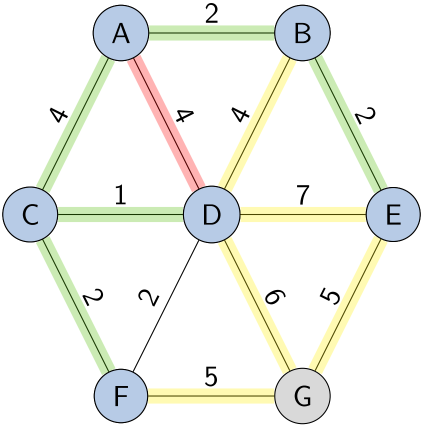
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

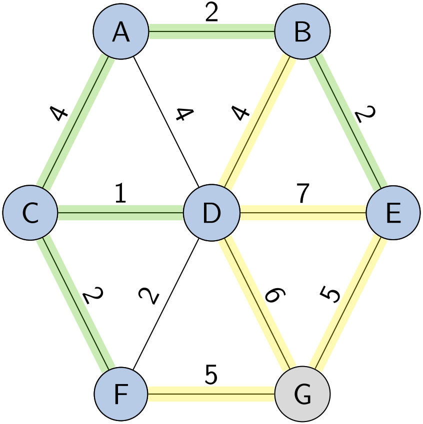
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

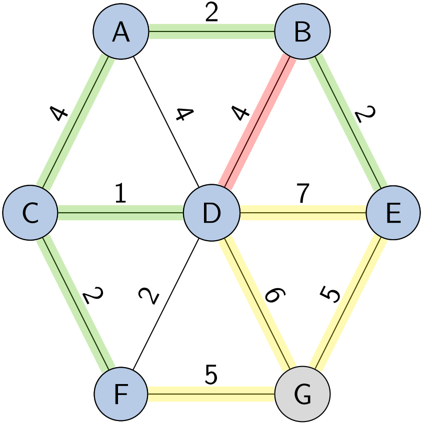
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

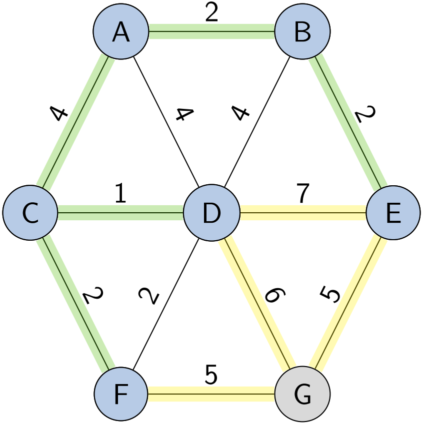
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

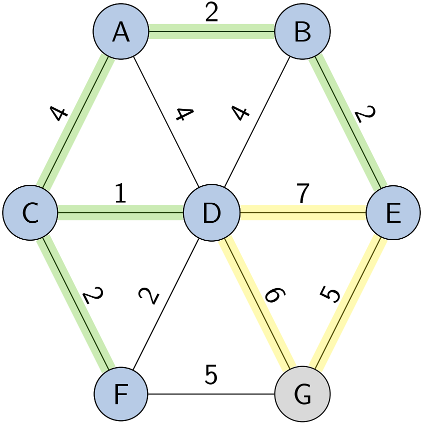
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

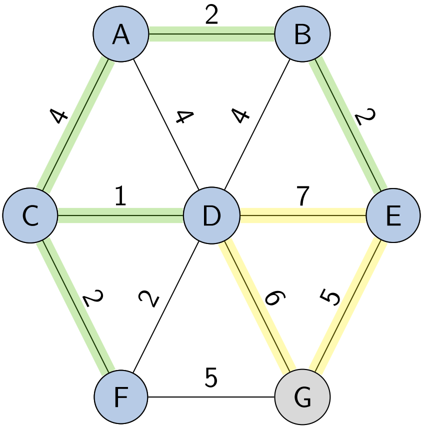
- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Algorithm Idea

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight



Minimum Spanning Trees

Prim's Algorithm

Algorithm Idea (Prim's)

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take the edge of minimum weight
  - If the edge is a back edge, discard it
  - Otherwise, add it to the MST

Minimum Spanning Trees

Prim's Algorithm

Algorithm Idea (Prim's)

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
    - Take the edge of minimum weight
    - If the edge is a back edge, discard it
    - Otherwise, add it to the MST

Sounds Familiar?

We used the same insights in Dijkstra's SSSP algorithm

Minimum Spanning Trees

Prim's Algorithm

```
vector<bool> visited(V, false);
vector<vector<pair<int,int>>> adj(V); // <to, weight>
// <weight, from, to>
priority_queue<tuple<int, int, int>,
  vector<tuple<int, int, int>>,
  greater<tuple<int,int,int>>> PQ;
void visit(int v) {
  visited[v] = true;
  for (auto p: adj[v]) {
    int u = p.first;
    int w = p.second;
    if (!visited[u]) PQ.push({w, v, u});
  }
}
visit(0); // arbitrary start vertex
while (!PQ.empty()) {
  auto front = PQ.top(); PQ.pop();
  int w, from, to;
  tie(w, from, to) = front; // unpack tuple
  if (!visited[to])
    cout << "Add " << from << "-" << to << " to MST\n";
    visit(to);
}
```

Lights Out

The town needs to save electricity and wants to turn off some street lights. However, every citizen should still be able to drive between any two intersections along an illuminated path.
Given the electricity cost per hour for each road, which roads should be left dark to save the most money?

Solution Idea

- Compute a MST of the input graph
- Sum the weights of edges not included in MST

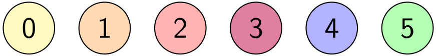
Union-Find

Union-Find Dijsoint Set (short: Union-Find) is a datastructure to manage disjoint sets. Every disjoint set is represented by one of its elements. A Union-Find supports the operations

- `unionSet(i,j)` – unify sets `i` and `j`
- `findSet(i)` – find the representant of `i`

Union-Find

Set	Representant

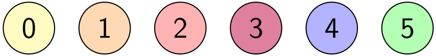


parent:

0	1	2	3	4	5

Union-Find

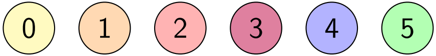
Set	Representant
{0}	0
{1}	1
{2}	2
{3}	3
{4}	4
{5}	5



	0	1	2	3	4	5
`parent:`						

Union-Find

Set	Representant
{0}	0
{1}	1
{2}	2
{3}	3
{4}	4
{5}	5



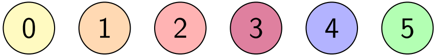
parent:

0	1	2	3	4	5
0	1	2	3	4	5

Union-Find

Set	Representant
{0}	0
{1}	1
{2}	2
{3}	3
{4}	4
{5}	5

`unionSet(1, 5)`

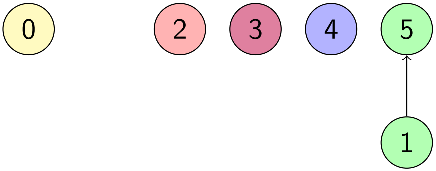


	0	1	2	3	4	5
`parent:`	0	1	2	3	4	5

Union-Find

`unionSet(1, 5)`

Set	Representant
{0}	0
{1}	1
{2}	2
{3}	3
{4}	4
{5}	5



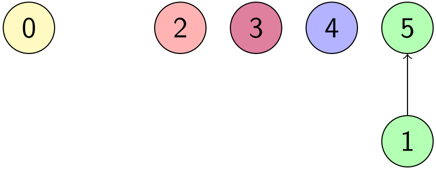
parent:

0	1	2	3	4	5
0	1	2	3	4	5

Union-Find

Set	Representant
{0}	0
{1}	1
{2}	2
{3}	3
{4}	4
{5}	5

`unionSet(1, 5)`



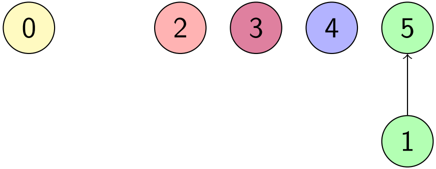
parent:

0	1	2	3	4	5
0	1	2	3	4	5

Union-Find

Set	Representant
{0}	0
{2}	2
{3}	3
{4}	4
{1, 5}	5

`unionSet(1, 5)`



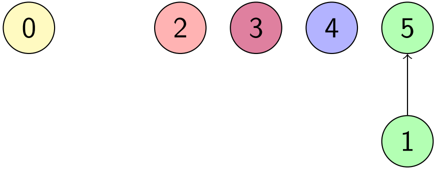
parent:

0	1	2	3	4	5
0	1	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2\}$	2
$\{3\}$	3
$\{4\}$	4
$\{1, 5\}$	5

`unionSet(1, 5)`



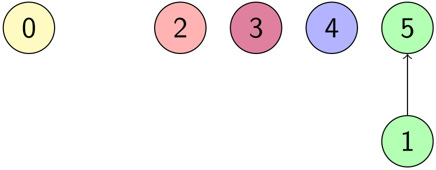
`parent:`

0	1	2	3	4	5
0	1	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2\}$	2
$\{3\}$	3
$\{4\}$	4
$\{1, 5\}$	5

`unionSet(1, 5)`



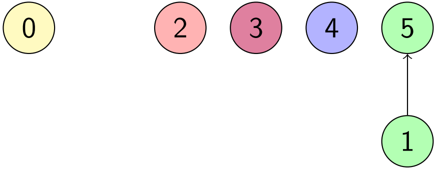
`parent:`

0	1	2	3	4	5
0	5	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2\}$	2
$\{3\}$	3
$\{4\}$	4
$\{1, 5\}$	5

`unionSet(4, 2)`



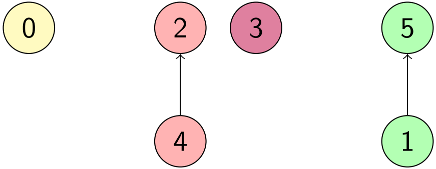
`parent:`

0	1	2	3	4	5
0	5	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2\}$	2
$\{3\}$	3
$\{4\}$	4
$\{1, 5\}$	5

`unionSet(4, 2)`

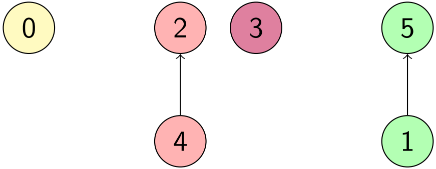


	0	1	2	3	4	5
`parent:`	0	5	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2\}$	2
$\{3\}$	3
$\{4\}$	4
$\{1, 5\}$	5

`unionSet(4, 2)`



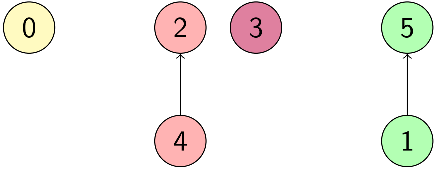
`parent:`

0	1	2	3	4	5
0	5	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2, 4\}$	2
$\{3\}$	3
$\{1, 5\}$	5

`unionSet(4, 2)`



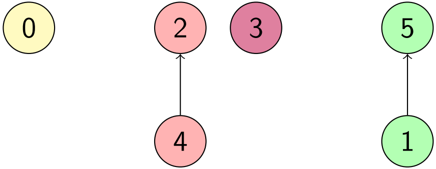
`parent:`

0	1	2	3	4	5
0	5	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2,4\}$	2
$\{3\}$	3
$\{1,5\}$	5

`unionSet(4, 2)`



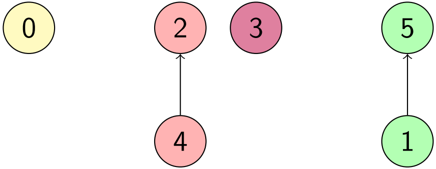
`parent:`

0	1	2	3	4	5
0	5	2	3	4	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2, 4\}$	2
$\{3\}$	3
$\{1, 5\}$	5

`unionSet(4, 2)`

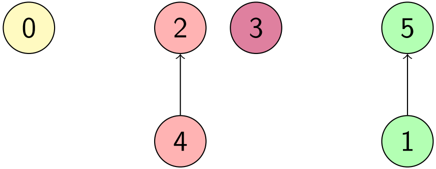


	0	1	2	3	4	5
`parent:`	0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2, 4\}$	2
$\{3\}$	3
$\{1, 5\}$	5

`findSet(4)`

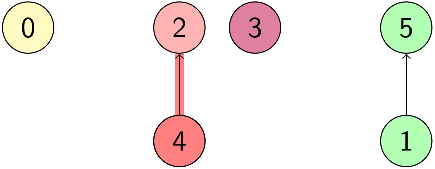


	0	1	2	3	4	5
`parent:`	0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2,4\}$	2
$\{3\}$	3
$\{1,5\}$	5

`findSet(4)`

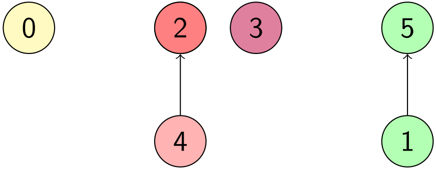


	0	1	2	3	4	5
`parent:`	0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2, 4\}$	2
$\{3\}$	3
$\{1, 5\}$	5

`findSet(4)`

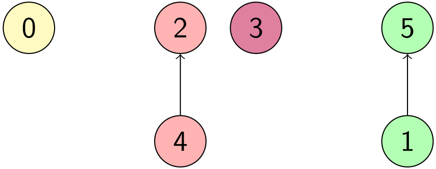


	0	1	2	3	4	5
`parent:`	0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2,4\}$	2
$\{3\}$	3
$\{1,5\}$	5

`findSet(4) = 2`



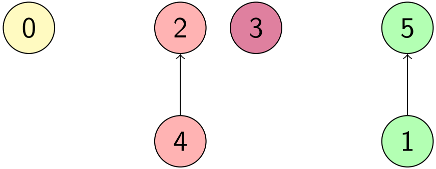
`parent:`

0	1	2	3	4	5
0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2,4\}$	2
$\{3\}$	3
$\{1,5\}$	5

`unionSet(2, 5)`



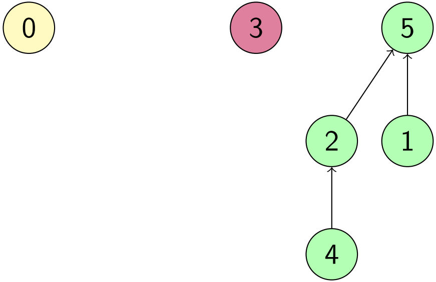
`parent:`

0	1	2	3	4	5
0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{2,4\}$	2
$\{3\}$	3
$\{1,5\}$	5

`unionSet(2, 5)`



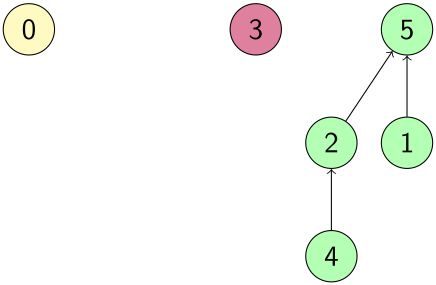
`parent:`

0	1	2	3	4	5
0	5	2	3	2	5

Union-Find

`unionSet(2, 5)`

Set	Representant
$\{0\}$	0
$\{2,4\}$	2
$\{3\}$	3
$\{1,5\}$	5

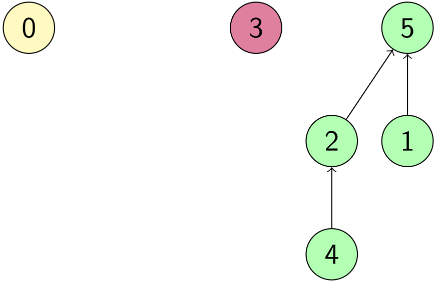


	0	1	2	3	4	5
`parent:`	0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{3\}$	3
$\{1, 2, 4, 5\}$	5

`unionSet(2, 5)`



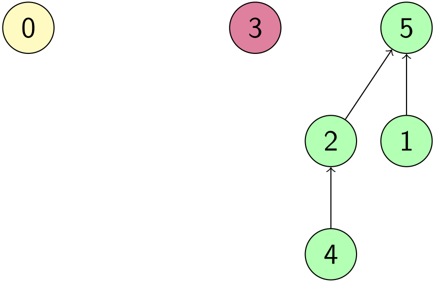
`parent:`

0	1	2	3	4	5
0	5	2	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{3\}$	3
$\{1, 2, 4, 5\}$	5

`unionSet(2, 5)`



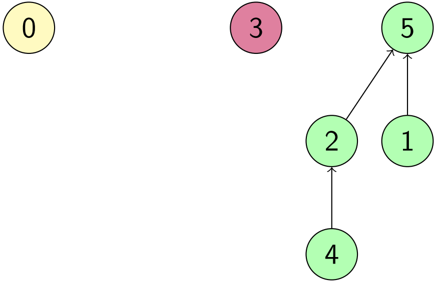
`parent:`

0	1	2	3	4	5
0	5	2	3	2	5

Union-Find

Set	Representant
{0}	0
{3}	3
{1, 2, 4, 5}	5

`unionSet(2, 5)`



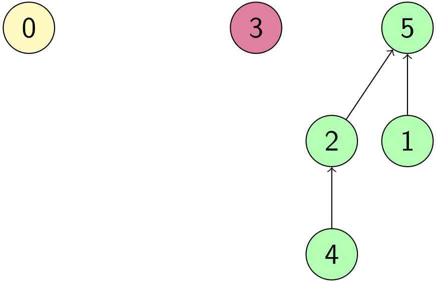
`parent:`

0	1	2	3	4	5
0	5	5	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{3\}$	3
$\{1, 2, 4, 5\}$	5

`findSet(4)`



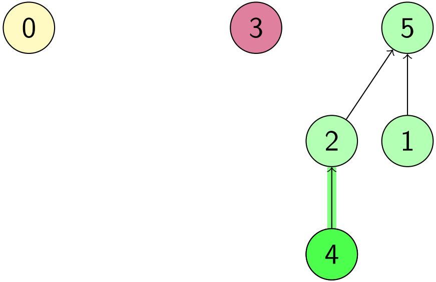
`parent:`

0	1	2	3	4	5
0	5	5	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{3\}$	3
$\{1, 2, 4, 5\}$	5

`findSet(4)`



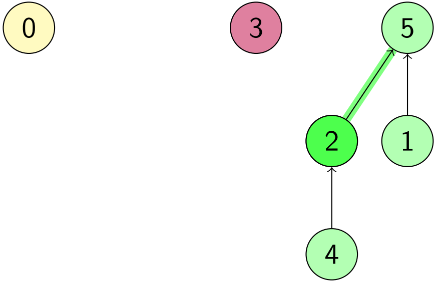
parent:

0	1	2	3	4	5
0	5	5	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{3\}$	3
$\{1, 2, 4, 5\}$	5

`findSet(4)`

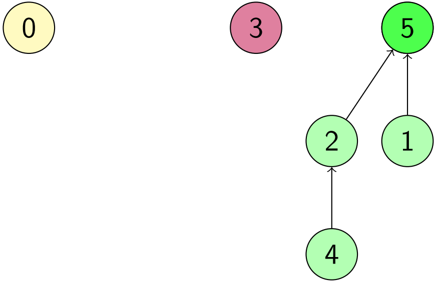


	0	1	2	3	4	5
`parent:`	0	5	5	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{3\}$	3
$\{1, 2, 4, 5\}$	5

`findSet(4)`



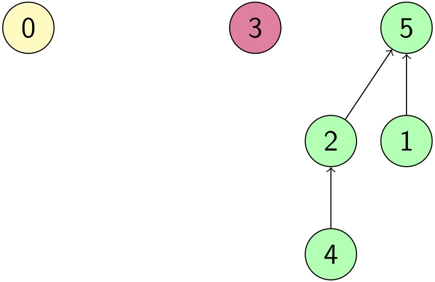
`parent:`

0	1	2	3	4	5
0	5	5	3	2	5

Union-Find

Set	Representant
$\{0\}$	0
$\{3\}$	3
$\{1, 2, 4, 5\}$	5

`findSet(4) = 5`



`parent:`

0	1	2	3	4	5
0	5	5	3	2	5

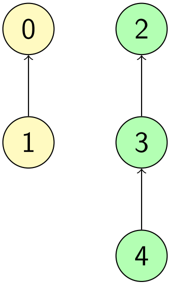
Union-Find

Example Code

```
class UnionFind {
private:
  vector<int> parent;
public:
  UnionFind(int N) {
    parent.resize(N);
    for (int i = 0; i < N; i++) parent[i] = i;
  }
  int findSet(int i) {
    if (parent[i] == i)
      return i;
    else
      return findSet(parent[i]);
  }
  bool isSameSet(int i, int j) {
    return findSet(i) == findSet(j);
  }
  void unionSet(int i, int j) {
    i = findSet(i), j = findSet(j);
    if (!isSameSet(i, j)) // or (i != j)
      parent[i] = j;
  }
};
```

Union-Find

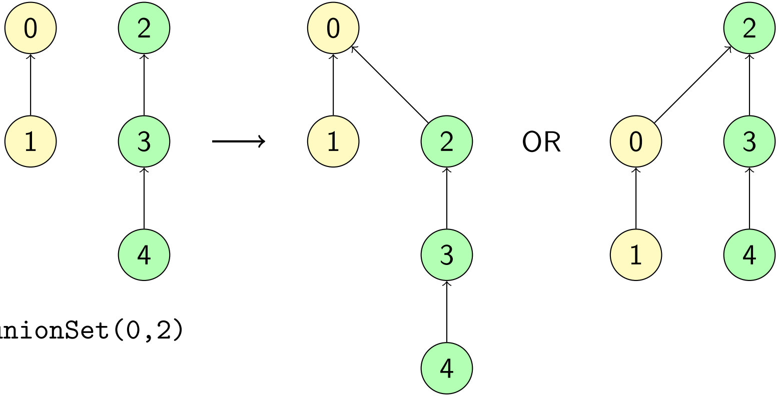
Union-by-Rank



`unionSet(0,2)`

Union-Find

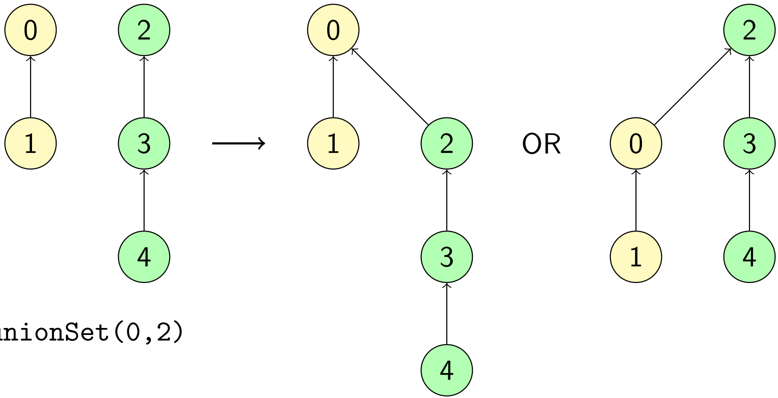
Union-by-Rank



`unionSet(0,2)`

Union-Find

Union-by-Rank

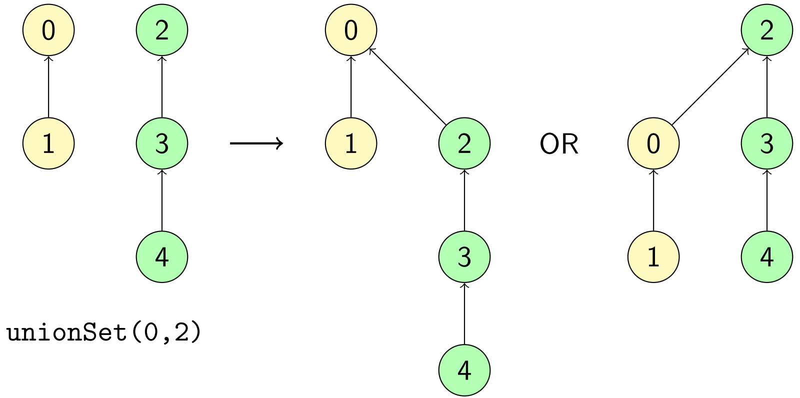


`unionSet(0,2)`

- `findSet`'s running time depends on the depth of the tree

Union-Find

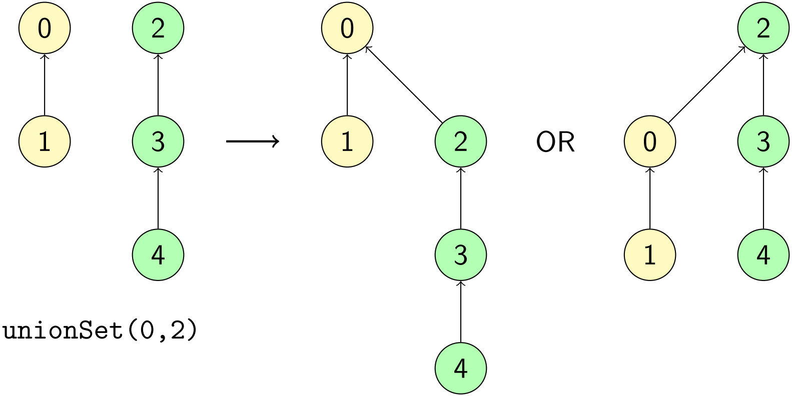
Union-by-Rank



- `findSet`'s running time depends on the depth of the tree
- Store the approximate depth of each vertex as `rank`

Union-Find

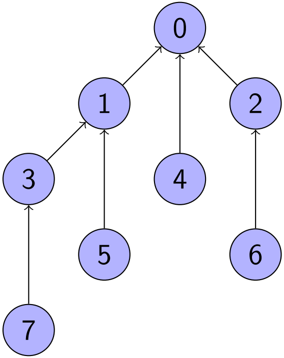
Union-by-Rank



- `findSet`'s running time depends on the depth of the tree
- Store the approximate depth of each vertex as `rank`
- Use `rank` to keep the total depth small during `unionSet`

Union-Find

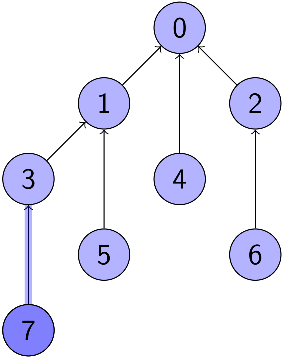
Path Compression



`findSet(7)`

Union-Find

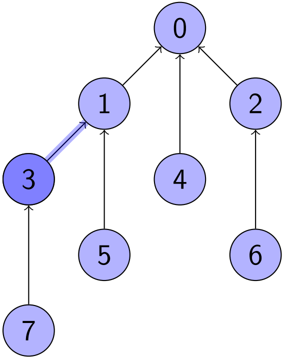
Path Compression



`findSet(7)`

Union-Find

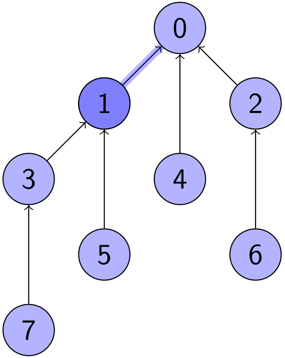
Path Compression



`findSet(7)`

Union-Find

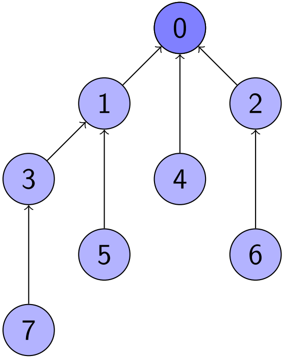
Path Compression



`findSet(7)`

Union-Find

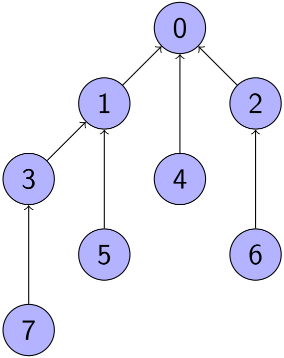
Path Compression



`findSet(7)`

Union-Find

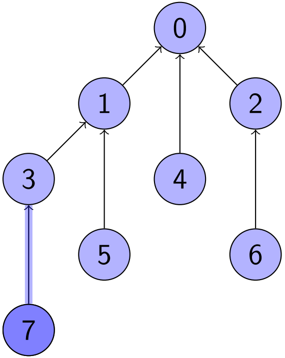
Path Compression



`findSet(7) = 0`

Union-Find

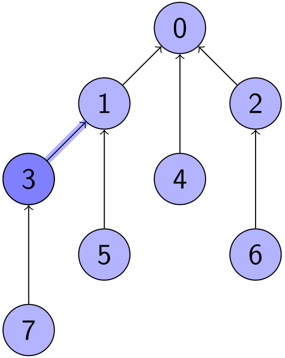
Path Compression



`findSet(7) = 0`

Union-Find

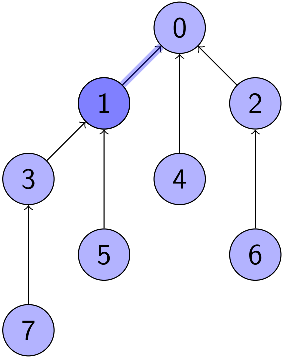
Path Compression



`findSet(7) = 0`

Union-Find

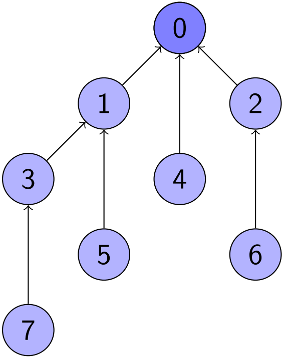
Path Compression



`findSet(7) = 0`

Union-Find

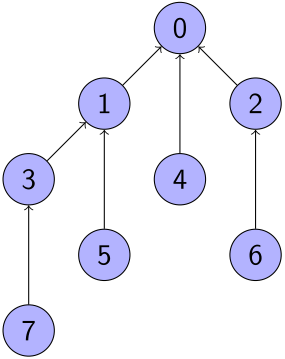
Path Compression



`findSet(7) = 0`

Union-Find

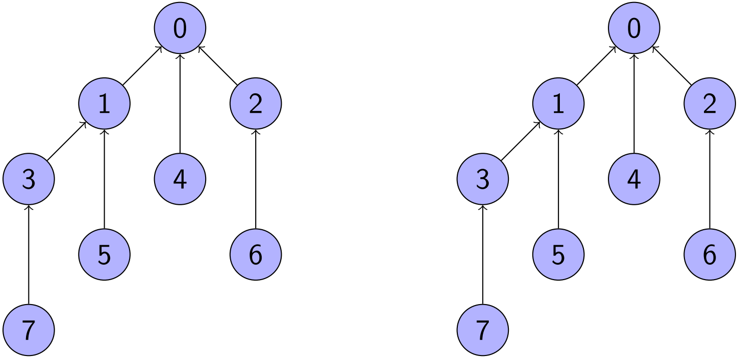
Path Compression



`findSet(7) = 0`

Union-Find

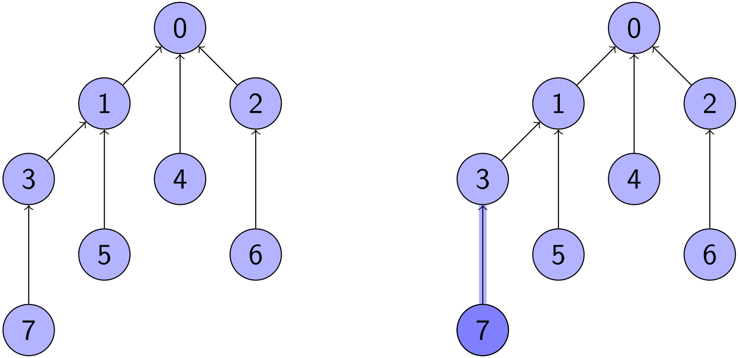
Path Compression



`findSet(7) = 0` `findSet(7)`

Union-Find

Path Compression

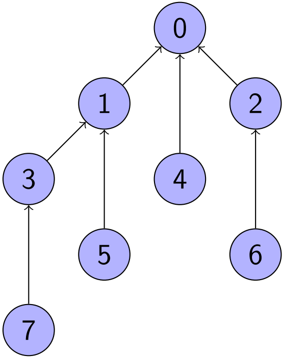


`findSet(7) = 0`

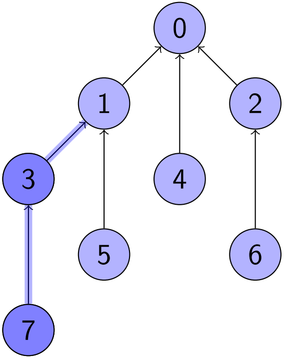
`findSet(7)`

Union-Find

Path Compression



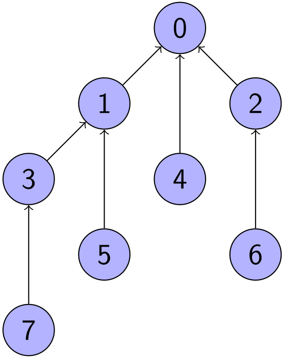
`findSet(7) = 0`



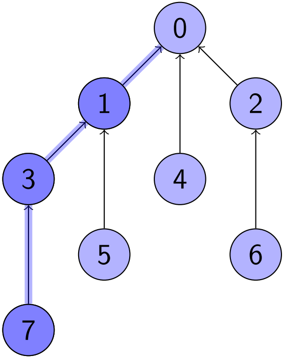
`findSet(7)`

Union-Find

Path Compression



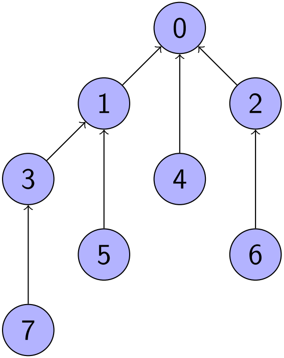
`findSet(7) = 0`



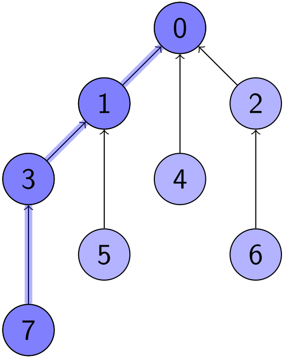
`findSet(7)`

Union-Find

Path Compression



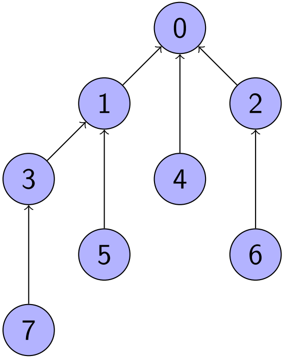
`findSet(7) = 0`



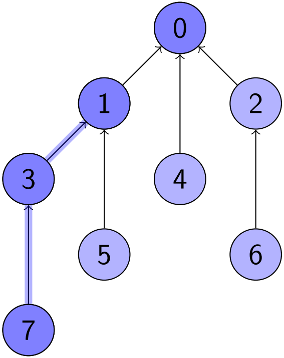
`findSet(7)`

Union-Find

Path Compression



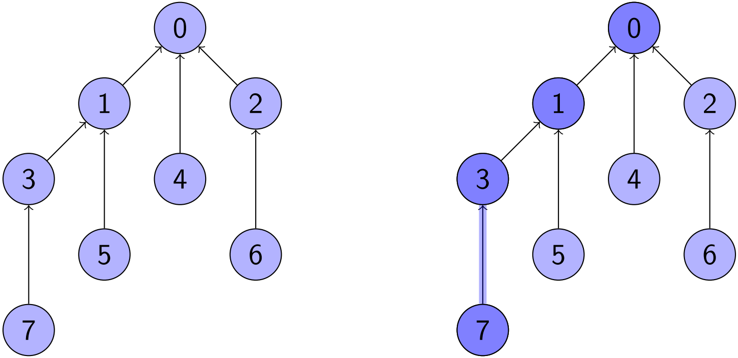
`findSet(7) = 0`



`findSet(7)`

Union-Find

Path Compression

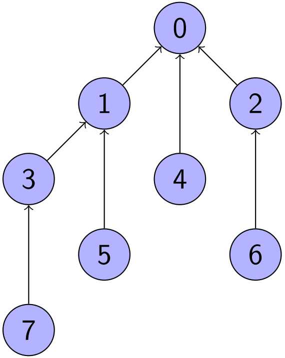


findSet(7) = 0

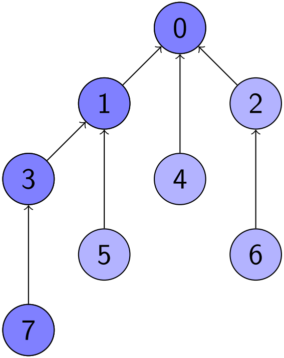
findSet(7)

Union-Find

Path Compression



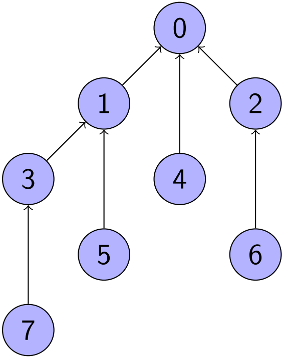
`findSet(7) = 0`



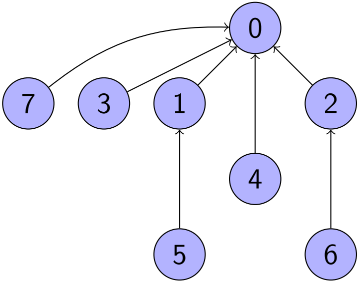
`findSet(7) = 0`

Union-Find

Path Compression



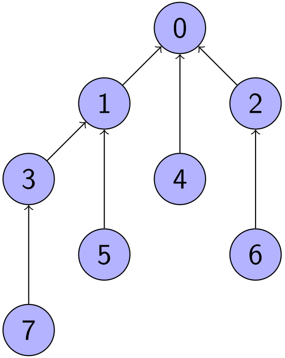
`findSet(7) = 0`



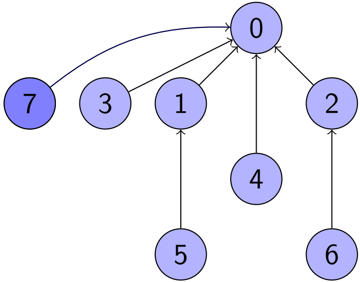
`findSet(7) = 0`

Union-Find

Path Compression



`findSet(7) = 0`

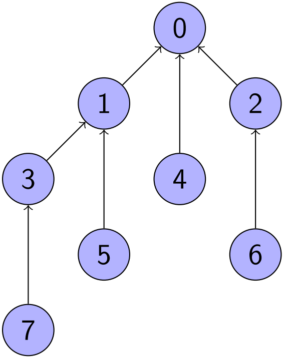


`findSet(7) = 0`

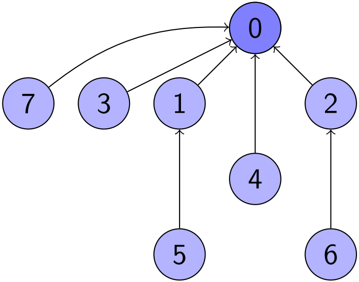
max planck institut
informatik

Union-Find

Path Compression



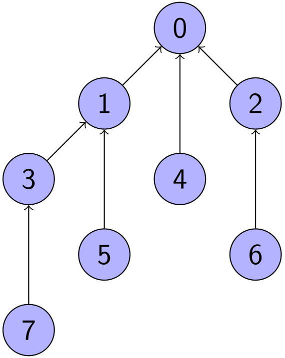
`findSet(7) = 0`



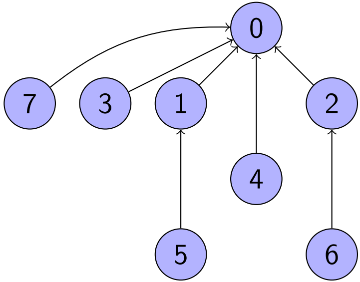
`findSet(7) = 0`

Union-Find

Path Compression



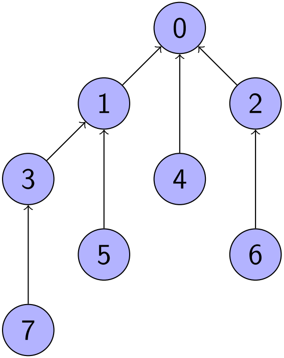
`findSet(7) = 0`



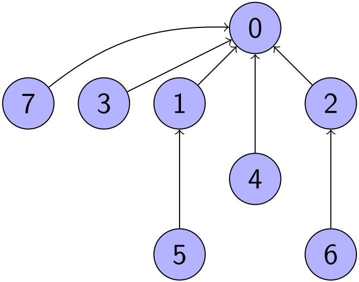
`findSet(7) = 0`

Union-Find

Path Compression



`findSet(7) = 0`

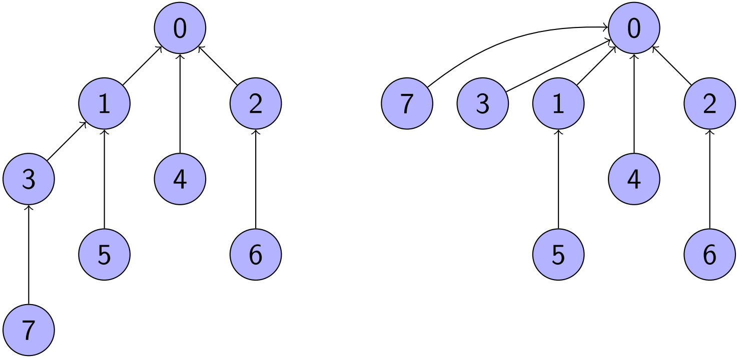


`findSet(7) = 0`

- The first `findSet` query updates all parents on the path to the root

Union-Find

Path Compression



`findSet(7) = 0` `findSet(7) = 0`

- The first `findSet` query updates all parents on the path to the root
- Subsequent `findSet` queries benefit from shallower tree

Union-Find

Improved Example Code

```
class UnionFind {
private:
    vector<int> parent, rank; // add rank vector
public:
  UnionFind(int N) {
    rank.assign(N, 0); // initialize
    parent.resize(N);
    for (int i = 0; i < N; i++) parent[i] = i;
  }
  void unionSet(int i, int j) {
    i = findSet(i), j = findSet(j);
    if (!isSameSet(i, j)) { // or (i != j)
      if (rank[i] > rank[j]) { // union by rank
        parent[j] = i;
      } else {
        parent[i] = j;
        if (rank[i] == rank[j]) rank[j]++;
      }
    }
  }
```

Union-Find

Improved Example Code

```
  int findSet(int i) {
    if (parent[i] == i)
      return i;
    else      // path compression
      return parent[i] = findSet(parent[i]);
  }
  bool isSameSet(int i, int j) {
    return findSet(i) == findSet(j);
  }
};
```

Union-Find

Improved Example Code

```
    int findSet(int i) {
      if (parent[i] == i)
        return i;
      else       // path compression
        return parent[i] = findSet(parent[i]);
    }
    bool isSameSet(int i, int j) {
      return findSet(i) == findSet(j);
    }
};
```

- Initially, we had $\mathcal{O}(N)$ worst case running time per query

Union-Find

Improved Example Code

```
  int findSet(int i) {
    if (parent[i] == i)
      return i;
    else      // path compression
      return parent[i] = findSet(parent[i]);
  }
  bool isSameSet(int i, int j) {
    return findSet(i) == findSet(j);
  }
};
```

- Initially, we had $\mathcal{O}(N)$ worst case running time per query
- With Path Compression and Union by Rank, we get *almost constant** running time per query

Union-Find

Conclusion

The Union-Find data structure is a useful tool for managing (potentially) large disjoint sets. It makes testing whether two elements belong to the same set easy.

Union-Find

Conclusion

The Union-Find data structure is a useful tool for managing (potentially) large disjoint sets. It makes testing whether two elements belong to the same set easy.

In particular, Union-Find is used in

- Kruskal's Algorithm for MSTs

Union-Find

Conclusion

The Union-Find data structure is a useful tool for managing (potentially) large disjoint sets. It makes testing whether two elements belong to the same set easy.

In particular, Union-Find is used in

- Kruskal's Algorithm for MSTs
- Tarjan's Algorithm for LCA

Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Prim's)

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take an edge of minimum weight
  - If the edge is a back edge, discard it
  - Otherwise, add it to the MST

Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Prim's)

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take an edge of minimum weight
  - If the edge is a back edge, discard it
  - Otherwise, add it to the MST

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost
  - If adding the edge forms a cycle, discard it
  - Otherwise, add it to the MST

Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Prim's)

- Start exploring at any vertex
- Maintain a sorted list of explored edges, the frontier
  - Take an edge of minimum weight
  - If the edge is a back edge, discard it
  - Otherwise, add it to the MST

Algorithm Idea (Kruskal's)

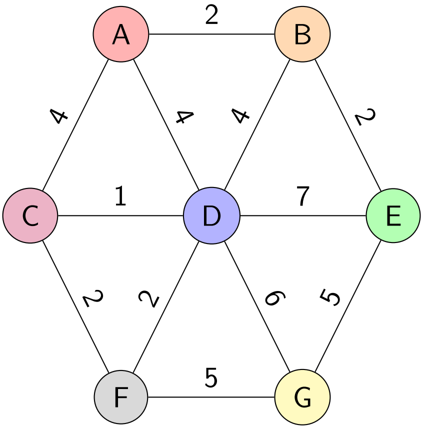
- Sort the edges by weight
- Greedily take an edge of minimum cost
  - If adding the edge forms a cycle, discard it
    - Union-Find
  - Otherwise, add it to the MST

Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

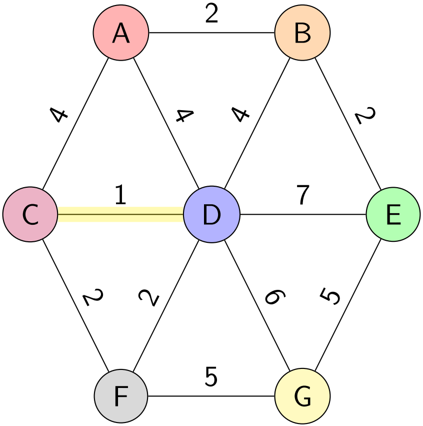


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

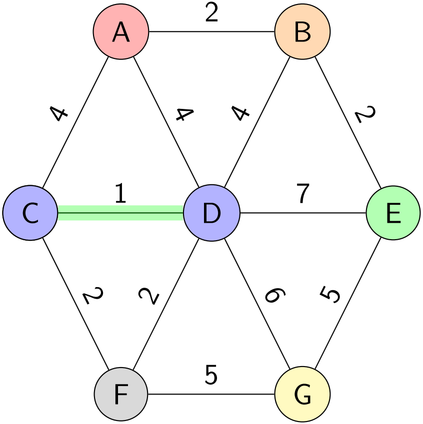


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

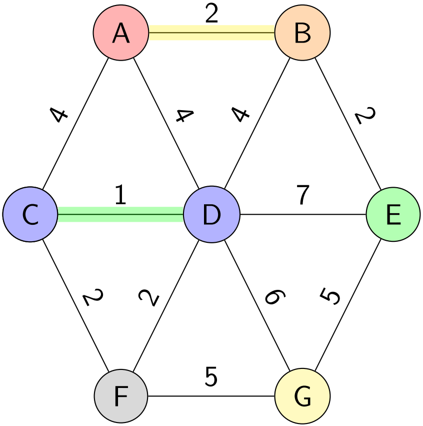


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

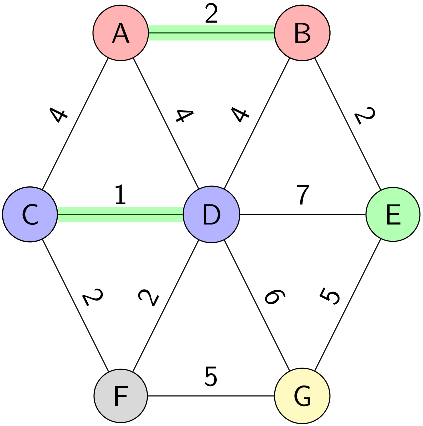


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

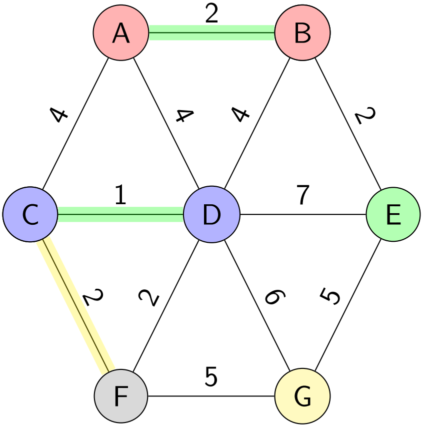


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

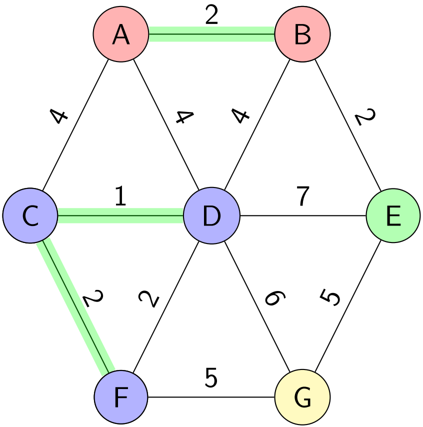


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

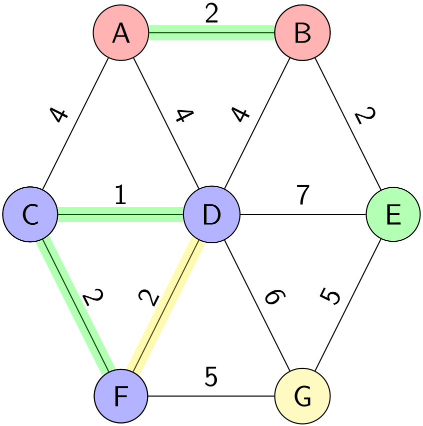


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

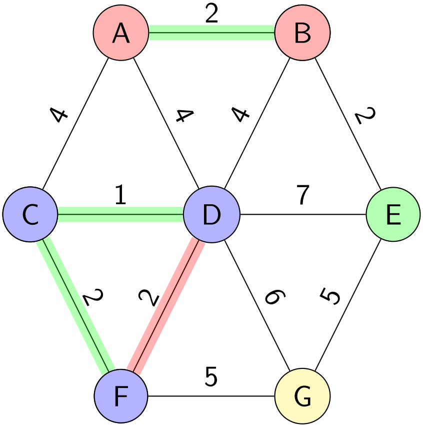


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

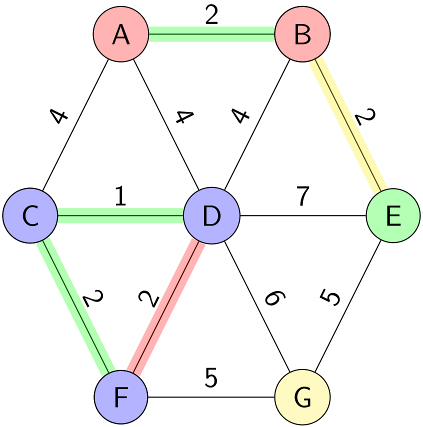


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

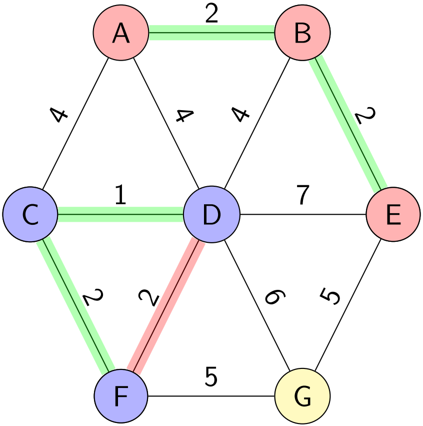


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

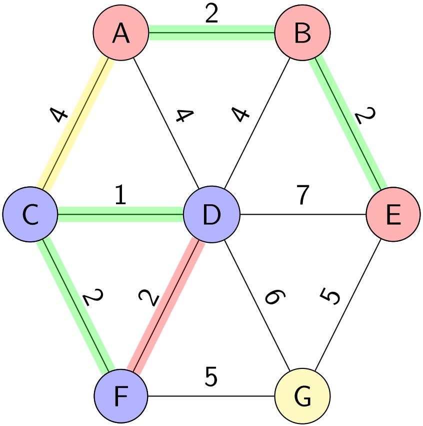


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

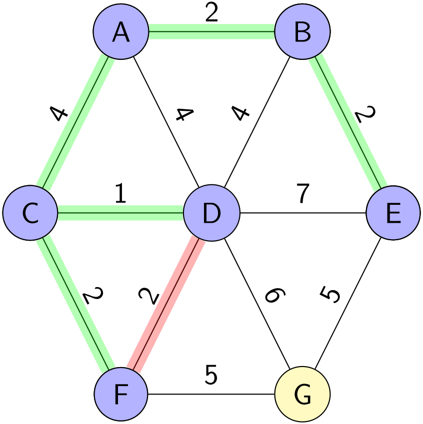


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

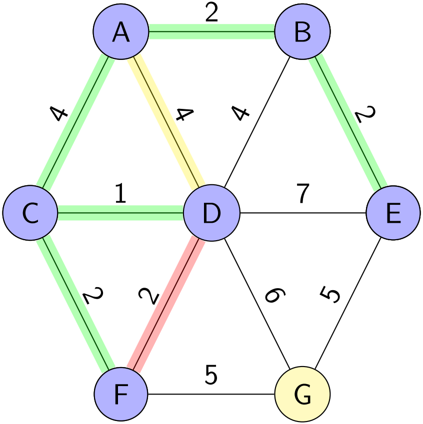


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

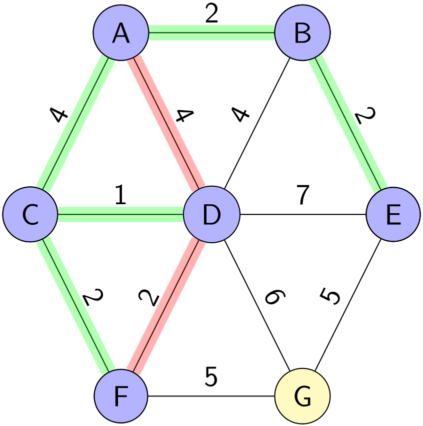


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

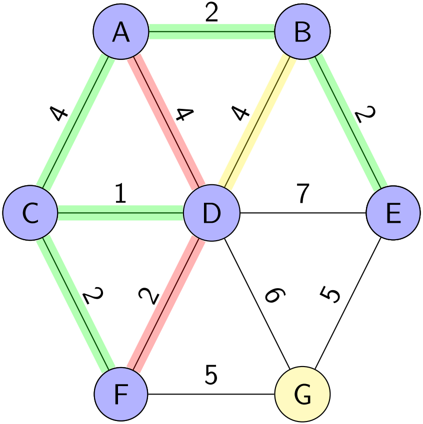


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

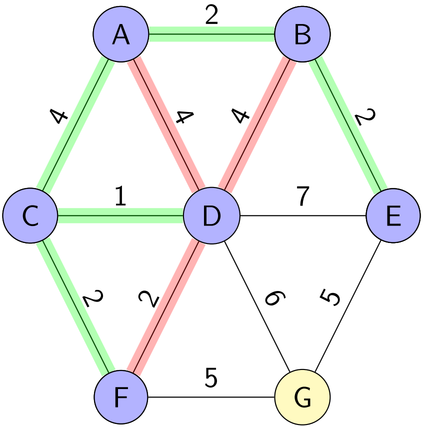
- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles



Minimum Spanning Trees

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

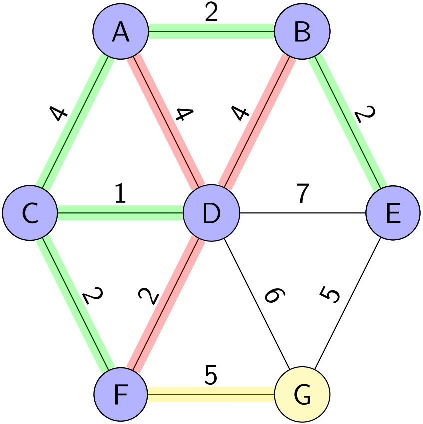


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

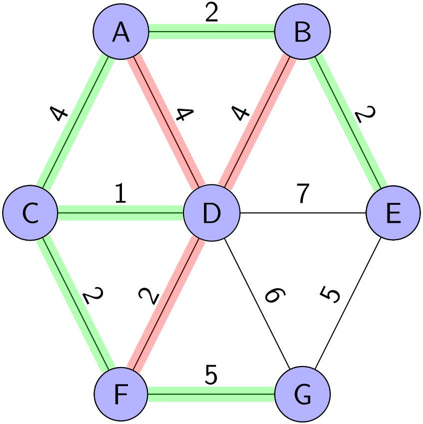


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles

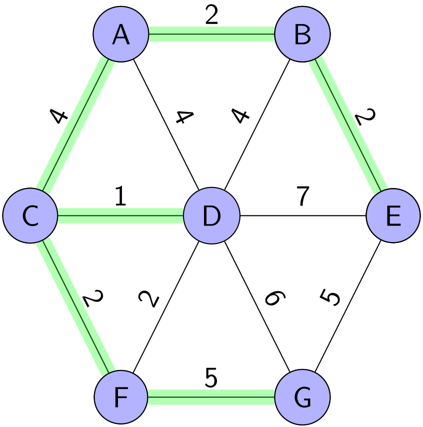


Minimum Spanning Trees

Kruskal's Algorithm

Algorithm Idea (Kruskal's)

- Sort the edges by weight
- Greedily take an edge of minimum cost, avoid cycles



Minimum Spanning Trees

Example Code: Kruskal's Algorithm

```
auto UF = UnionFind(V);

// <weight, u, v>
vector<tuple<int, int, int>> edgeList;
// sort increasing by weight
sort(edgeList.begin(), edgeList.end());

for (auto edge: edgeList) {
  int w, u, v;
  tie(w, u, v) = edge; // unpack tuple
  if (!UF.isSameSet(u, v)) {
    cout << "Add " << u << "-" << v << " to MST" << endl;
    UF.unionSet(u, v);
  }
}
```

Minimum Spanning Tree

Overview

Prim's Algorithm

- Uses PriorityQueue
- Greedily picks edges from the frontier
- Avoids cycles using `visited` flags
- Runs in $\mathcal{O}(E \log E)$

Kruskal's Algorithm

- Requires Union-Find
  - But results in shorter code
- Greedily picks edges from sorted list
- Avoids cycles using Union-Find
- Runs in $\mathcal{O}(E \log E)$

Minimum Spanning Tree

Overview

Prim's Algorithm

- Uses PriorityQueue
- Greedily picks edges from the frontier
- Avoids cycles using `visited` flags
- Runs in $\mathcal{O}(E \log E)$

Kruskal's Algorithm

- Requires Union-Find
  - But results in shorter code
- Greedily picks edges from sorted list
- Avoids cycles using Union-Find
- Runs in $\mathcal{O}(E \log E)$

- Both algorithms can early exit once (V-1) edges have been added to the MST

Union-Find

Conclusion

The Union-Find data structure is a useful tool for managing (potentially) large disjoint sets. It makes testing whether two elements belong to the same set easy.

In particular, Union-Find is used in

- Kruskal's Algorithm for MSTs
- Tarjan's Algorithm for LCA

Union-Find

Conclusion

The Union-Find data structure is a useful tool for managing (potentially) large disjoint sets. It makes testing whether two elements belong to the same set easy.

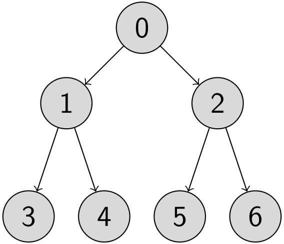
In particular, Union-Find is used in

- Kruskal's Algorithm for MSTs
- Tarjan's Algorithm for LCA

Lowest Common Ancestor

Definition: LCA

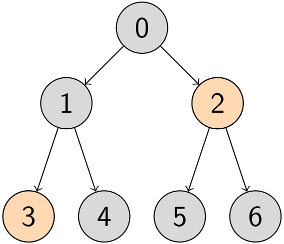
Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.



Lowest Common Ancestor

Definition: LCA

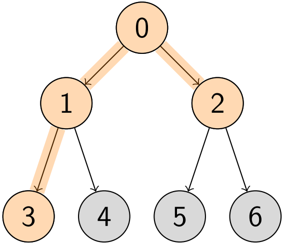
Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.



Lowest Common Ancestor

Definition: LCA

Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.

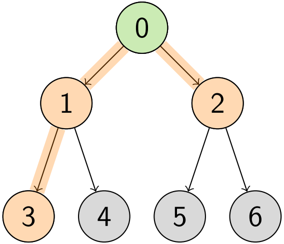


max planck institut informatik

Lowest Common Ancestor

Definition: LCA

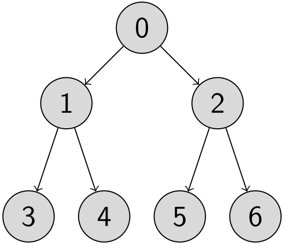
Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.



Lowest Common Ancestor

Definition: LCA

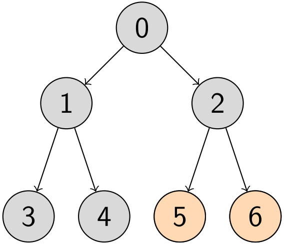
Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.



Lowest Common Ancestor

Definition: LCA

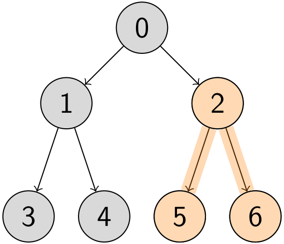
Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.



Lowest Common Ancestor

Definition: LCA

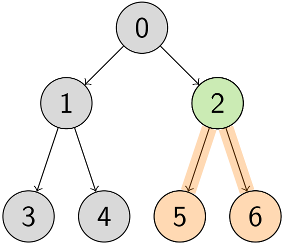
Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.



Lowest Common Ancestor

Definition: LCA

Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.



Lowest Common Ancestor

Definition: LCA

Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.

Algorithm Idea (Naive)

In order to compute LCA(u, v), we can

Lowest Common Ancestor

Definition: LCA

Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.

Algorithm Idea (Naive)

In order to compute LCA(u, v), we can

- Record the path from u and v to the root

Lowest Common Ancestor

Definition: LCA

Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.

Algorithm Idea (Naive)

In order to compute LCA(u, v), we can

- Record the path from u and v to the root
- Compare the suffixes of both paths

Lowest Common Ancestor

Definition: LCA

Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.

Algorithm Idea (Naive)

In order to compute $\text{LCA}(u, v)$, we can

- Record the path from u and v to the root
- Compare the suffixes of both paths

Paths $\langle 5, 2, 0 \rangle$ and $\langle 6, 2, 0 \rangle$ share the suffix $\langle 2, 0 \rangle$.
Thus, $\text{LCA}(5, 6) = 2$.

Lowest Common Ancestor

Definition: LCA

Given a rooted, directed tree and two vertices u and v. A vertex q is the *Lowest Common Ancestor* (LCA) of u and v if and only if q is the lowest (deepest, furthest from the root) vertex such that both u and v are contained in the subtree starting at q.

Algorithm Idea (Naive)

In order to compute LCA(u, v), we can

- Record the path from u and v to the root
- Compare the suffixes of both paths

Runs in $\mathcal{O}(depth) \subseteq \mathcal{O}(V)$

Lowest Common Ancestor

Problem: Multiple LCAs

Given a rooted, directed tree and n pairs of vertices u_i and v_i.
For each i, output $\mathrm{LCA}(u_i, v_i)$.

Lowest Common Ancestor

Problem: Multiple LCAs

Given a rooted, directed tree and n pairs of vertices u_i and v_i.
For each i, output $\mathrm{LCA}(u_i, v_i)$.

Idea

Run the previous algorithm n times.

- Runs in $\mathcal{O}(n \cdot depth) \subseteq \mathcal{O}(n \cdot V)$
- Can we do better?

Lowest Common Ancestor

Problem: Multiple LCAs

Given a rooted, directed tree and n pairs of vertices u_i and v_i.
For each i, output $\text{LCA}(u_i, v_i)$.

Idea

Run the previous algorithm n times.

- Runs in $\mathcal{O}(n \cdot depth) \subseteq \mathcal{O}(n \cdot V)$
- Can we do better?

Better Idea

Take advantage of the fact that the tree never changes.

Lowest Common Ancestor

Problem: Multiple LCAs

Given a rooted, directed tree and n pairs of vertices u_i and v_i.
For each i, output $\mathrm{LCA}(u_i, v_i)$.

Idea

Run the previous algorithm n times.

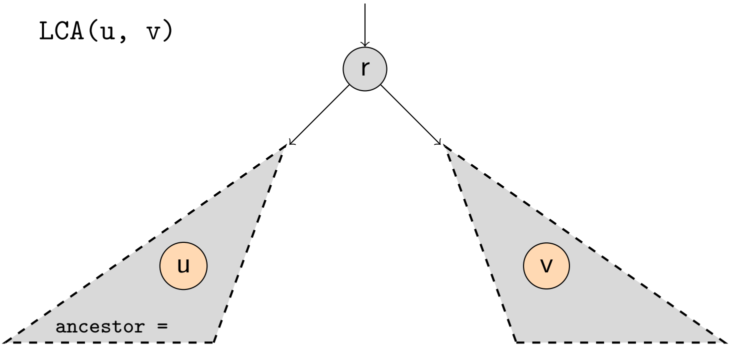
- Runs in $\mathcal{O}(n \cdot depth) \subseteq \mathcal{O}(n \cdot V)$
- Can we do better?

Better Idea

Take advantage of the fact that the tree never changes.

- We *can* do better

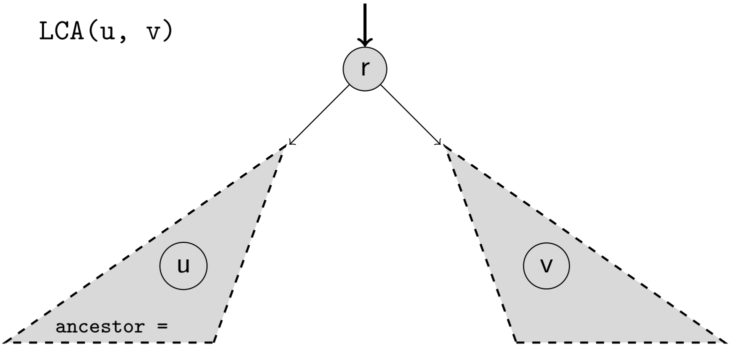
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

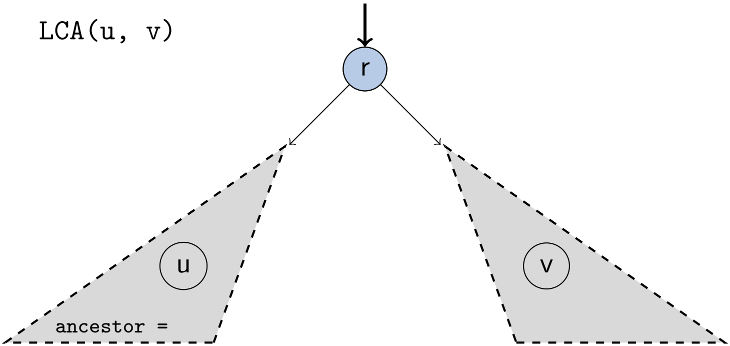
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

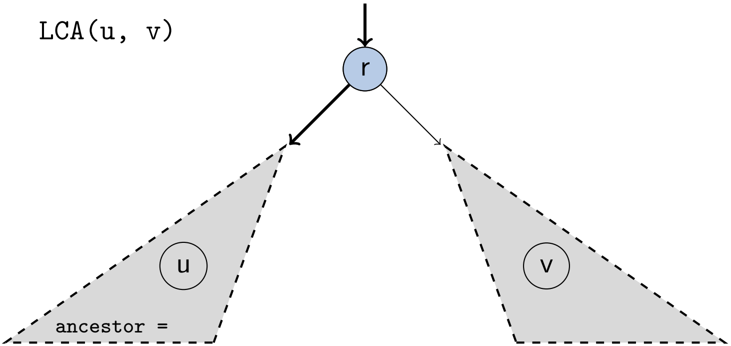
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

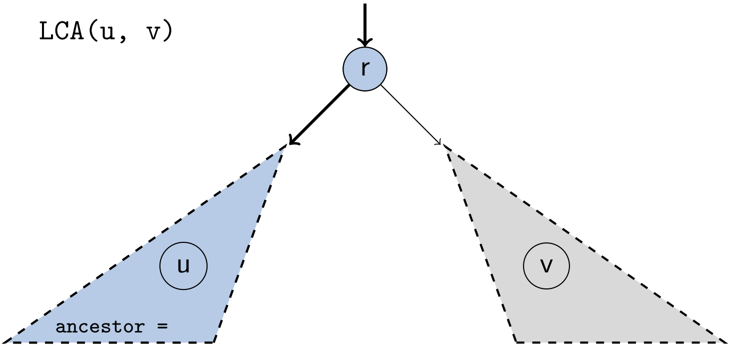
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

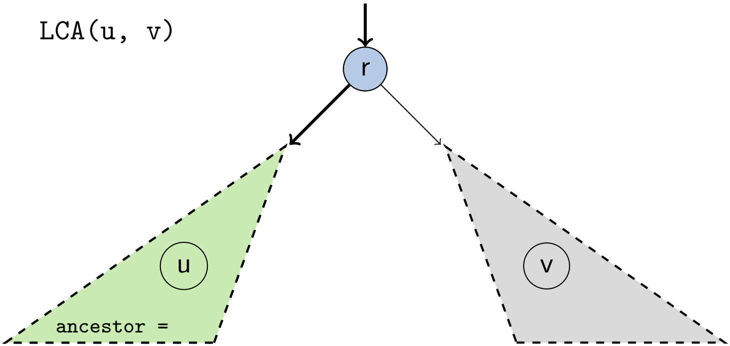
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

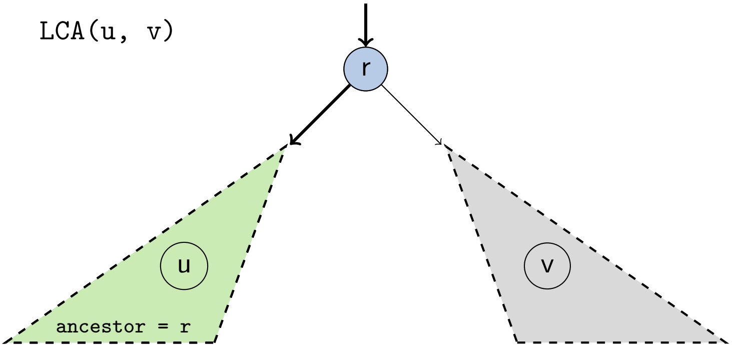
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

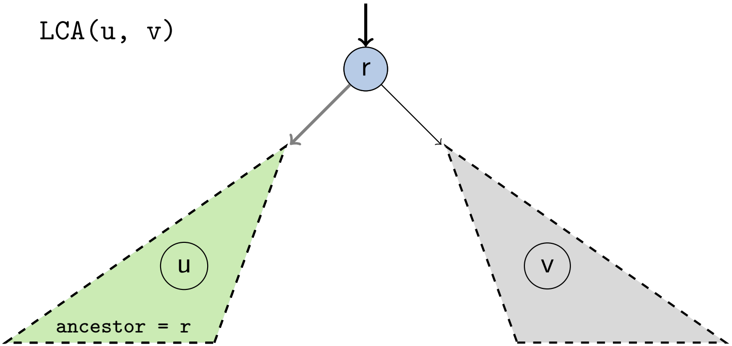
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

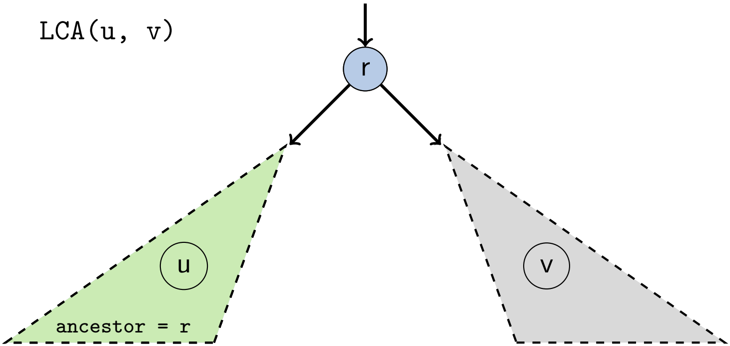
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

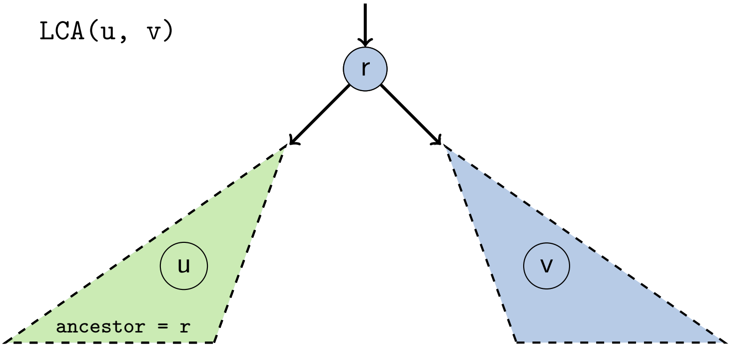
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

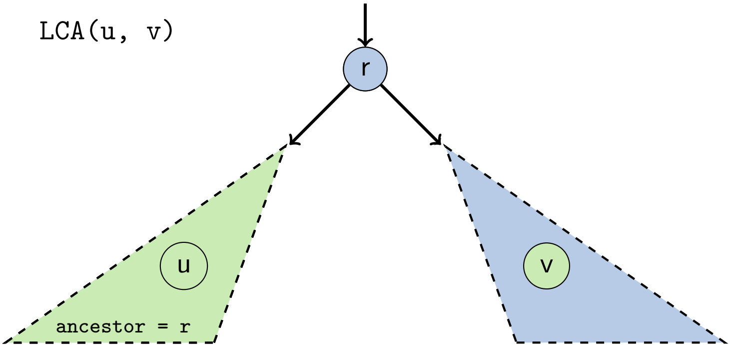
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

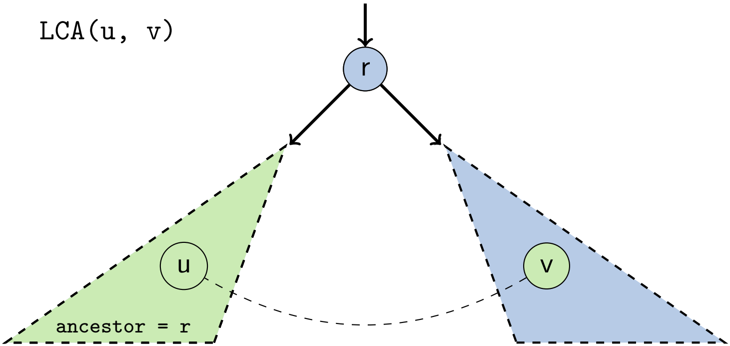
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

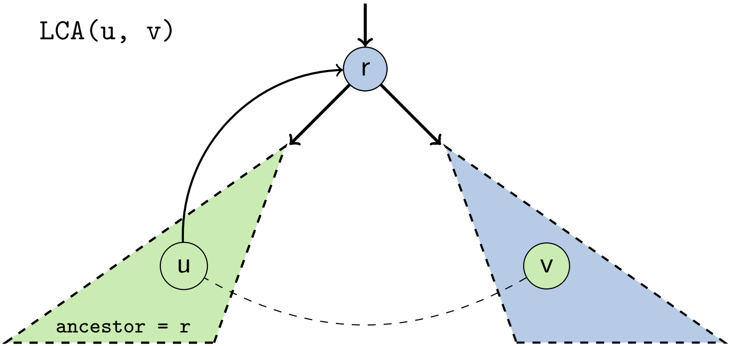
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

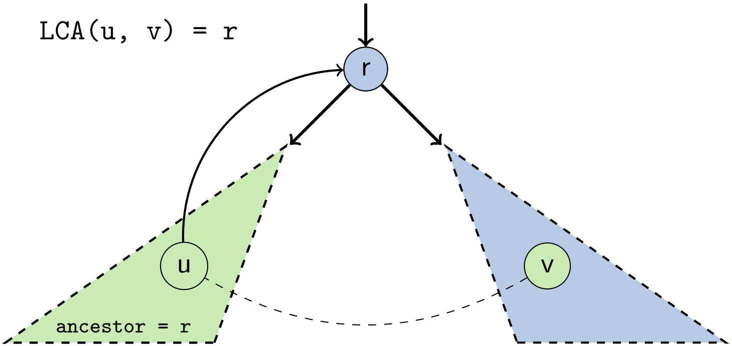
Lowest Common Ancestor



Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

Lowest Common Ancestor



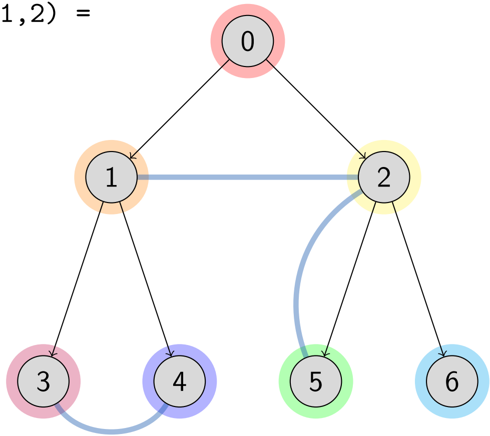
Insights: Multiple LCAs

- Traverse the Tree via DFS
- After exploring v, if u was finished before, `ancestor[u]` is the LCA
  - The v-subtree is still being explored, thus r is ancestor of v
  - Make sure that once u is finished, r becomes its ancestor

Lowest Common Ancestor

LCA Example

```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



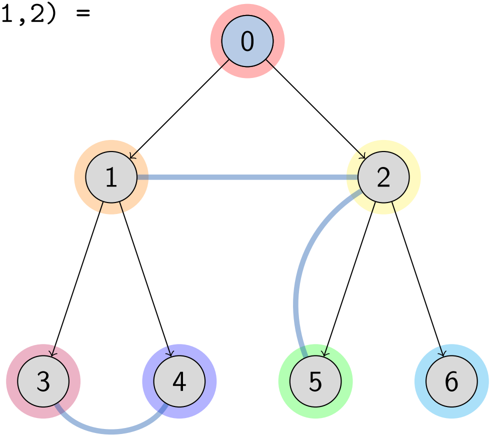
Lowest Common Ancestor

LCA Example

`LCA(3,4) =`

`LCA(2,5) =`

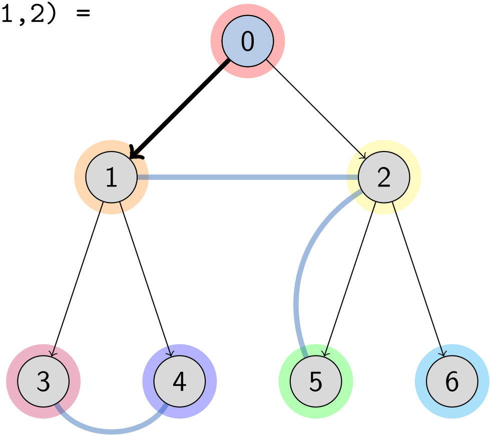
`LCA(1,2) =`



Lowest Common Ancestor

LCA Example

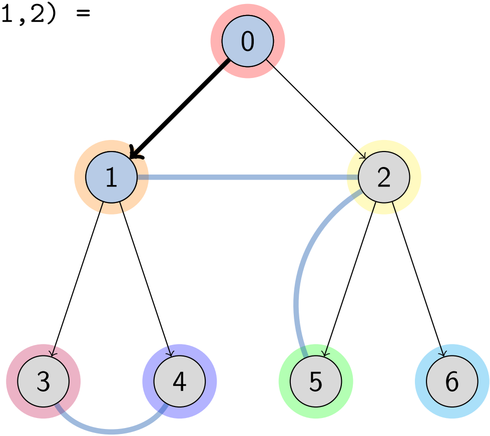
```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

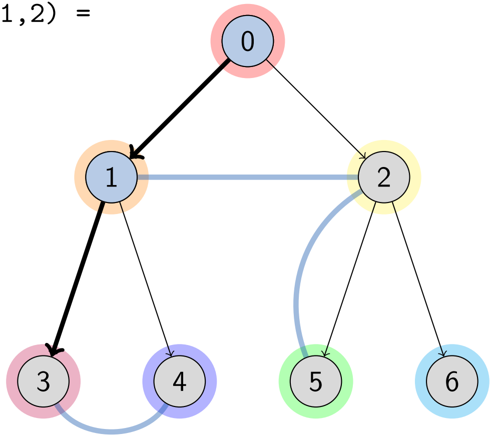
```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

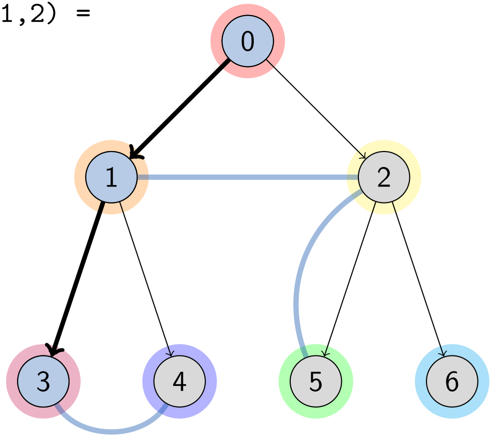
```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



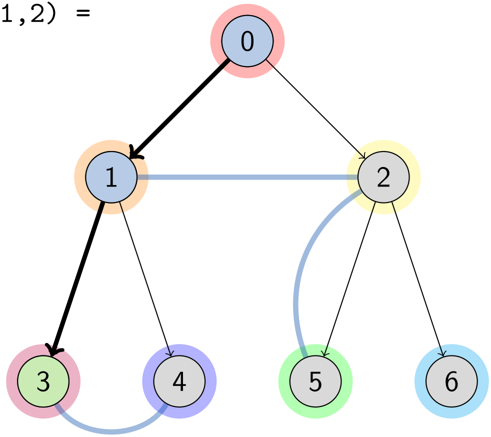
Lowest Common Ancestor

LCA Example

`LCA(3,4) =`

`LCA(2,5) =`

`LCA(1,2) =`



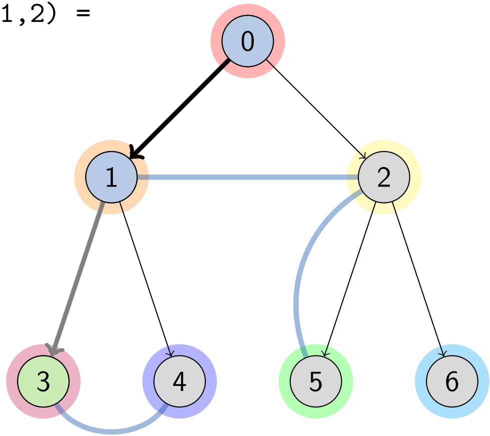
Lowest Common Ancestor

LCA Example

`LCA(3,4) =`

`LCA(2,5) =`

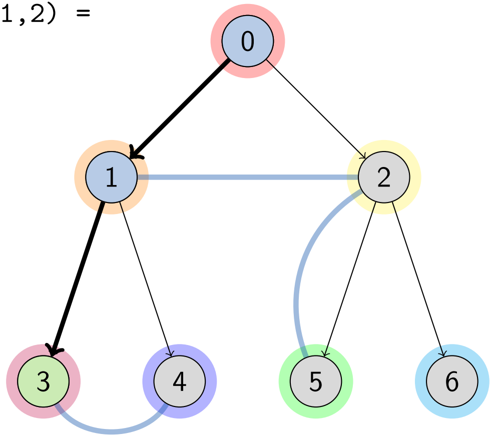
`LCA(1,2) =`



Lowest Common Ancestor

LCA Example

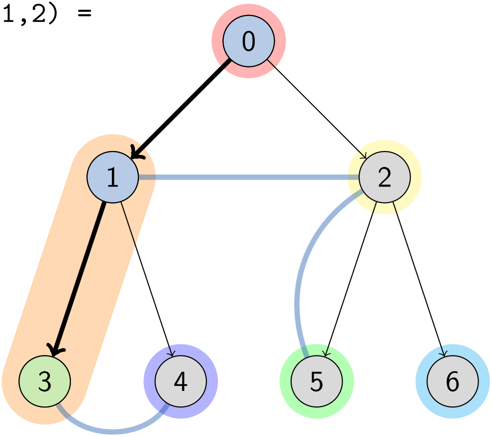
```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

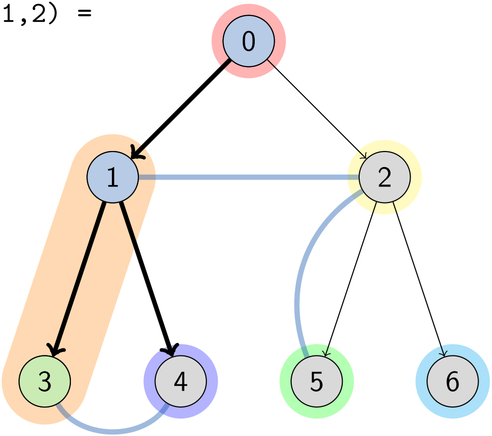
```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

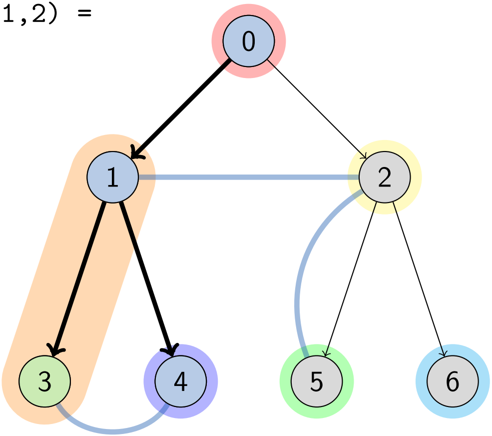
```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



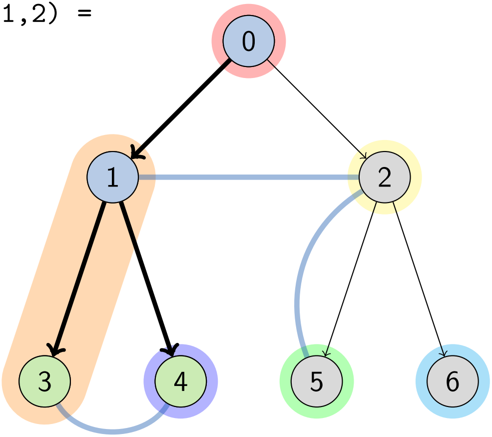
Lowest Common Ancestor

LCA Example

`LCA(3,4) =`

`LCA(2,5) =`

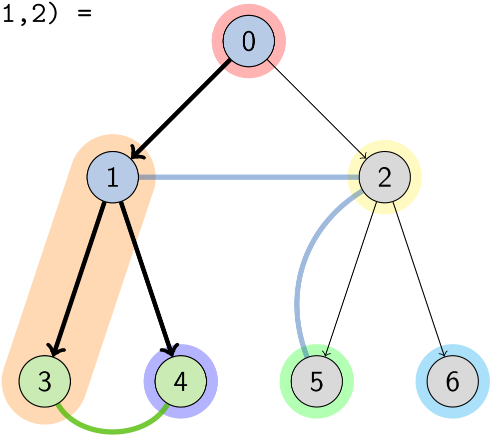
`LCA(1,2) =`



Lowest Common Ancestor

LCA Example

```
LCA(3,4) =
LCA(2,5) =
LCA(1,2) =
```



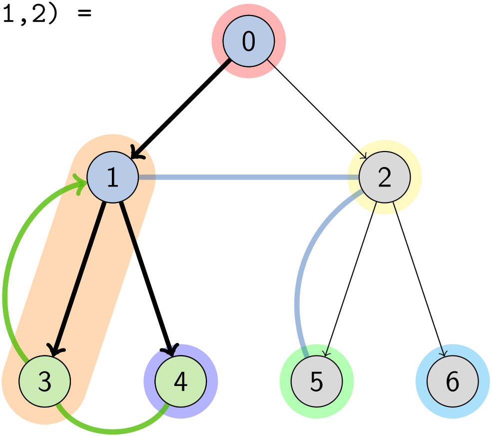
Lowest Common Ancestor

LCA Example

`LCA(3,4) =`

`LCA(2,5) =`

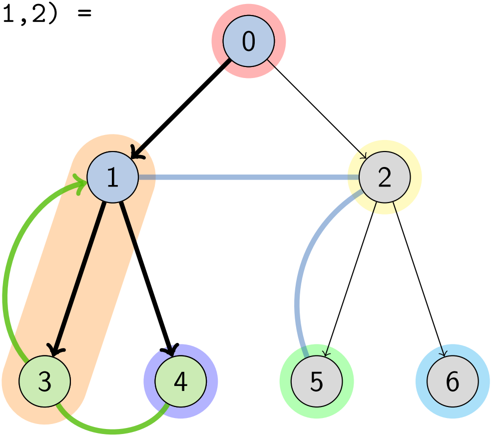
`LCA(1,2) =`



Lowest Common Ancestor

LCA Example

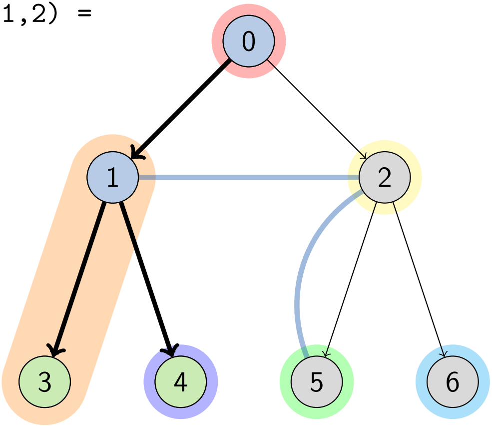
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

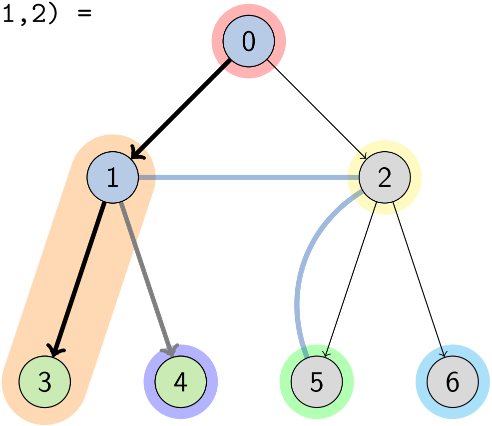
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

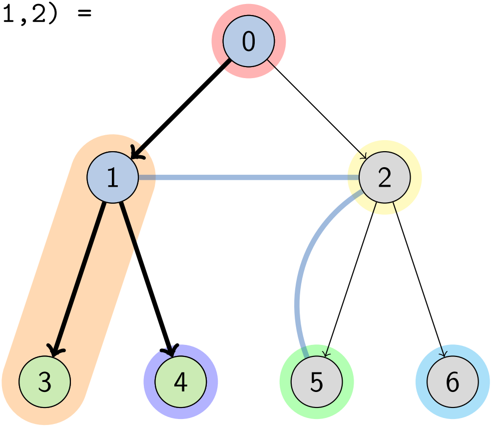
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

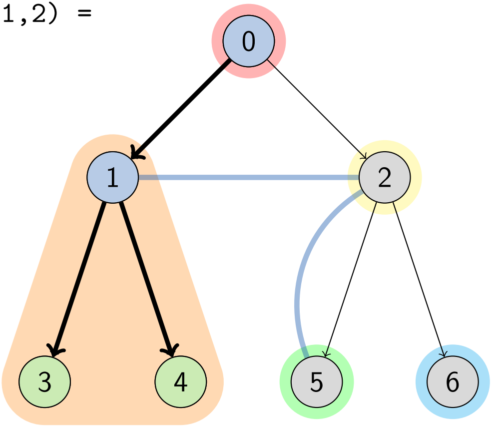
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

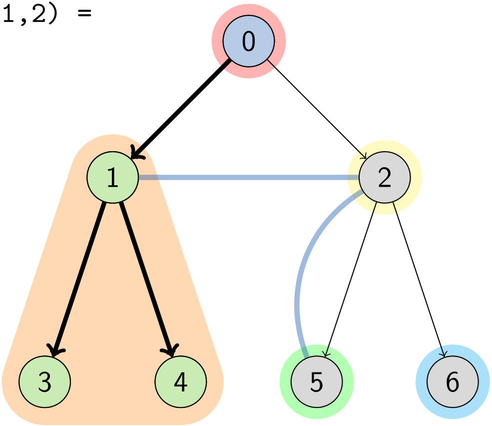
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

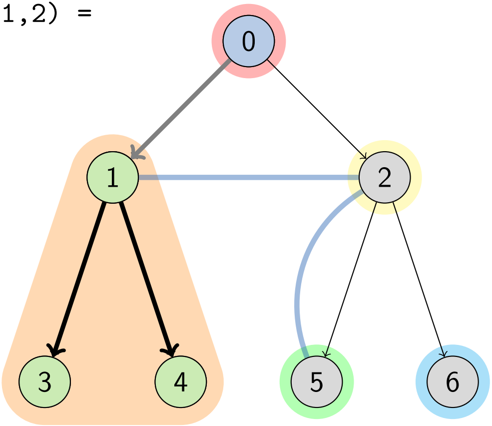
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

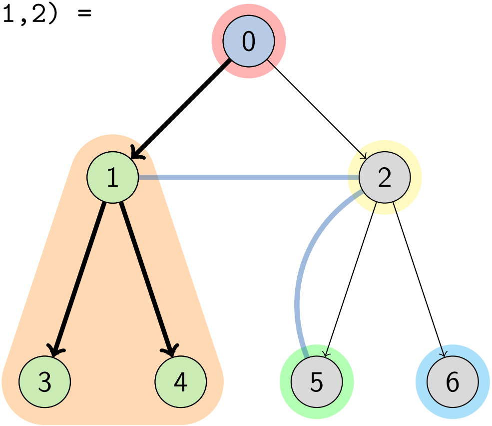
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

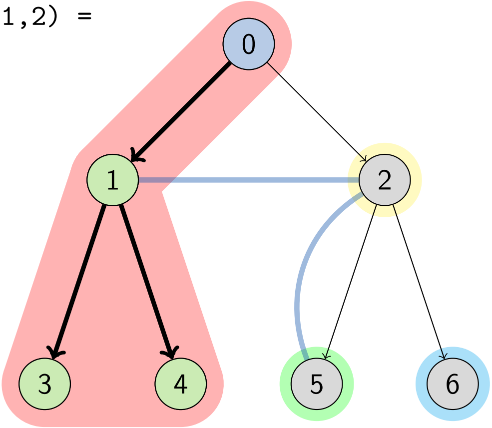
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

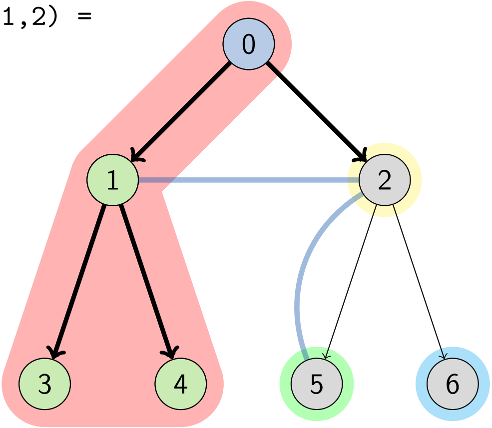
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



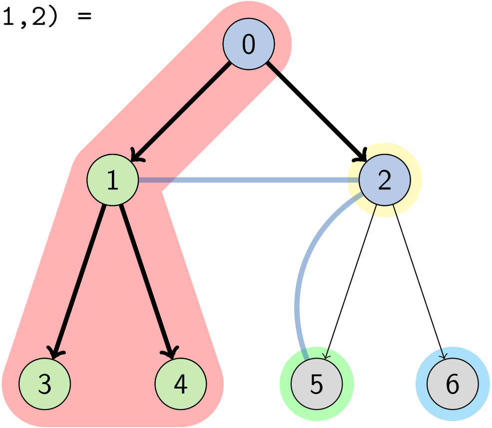
Lowest Common Ancestor

LCA Example

`LCA(3,4) = 1`

`LCA(2,5) =`

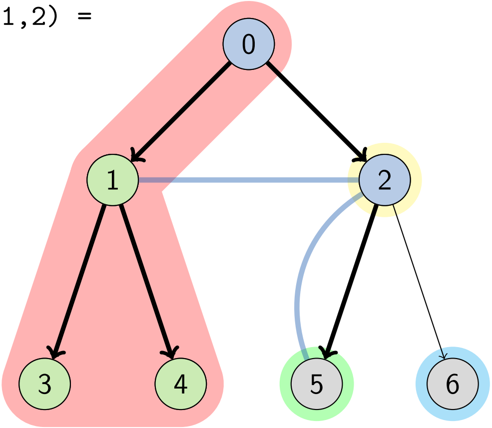
`LCA(1,2) =`



Lowest Common Ancestor

LCA Example

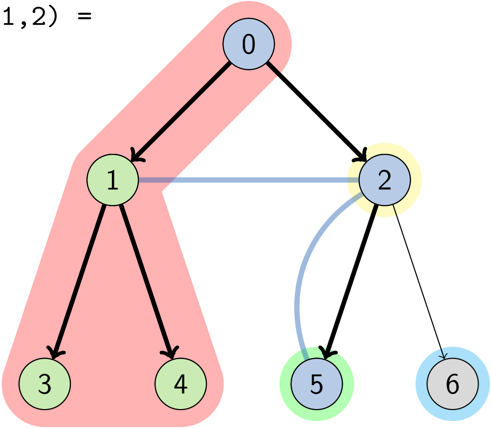
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

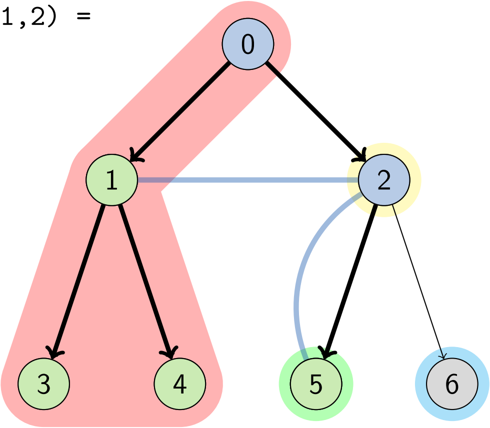
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

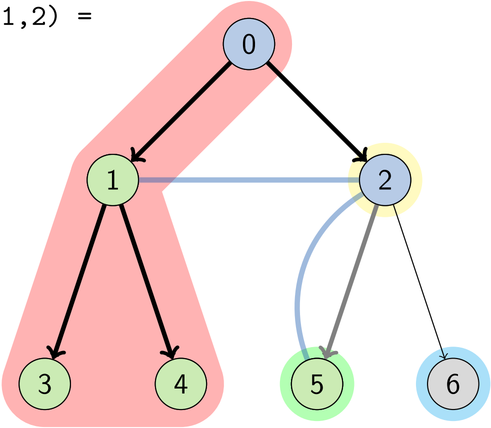
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

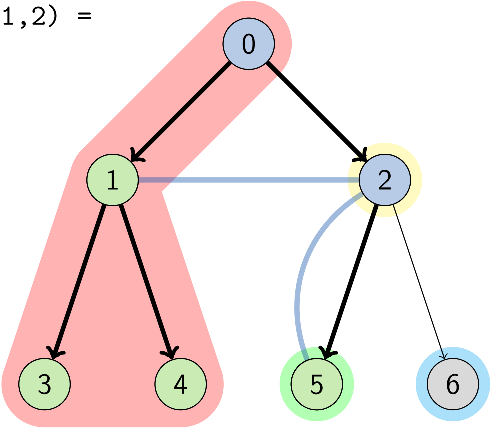
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

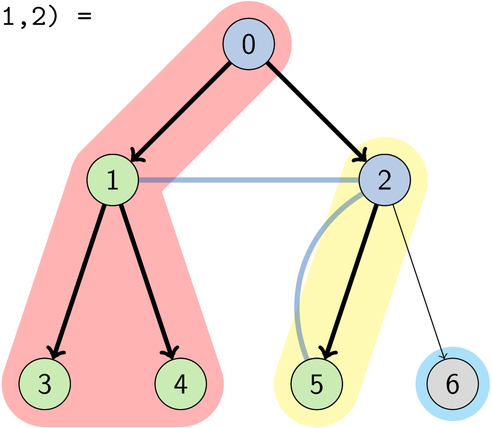
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

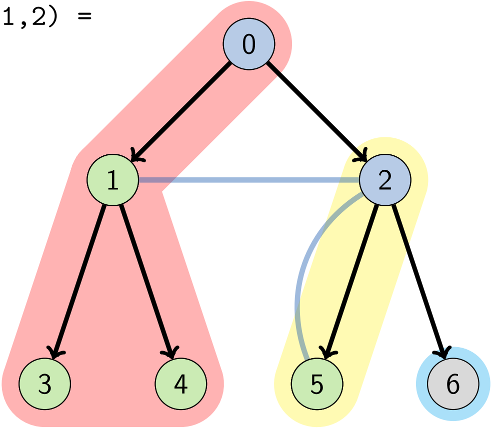
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



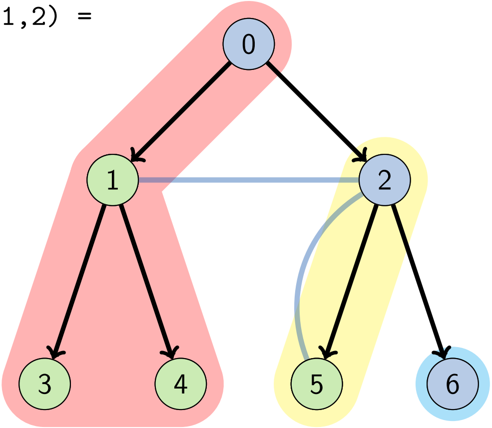
Lowest Common Ancestor

LCA Example

`LCA(3,4) = 1`

`LCA(2,5) =`

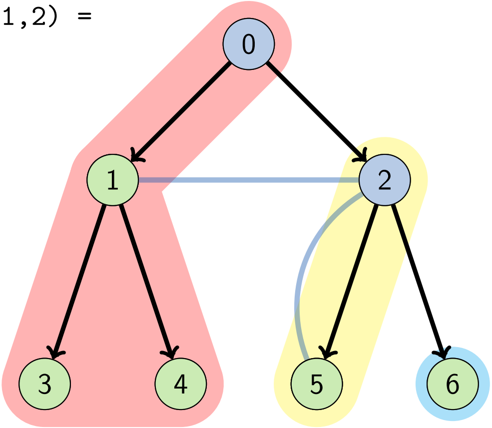
`LCA(1,2) =`



Lowest Common Ancestor

LCA Example

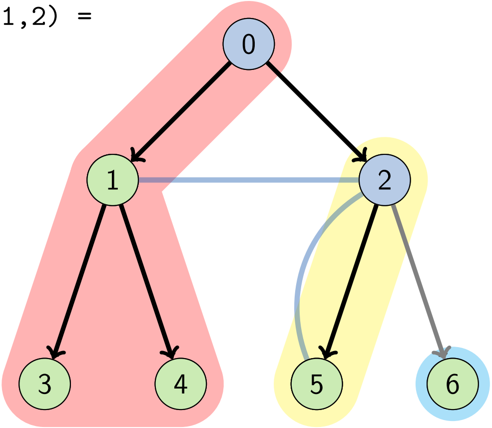
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

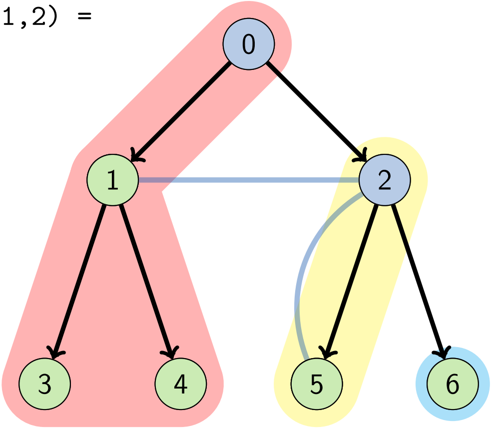
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

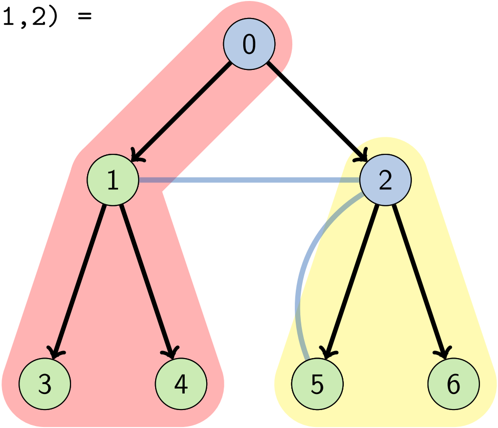
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

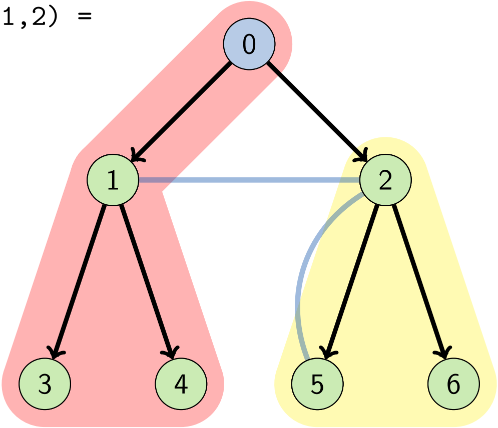
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

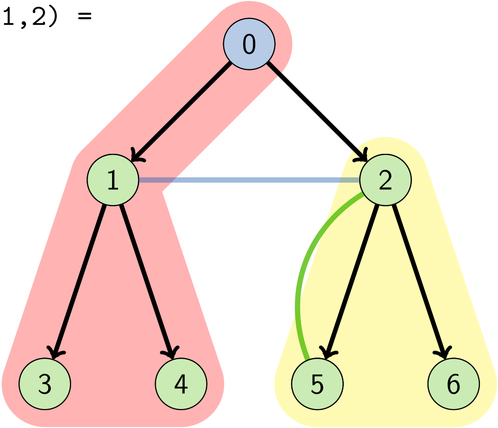
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

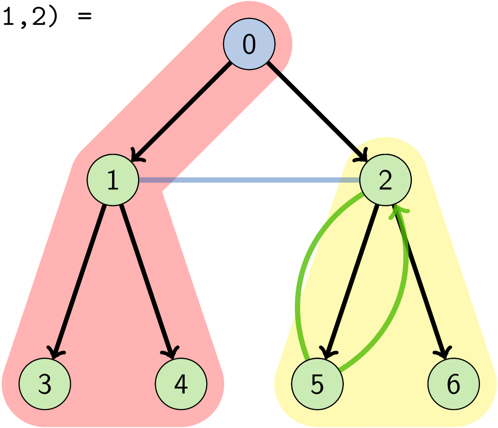
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

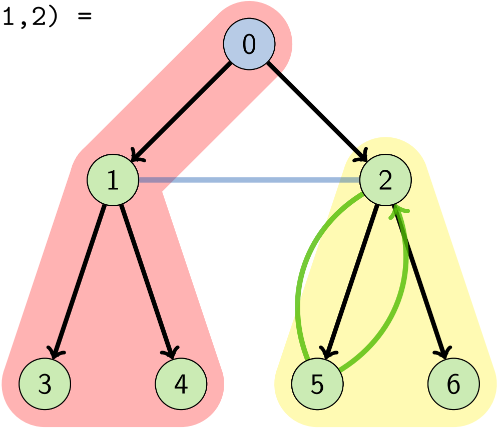
```
LCA(3,4) = 1
LCA(2,5) =
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

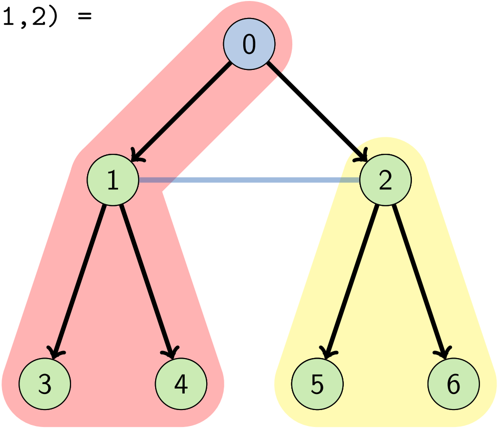
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

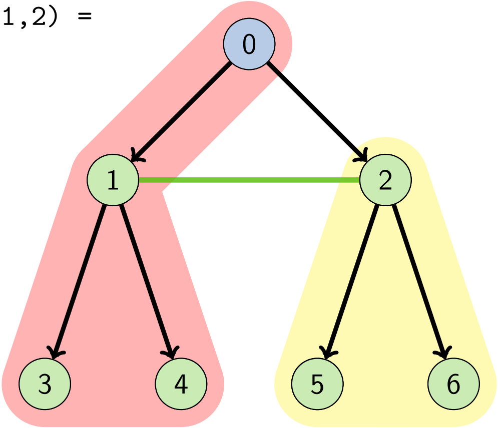
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

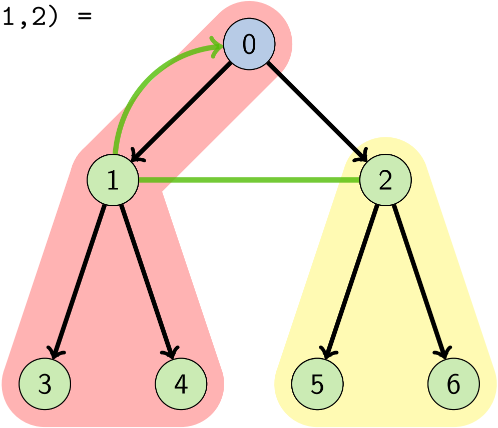
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

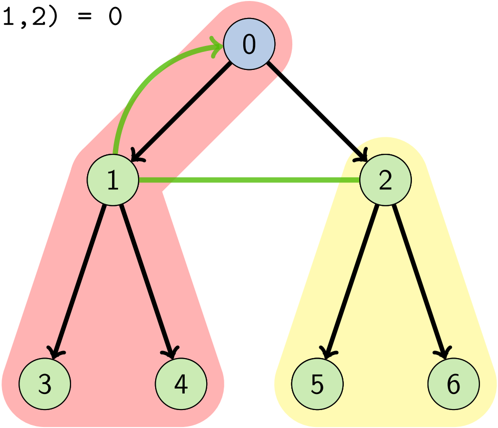
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) =
```



Lowest Common Ancestor

LCA Example

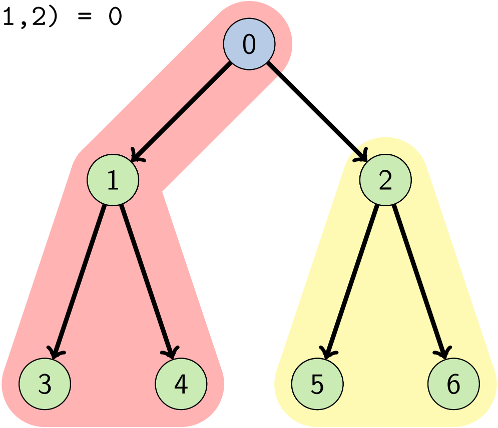
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) = 0
```



Lowest Common Ancestor

LCA Example

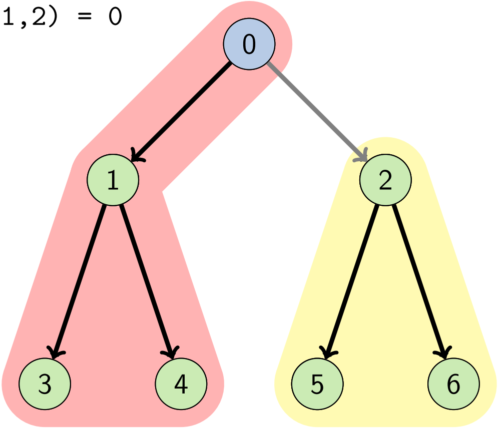
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) = 0
```



Lowest Common Ancestor

LCA Example

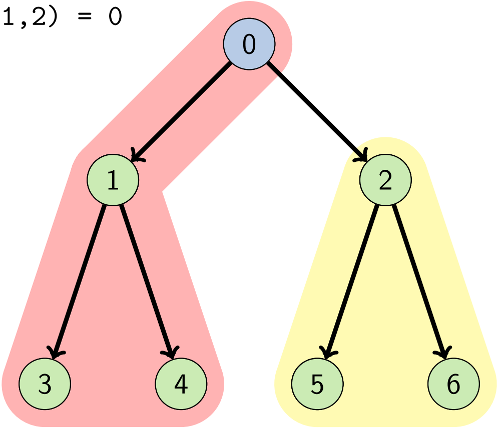
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) = 0
```



Lowest Common Ancestor

LCA Example

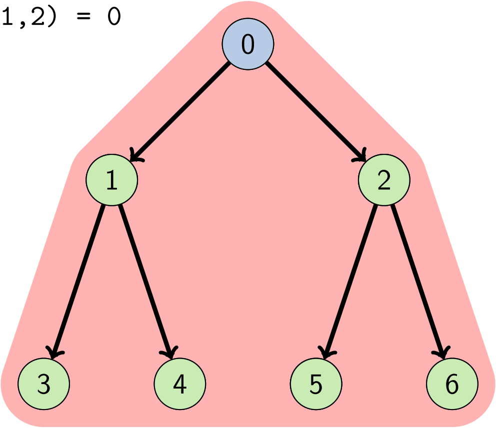
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) = 0
```



Lowest Common Ancestor

LCA Example

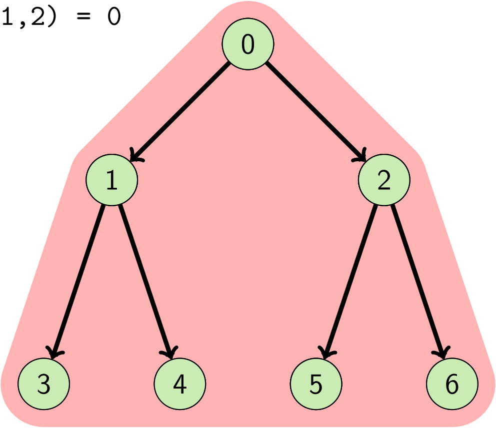
```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) = 0
```



Lowest Common Ancestor

LCA Example

```
LCA(3,4) = 1
LCA(2,5) = 2
LCA(1,2) = 0
```



Lowest Common Ancestor

Example Code: Tarjan LCA

```cpp
int V, E, root, Q;
vector<vector<int>> adj(V); // directed, rooted tree

vector<bool> visited(V, false);
auto UF = UnionFind(V);
vector<vector<int>> queries(V); // each vertex' queries
vector<int> ancestor(V);
for (int i = 0; i < V; i++) ancestor[i] = i;

void dfs (int u) {
  for (auto v: adj[u]) {
    dfs(v);
    UF.unionSet(u, v);           // combine with children
    ancestor[UF.findSet(u)] = u; // don't lose ancestor
  }
  visited[u] = true;
  for (auto v: queries[u]) // check all (u,v) query pairs
    if (visited[v])
      cout << "LCA of " << u << " and " << v <<
              " is " << ancestor[UF.findSet(v)] << endl;
}
dfs(root);
```

Trees

Recap

- Trees
- Prim's Algorithm
- Union-Find
- Kruskal's Algorithm
- LCA