Faculty of Information Technology,
Monash University

FIT2004: Algorithms and Data Structures

Week 10: Minimum Spanning Trees

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Announcements

- Start preparing for the final exam
  - Listen to the lectures (or read slides)
  - Read Lecture Notes
  - Solve tutorial questions
  - Attempt past exam papers and MSTs (available on Moodle)
  - Most importantly, do not hesitate to seek help

Recommended reading

- Lecture Notes: Chapters 14 and 15
- Cormen et al. Introduction to Algorithms.
  - Chapter 23, Pages 624-638
- http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/Undirected/
- http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/DAG/

Outline

1. Introduction
2. Prim's Algorithm
3. Kruskal's Algorithm

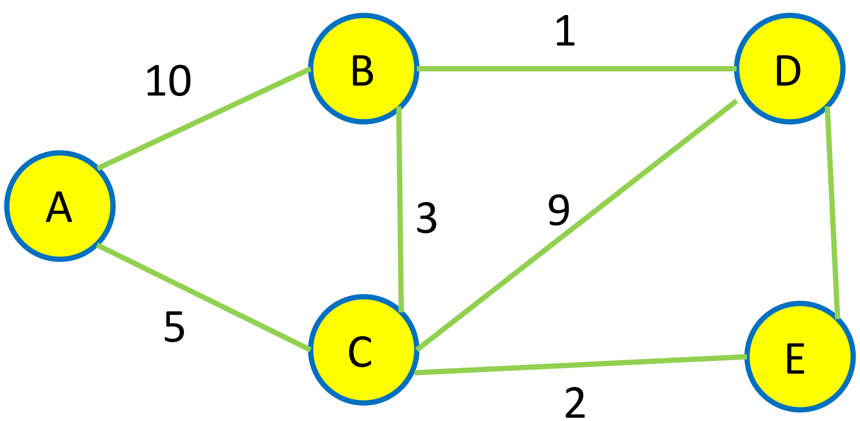
What is a Spanning Tree

Tree:

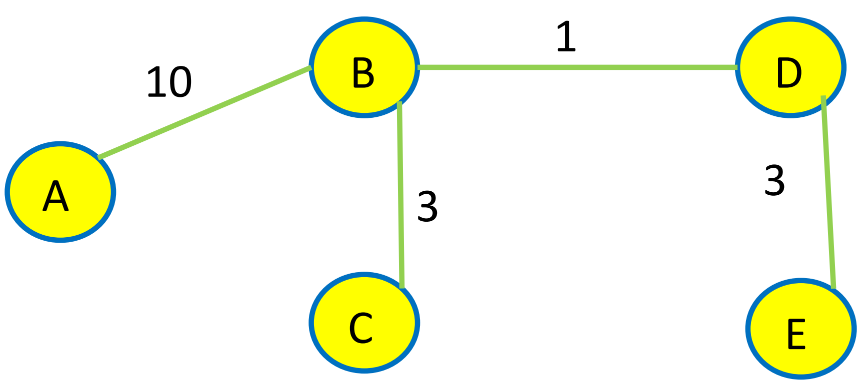
A tree is a connected undirected graph with no cycles in it.

Spanning Tree:

- A spanning tree of a general undirected weighted graph G is a tree that spans G (i.e., a tree that includes every vertex of G) and is a subgraph of G (i.e., every edge in the spanning tree belongs to G).

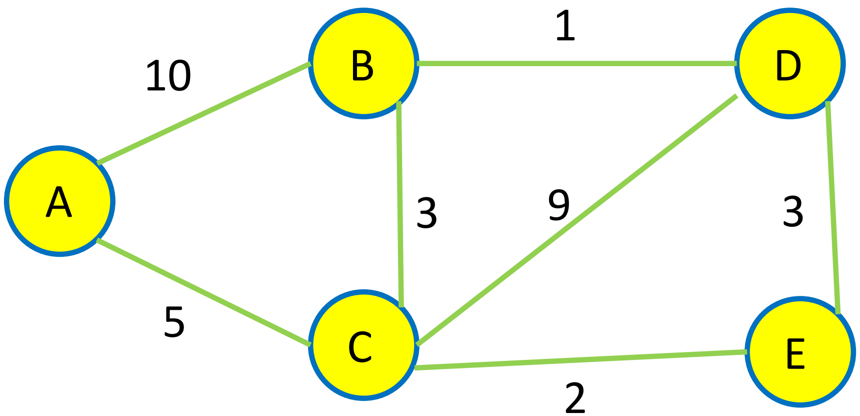


An undirected graph

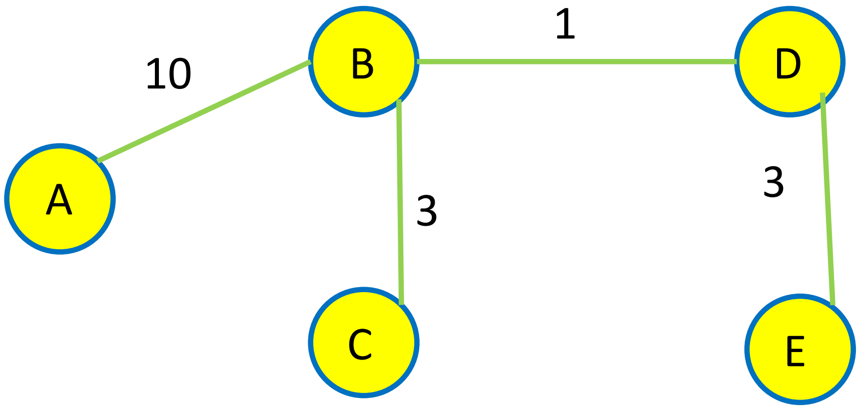


Spanning Tree

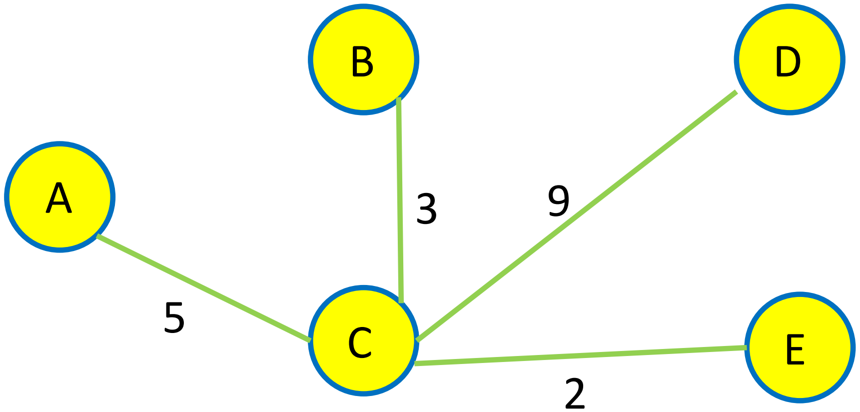
Spanning Tree Examples



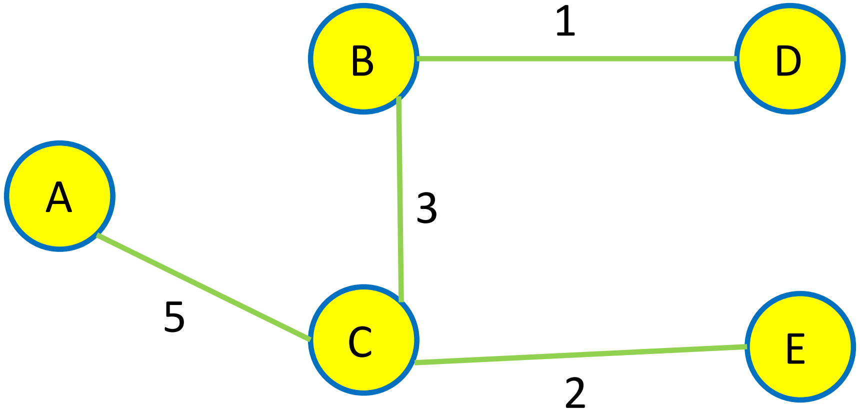
Graph



Spanning Tree 1



Spanning Tree 2



Spanning Tree 3

What is a Spanning Tree

Tree:

A tree is a connected undirected graph with no cycles in it.

Spanning Tree:

- A spanning tree of a general undirected weighted graph G is a tree that spans G (i.e., a tree that includes every vertex of G) and is a subgraph of G (i.e., every edge in the spanning tree belongs to G).

Is it true that a spanning tree of a connected graph G is a maximal set of edges of G that contains no cycles?

- Yes

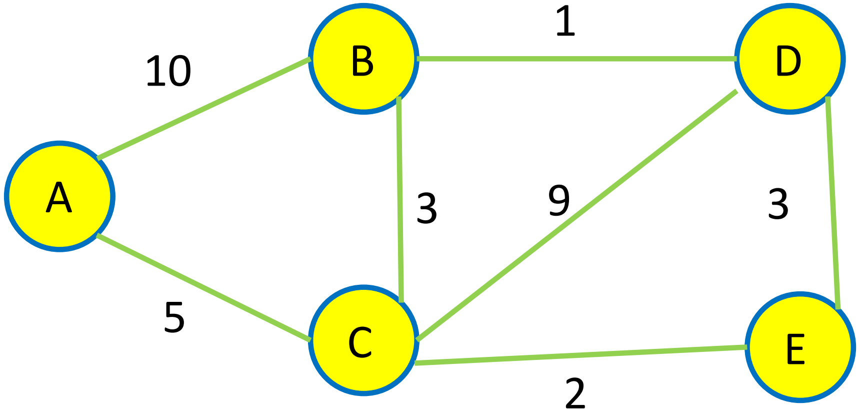
Is it true that a spanning tree of a connected graph G is a minimal set of edges that connect all vertices?

- Yes

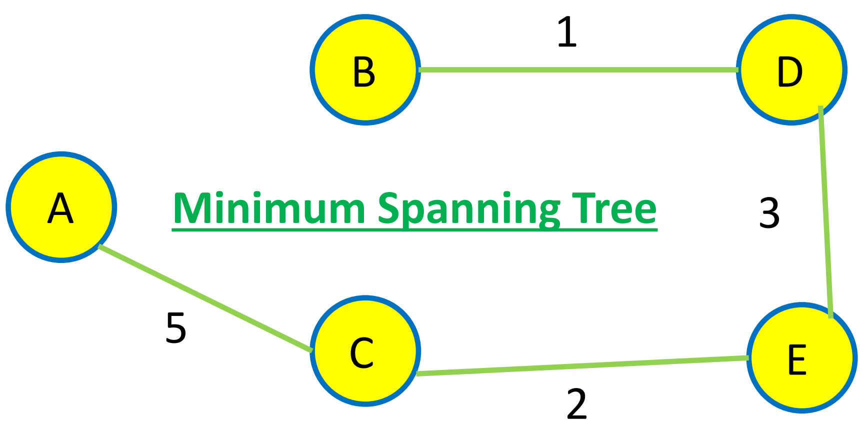
Minimum Spanning Tree (MST)

- Weight of a spanning tree is the sum of the weights of the edges in the tree.
- A Minimum spanning tree of a weighted general graph G is a tree that spans G, whose weight is minimum over all possible spanning trees for this graph.
- There may be more than one minimum spanning tree for a graph G (e.g., two or more spanning trees with the same minimum weight).

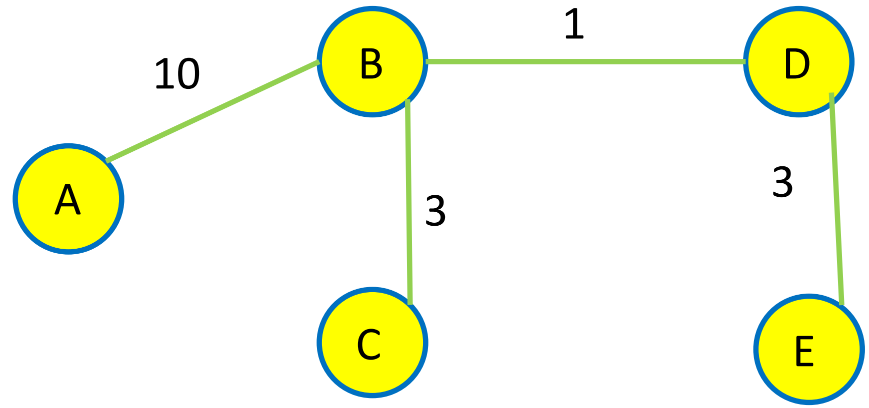
What is the weight of the MST in this graph?
How many MSTs are in this graph?



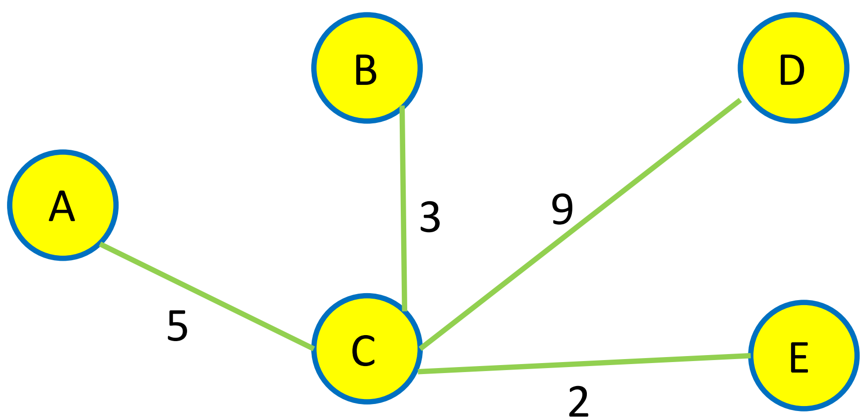
Spanning Trees and MSTs



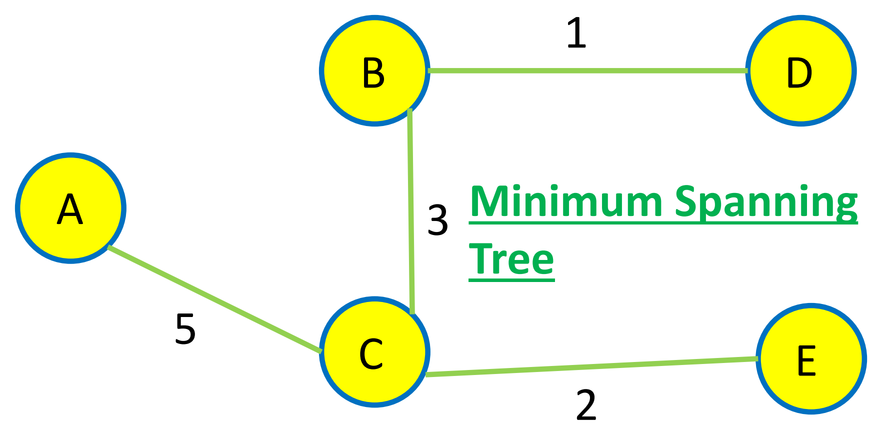
Spanning Tree 1: Weight 11



Spanning Tree 2: Weight 17



Spanning Tree 3: Weight 19



Spanning Tree 4: Weight 11

MST Algorithms

Let M denote the MST we are constructing, initialized to be empty

An edge e is said to be safe if {M U e} is a subset of some MST

General Strategy:

- M = null
- **while** M can be grown safely:
- find an edge e=<x,y> along which M **is** safe to grow
- M = {M} union {<x,y>}
- **return** M

We will study two **greedy** algorithms that follow this strategy

MST Algorithms

- Prim's Algorithm (very similar to Dijkstra's Algorithm)
  - In fact, Dijkstra published his algorithm for both MST and shortest path in the same paper (1959)
  - The algorithm is also often called Prim-Dijkstra Algorithm
  - M is always a tree and, in each iteration, we choose the shortest edge connected to M avoiding cycles
  - Time complexity: O(E log V)

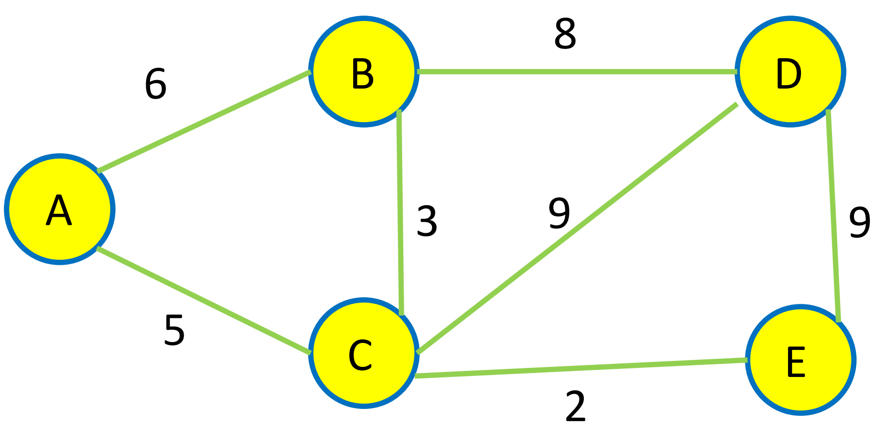
- Kruskal's Algorithm
  - We process edges in ascending order of edge weights and M is a forest (i.e., a set of trees)
  - Time complexity: O(E log V)

Outline

1. Introduction
2. Prim's Algorithm
3. Kruskal's Algorithm

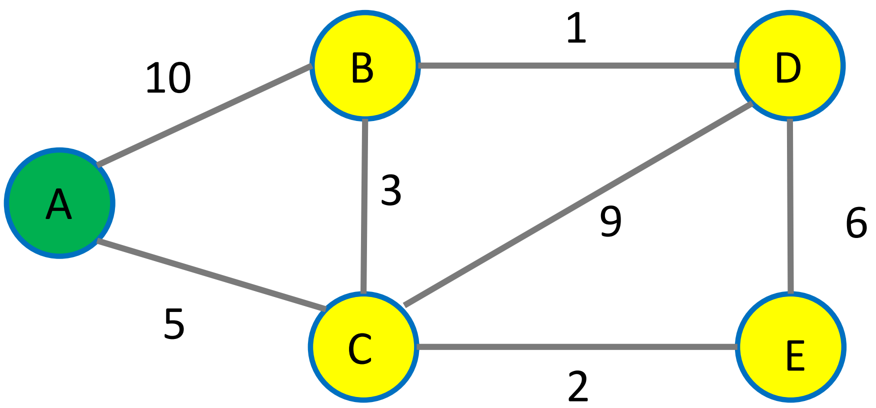
Prim's Algorithm: Overview

- Start by picking any vertex v to be the root of the tree M.
- While the tree M does not contain **all** vertices in the graph
  - Find shortest edge e connected to the growing subtree M that does not create a cycle
  - add e to the tree M



Prim's Algorithm

u:



Q:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Q is a priority queue, where priority is based on distance

Pred and Dist are the usual ID-indexed arrays

Pred:

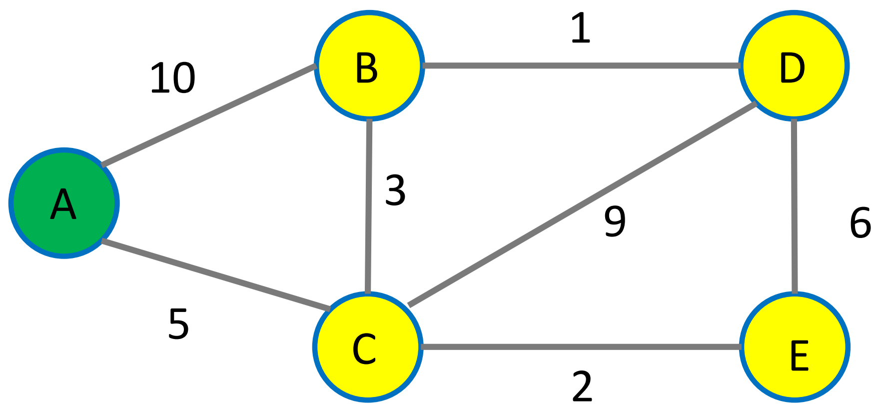
A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

Prim's Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

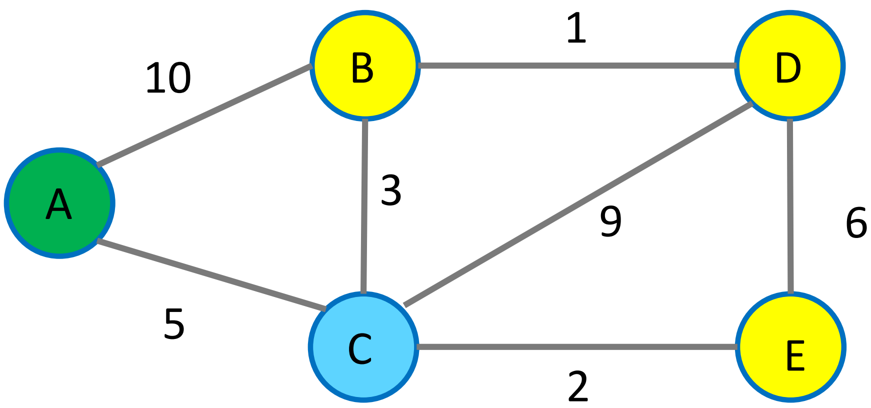
A	B	C	D	E
0	Inf	Inf	Inf	inf

Q is a priority queue, where priority is based on distance

Pred and Dist are the usual ID-indexed arrays

Prim's Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

For each neighbour v of u, try to update distance

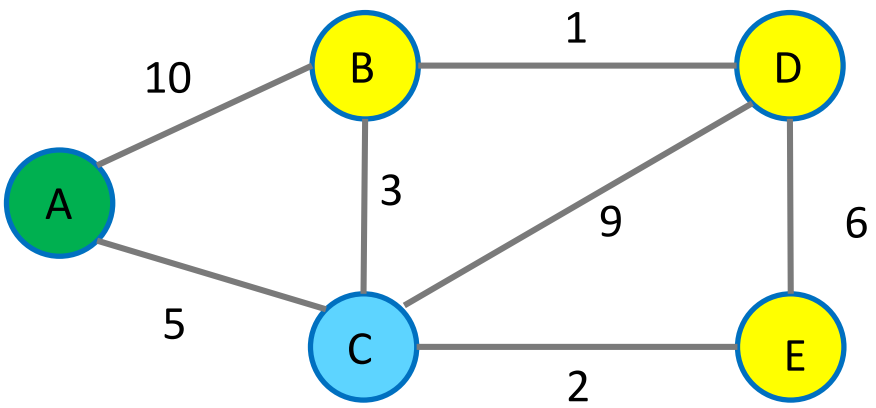
5 < inf

Dist[C] = 5

Pred[C] = A

Prim's Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

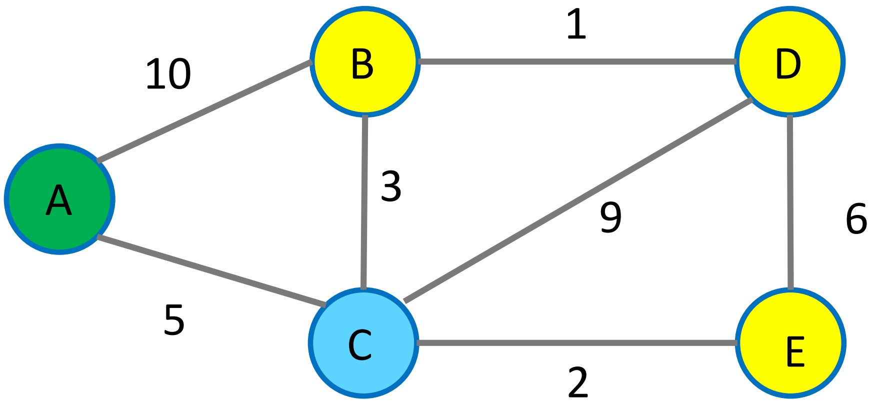
Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

For each neighbour v of u, try to update distance

This time we just care about the edge, not the distance to u + the edge

Prim's Algorithm



u: A

Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

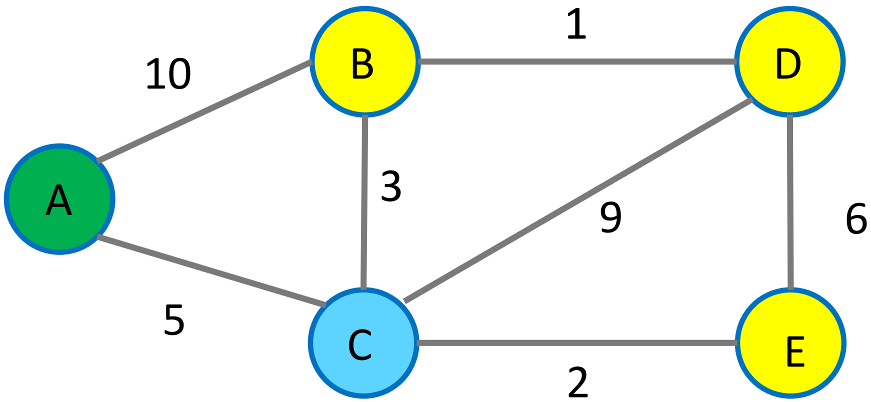
For each neighbour v of u, try to update distance

W(A,C) = 5
Dist[C] = inf

5 < inf, so update
Dist[C] = 5
Pred[C] = A

Prim’s Algorithm

u: A



Q:

C	B	D	E
5	Inf	Inf	Inf

Note that this changes the order of Q

Pred:

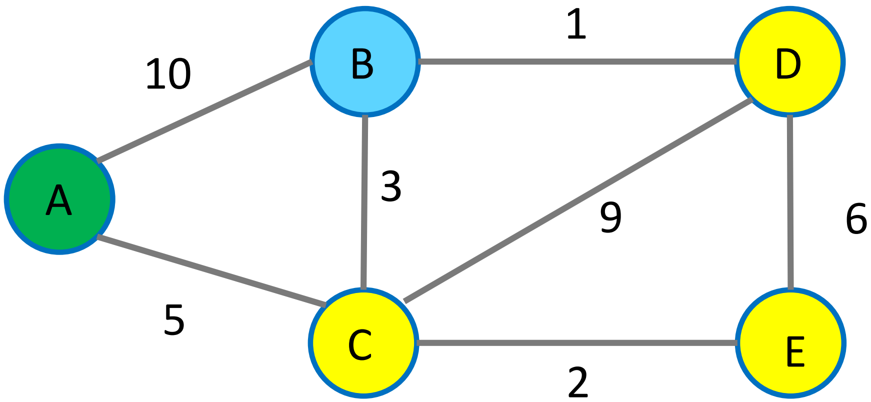
A	B	C	D	E
-	-	A	-	-

Dist:

A	B	C	D	E
0	Inf	5	Inf	inf

Prim's Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

Doing the same for B
- 0 + 10 < inf
- Dist[B] = 10
- Pred[B] = A

Pred:

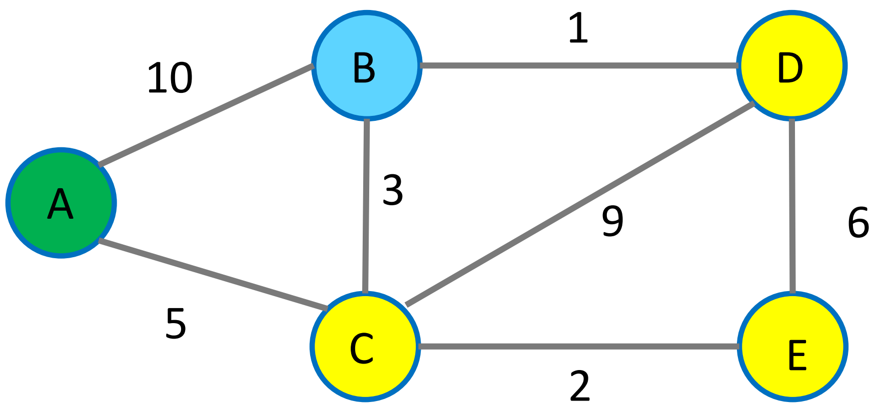
A	B	C	D	E
-	-	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	inf

Prim’s Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

Doing the same for B

- 0 + 10 < inf
- Dist[B] = 10
- Pred[B] = A

Pred:

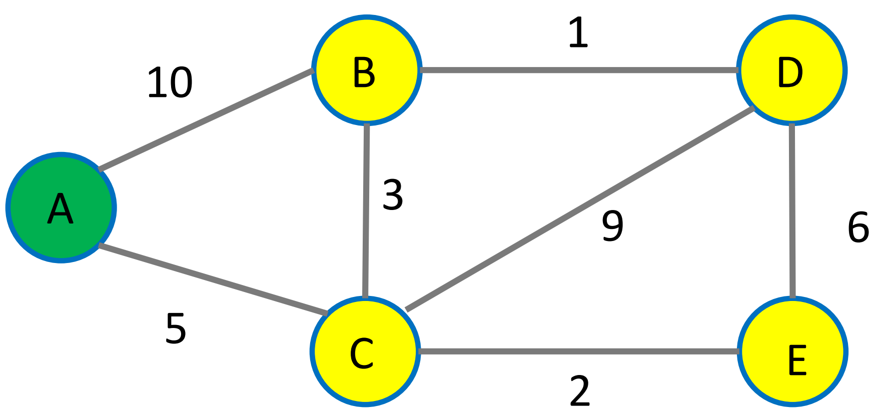
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	inf

Prim's Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

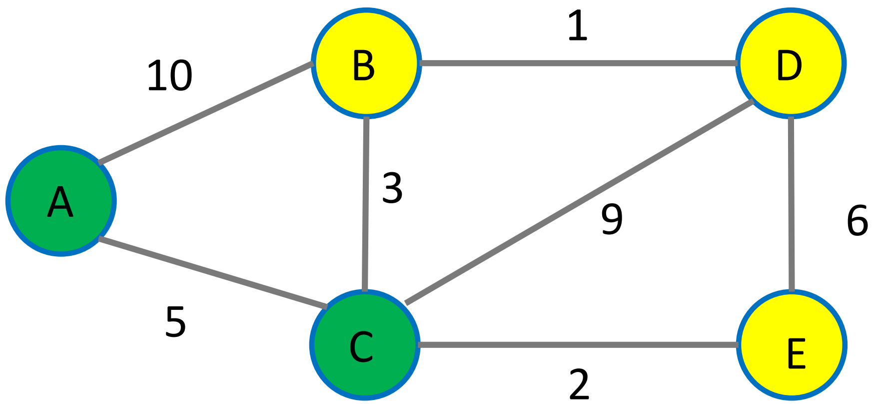
Dist:

A	B	C	D	E
0	10	5	Inf	inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest dist

Prim's Algorithm

u: C



Q:

B	D	E
10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

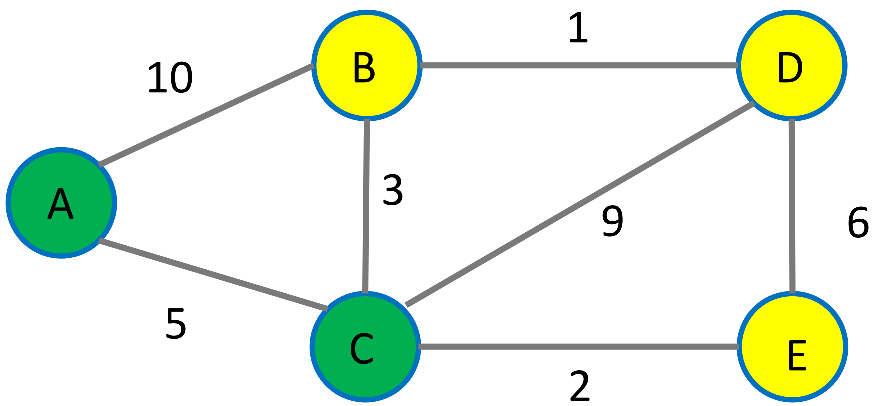
Dist:

A	B	C	D	E
0	10	5	Inf	inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest dist
- This vertex is now in the MST

Prim's Algorithm

u: C



Q:

B	D	E
10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

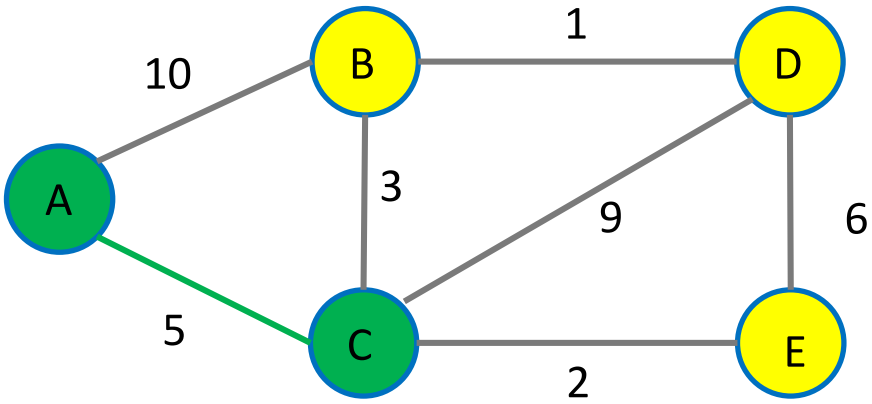
Dist:

A	B	C	D	E
0	10	5	Inf	inf

- The edge we add is the edge between u and pred[u]
- So in this case, A->C

Prim's Algorithm

u: C



Q:

B	D	E
10	Inf	Inf

Pred:

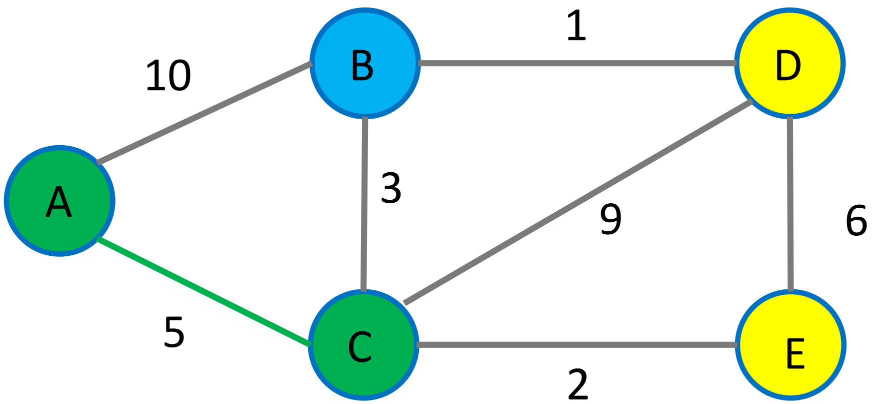
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	inf

- The edge we add is the edge between u and pred[u]
- So in this case, A->C

Prim's Algorithm



u: C

Q:

B	D	E
10	Inf	Inf

- Update B from C
- w(C, B) = 3
- Dist[B] = 10
- So update dist[B] and pred[B]

Pred:

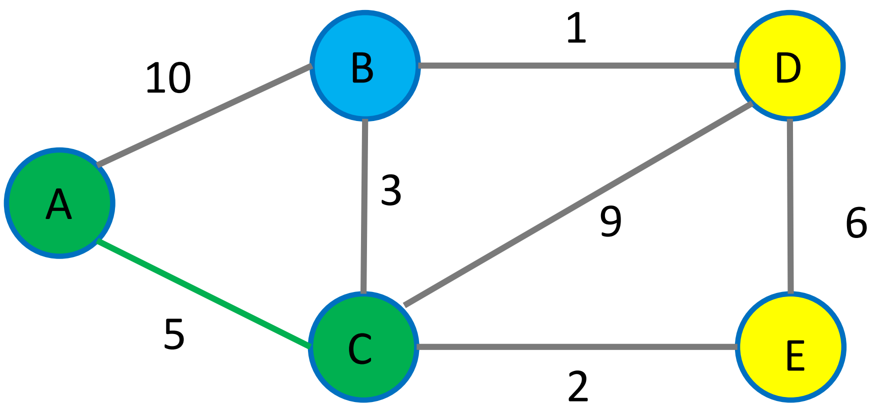
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	inf

Prim's Algorithm

u: C



Q:

B	D	E
3	Inf	Inf

Pred:

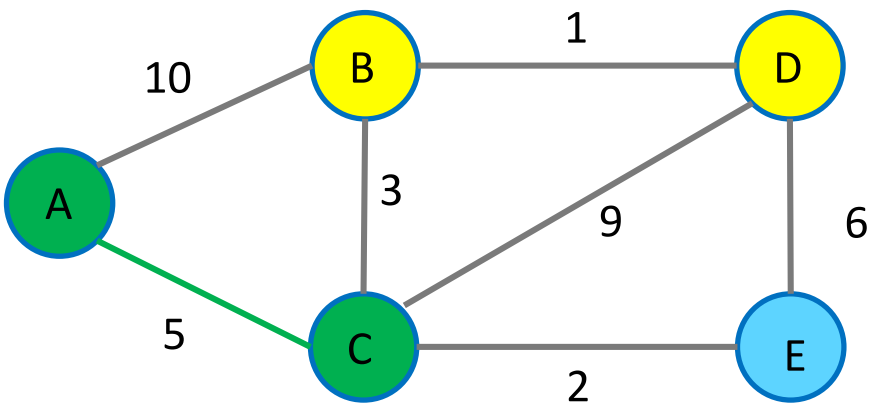
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	3	5	Inf	inf

Prim's Algorithm

u: C



Q:

B	D	E
3	Inf	Inf

Pred:

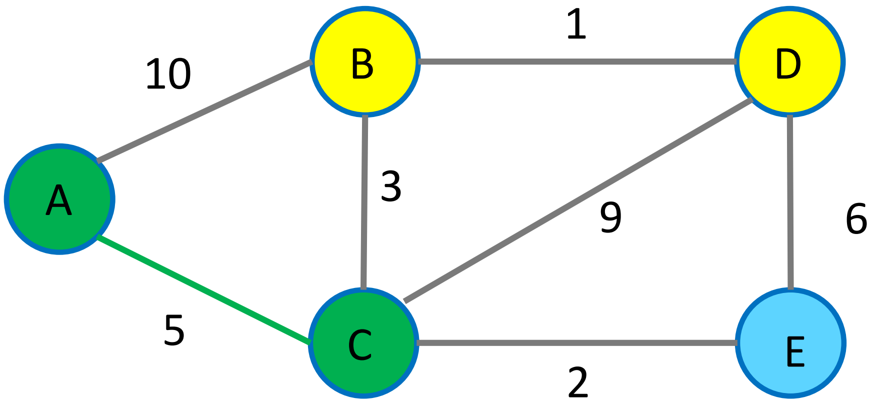
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	3	5	Inf	inf

- update E from C

Prim's Algorithm



u: C

Q:

E	B	D
2	3	inf

Pred:

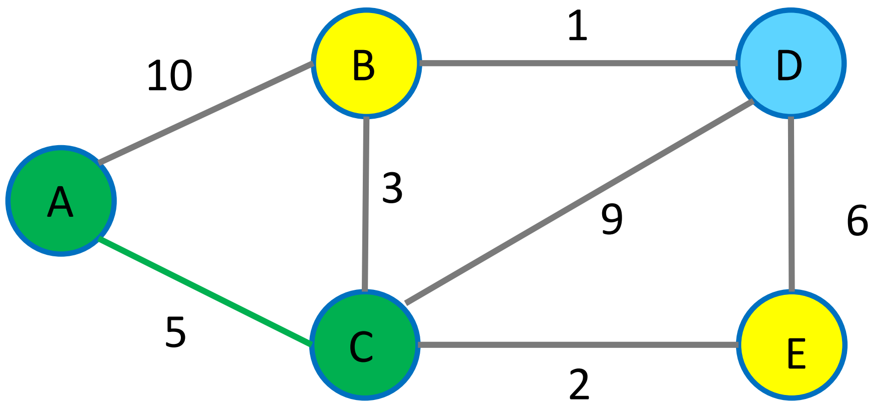
A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	3	5	Inf	2

Prim’s Algorithm

u: C



Q:

E	B	D
2	3	inf

- Update D from C

Pred:

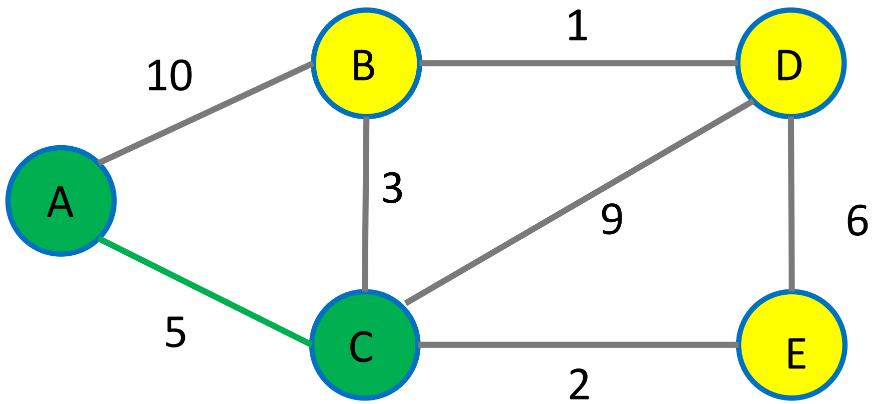
A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	3	5	Inf	2

Prim's Algorithm

u: C



Q:

E	B	D
2	3	9

- Done with C
- Pop another vertex from Q and add it to the MST

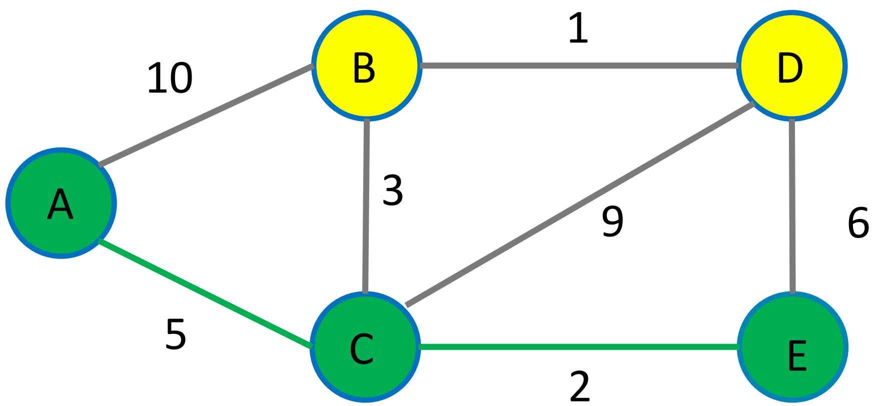
Pred:

A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	3	5	9	2

Prim’s Algorithm



u: E

Q:

B	D
3	9

Pred:

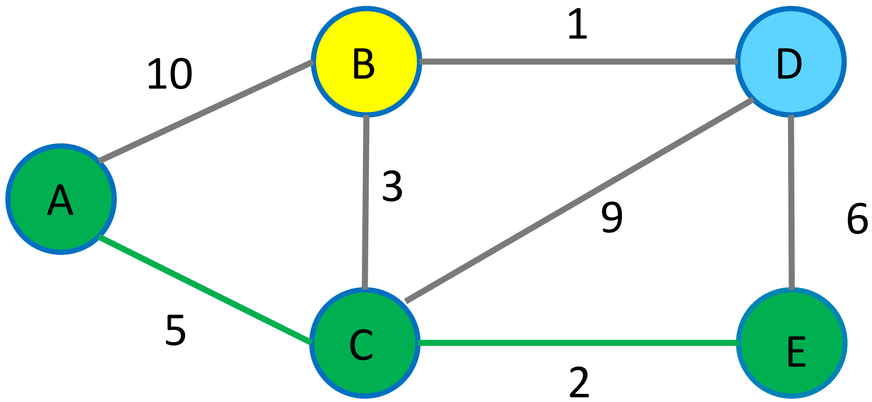
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	3	5	9	2

Prim's Algorithm

u: E



Q:

B	D
3	6

Pred:

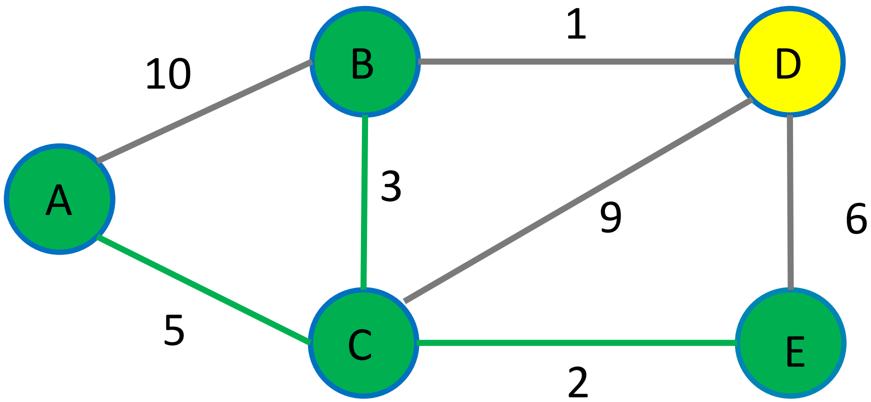
A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	3	5	6	2

Prim's Algorithm

u: B



Q:

D
6

Pred:

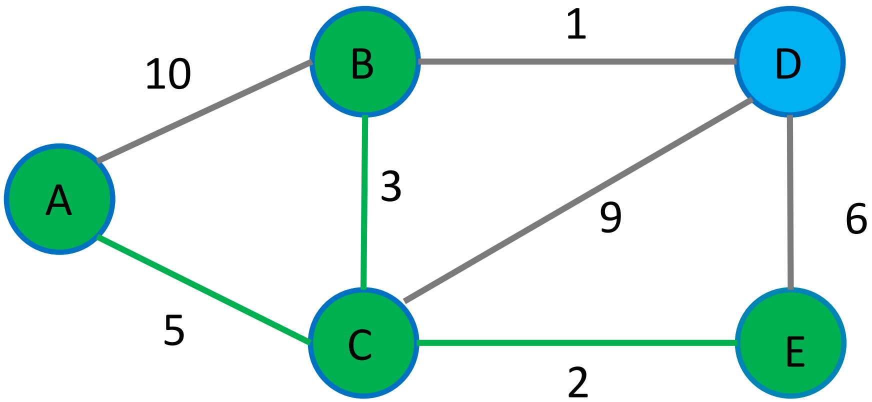
A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	3	5	6	2

Prim's Algorithm

u: B



Q:

D
6

Pred:

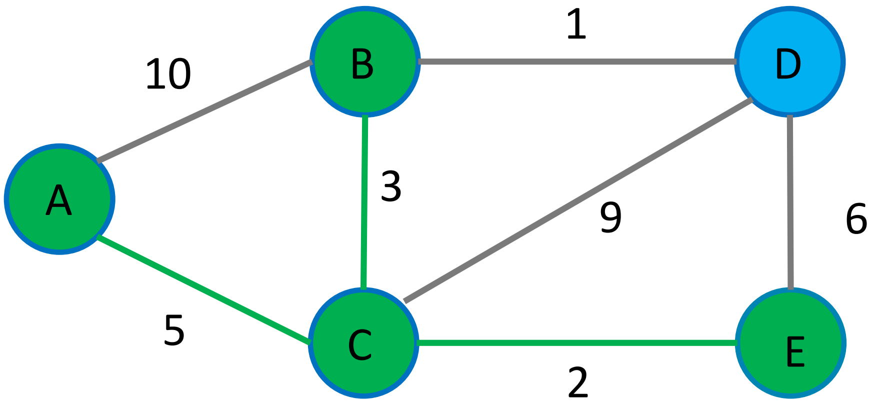
A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	3	5	6	2

Prim’s Algorithm

u: B



Q:

D
1

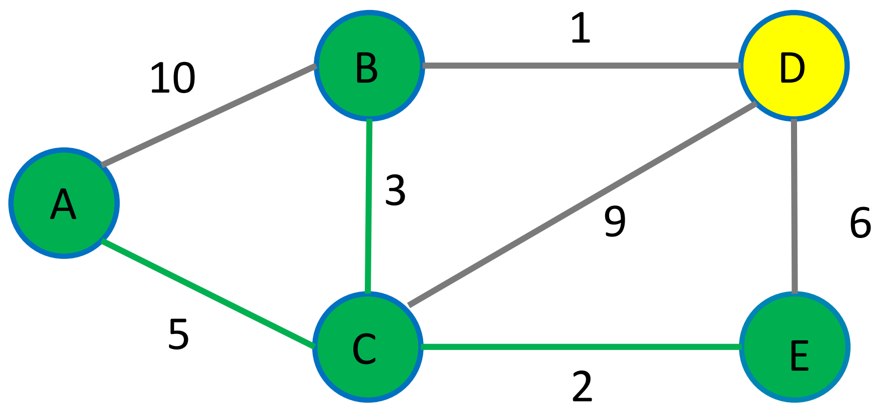
Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim's Algorithm



u: B

Q:

D
1

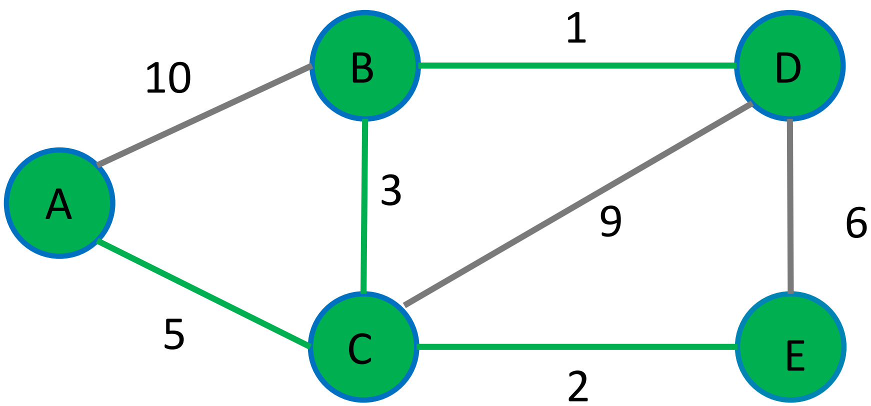
Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim's Algorithm



u: D

Q:

Pred:

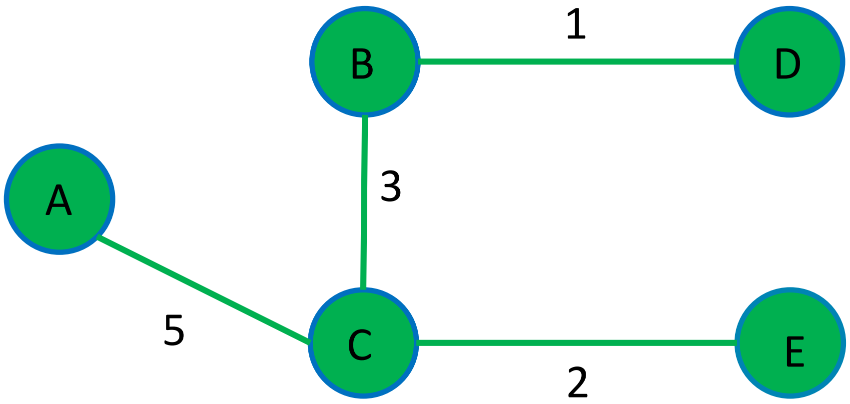
A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim's Algorithm

u: D



Q:

Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim's Algorithm

Algorithm 69 Prim's algorithm

```
function PRIM(G = (V, E), r)
    dist[1..n] = ∞
    parent[1..n] = null
    T = ({r}, ∅)
    dist[r] = 0
    Q = priority_queue(V[1..n], key(v) = dist[v])
    while Q is not empty do
        u = Q.pop_min()
        T.add_vertex(u)
        T.add_edge(parent[u], u)
        for each edge e = (u, v) adjacent to u do
            if not v ∈ T and dist[v] > w(u, v) then
                // Remember to update the key of v in the priority queue!
                dist[v] = w(u, v)
                parent[v] = u
    return T
```

Prim's Algorithm: Complexity

It is very similar to Dijkstra's Algorithm and its complexity is the same as Dijkstra's Algorithm

- O(V log V + E log V) if min-heap is used

Since the input graph G is connected, E ≥ V-1. Hence, the complexity can be simplified to O(E log V).

Prim's Algorithm: Correctness

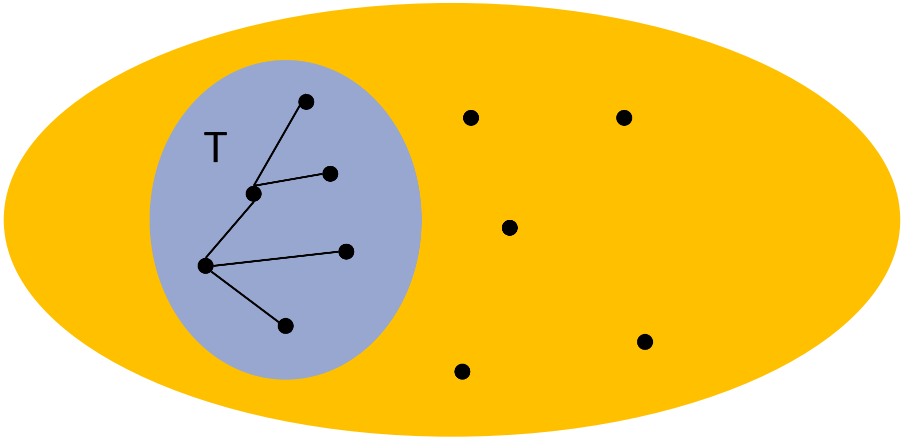
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**



Prim's Algorithm: Correctness

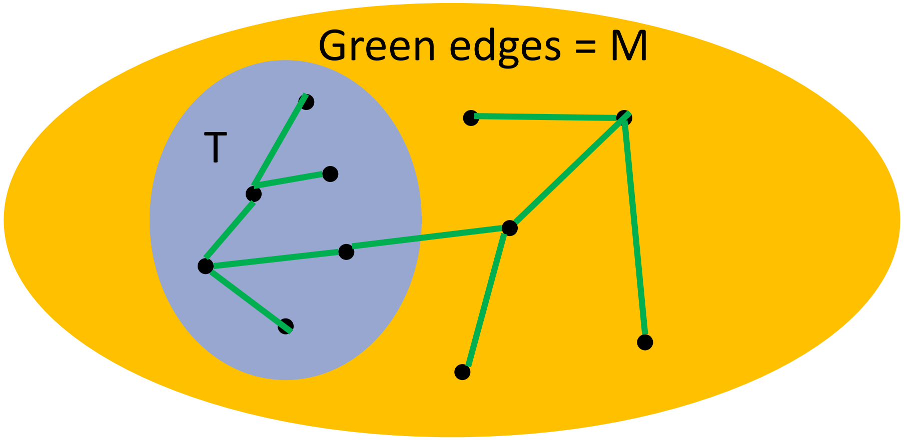
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**



Prim's Algorithm: Correctness

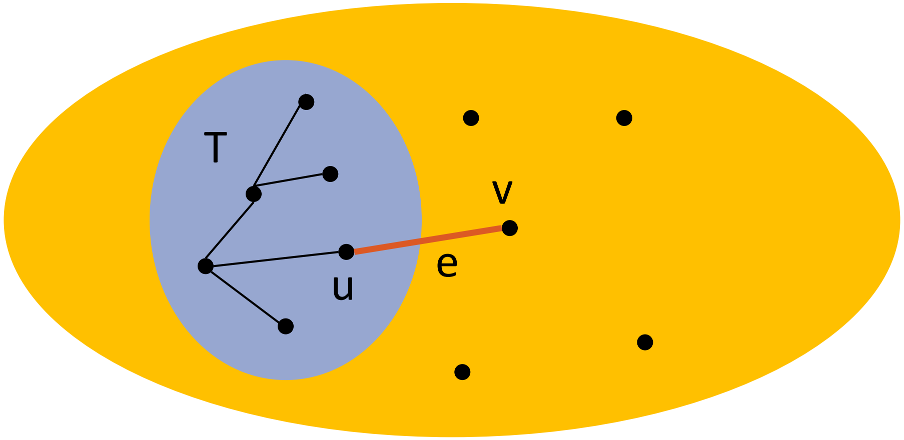
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **v** in **T** to some **u** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then T ∪ {e} is a subset of **M**, which is an MST, so the invariant holds



Prim's Algorithm: Correctness

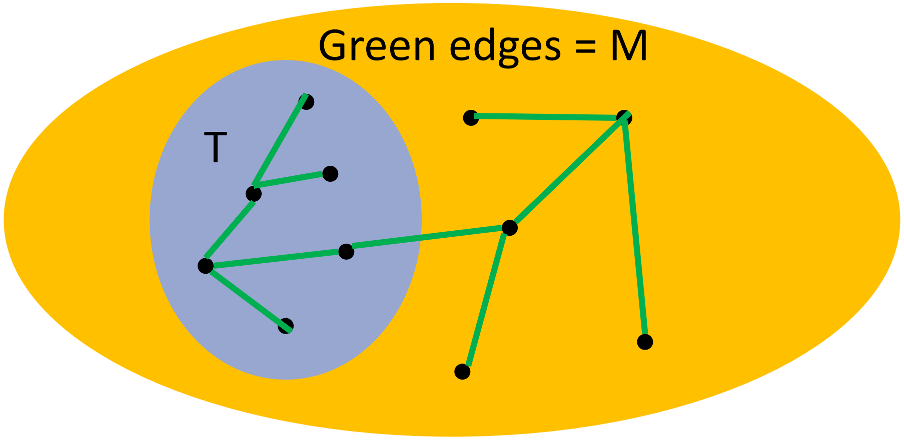
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **v** in **T** to some **u** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then **T U {e}** is a subset of **M**, which is an MST, so the invariant holds



Prim's Algorithm: Correctness

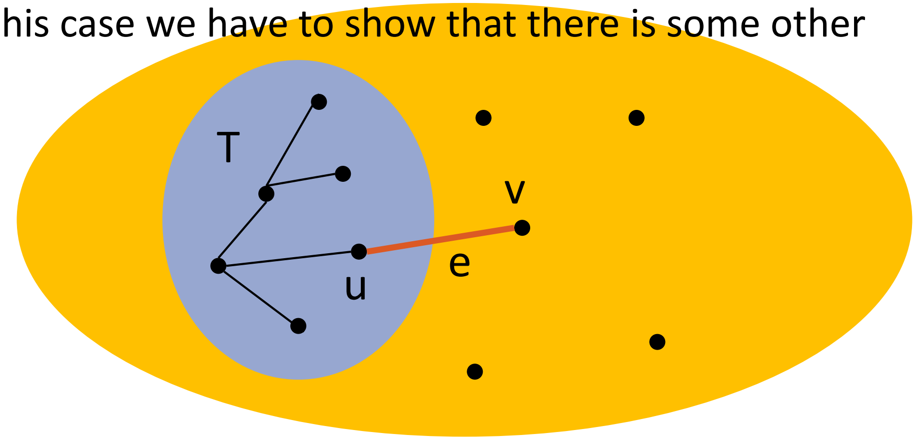
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **v** in **T** to some **u** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds
- The interesting case is where **e** is not in **M**. In this case we have to show that there is some other MST which contains **T ∪ {e}**



Prim's Algorithm: Correctness

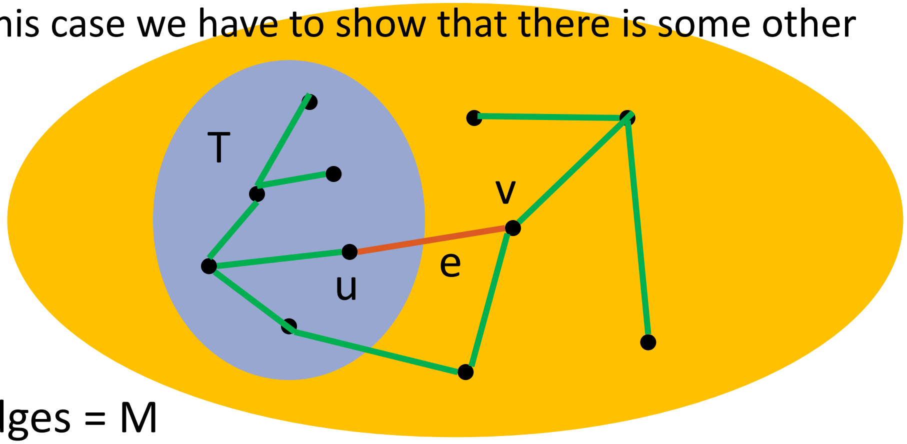
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects some **v** in **T** to some **u** not in **T** (i.e. this is the edge Prim's algorithm will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds
- The interesting case is where **e** is not in **M**. In this case we have to show that there is some other MST which contains **T ∪ {e}**

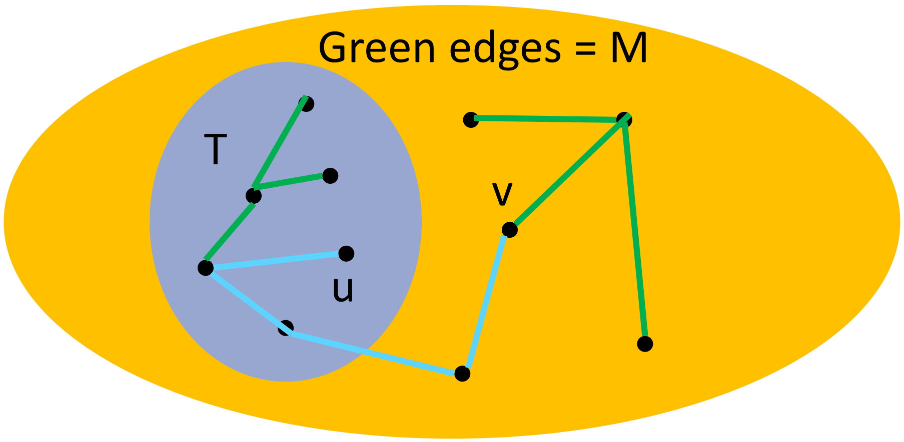


Green edges = M

Prim's Algorithm: Correctness

Inductive step:

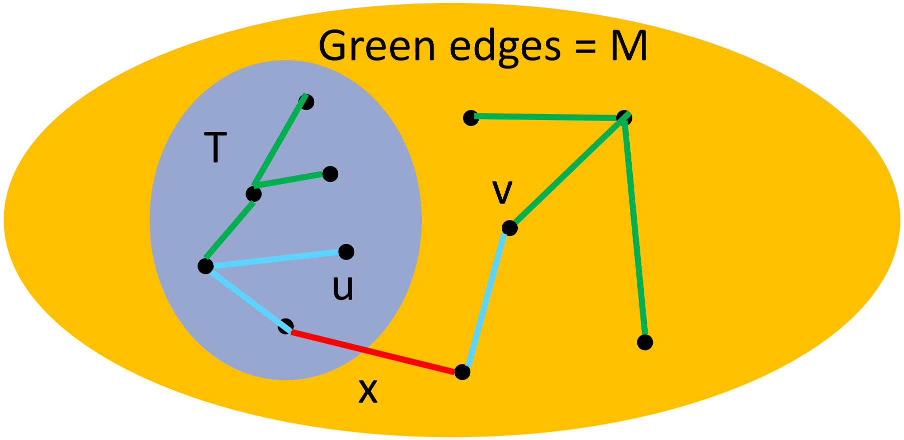
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).



Prim's Algorithm: Correctness

Inductive step:

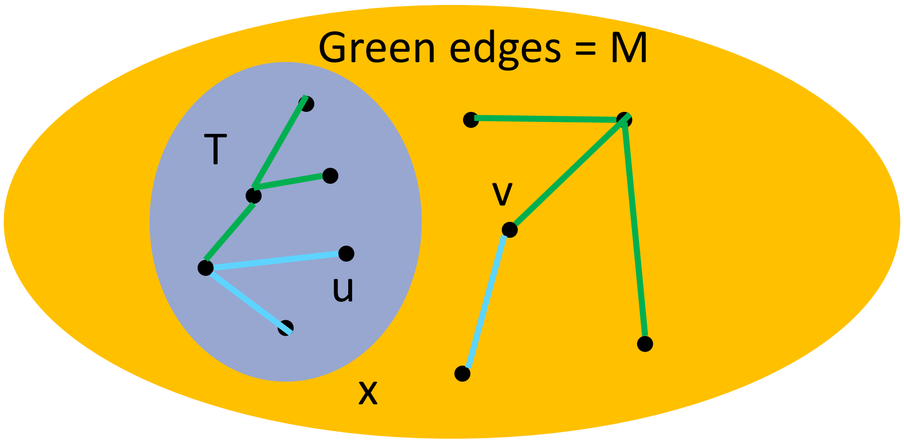
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).
- One vertex of this edge is in **T**, the other is not.
- Removing this edge would disconnect **M**



Prim’s Algorithm: Correctness

Inductive step:

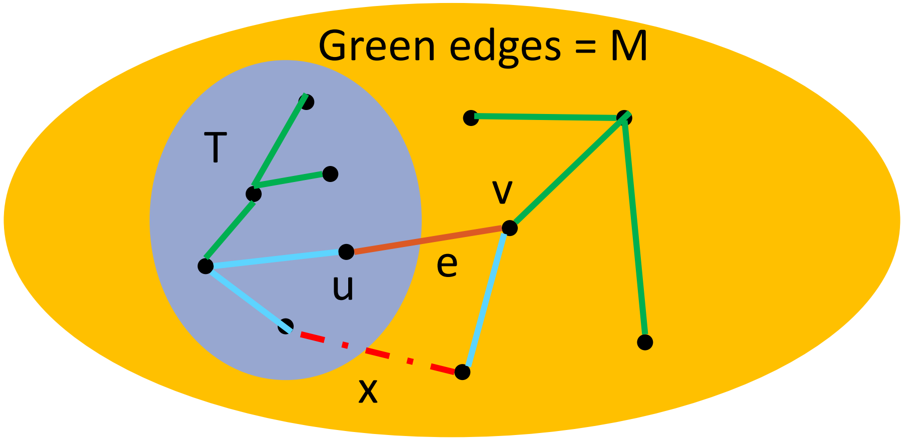
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).
- One vertex of this edge is in **T**, the other is not.
- Removing this edge would disconnect **M**



Prim's Algorithm: Correctness

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since **v** is not in **T**). Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).
- One vertex of this edge is in **T**, the other is not.
- Removing this edge would disconnect **M**
- Adding the edge **(u,v)** would form a new spanning tree, **M'**
- Since the algorithm always selects the lightest edge incident to **T**, we know that **w(e) ≤ w(x)**
- So the weight of **M'** is no greater than the weight of **M**, therefore choosing e is correct



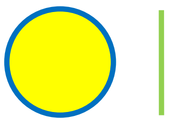
Outline

1. Introduction
2. Prim's Algorithm
3. Kruskal's Algorithm

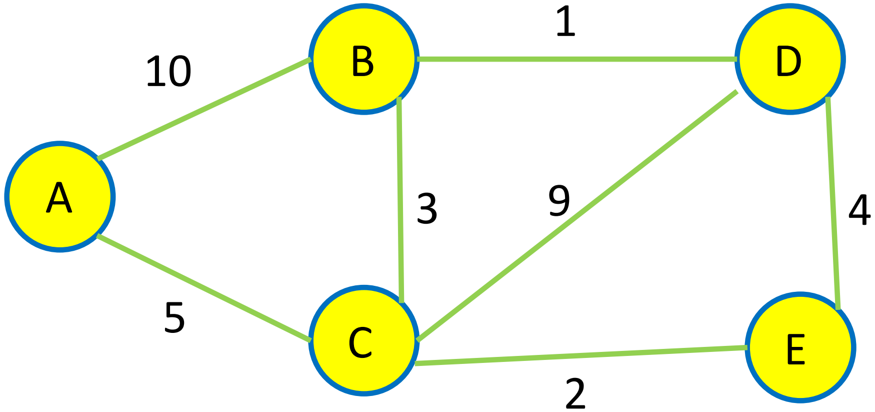
Kruskal's Algorithm

Kruskals(G(V, E))
    Sort the edges in ascending order of weights
    Let T be a graph with V as its vertices, and no edges
    For each edge (v, u) in ascending order
        If adding (v,u) does not create a cycle in T
            Add (v,u) to T
    Return T

Finalized:



How to determine if the edge will create a cycle???



Sorted Edges:

B→D,1	C→E,2	C→B,3	E→D,4	A→C,5	C→D,9	A→B,10

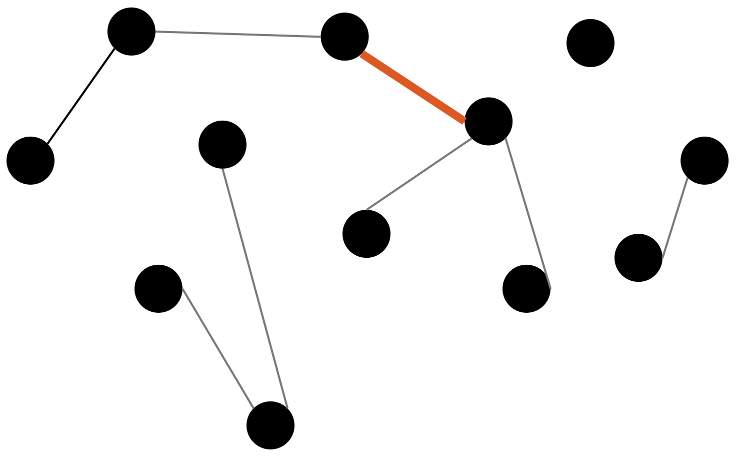
Finalized (in MST):

B→D	C→E	C→B	A→C

Kruskal's Algorithm

At some point in the algorithm, we want to add an edge

Can we add the red edge?

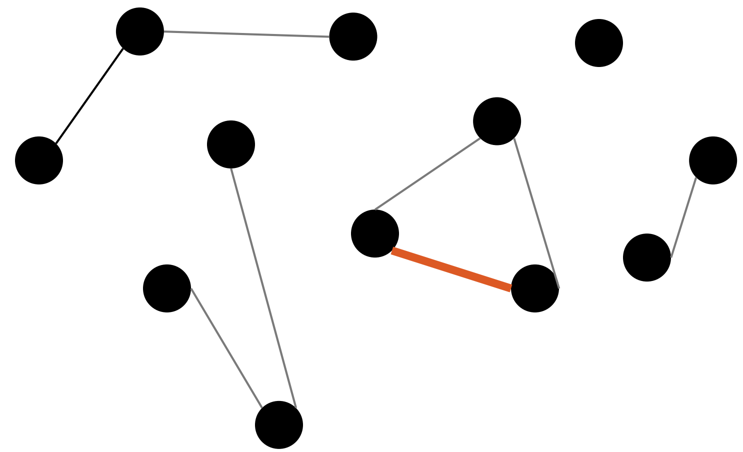


Kruskal’s Algorithm

At some point in the algorithm, we want to add an edge

Can we add the red edge?

Can we define what makes an edge “okay?

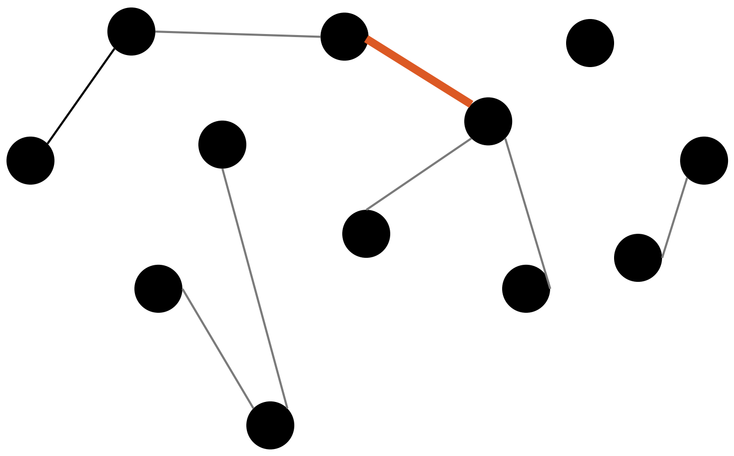


Quiz time!

https://flux.qa - RFIBMB

Kruskal's Algorithm

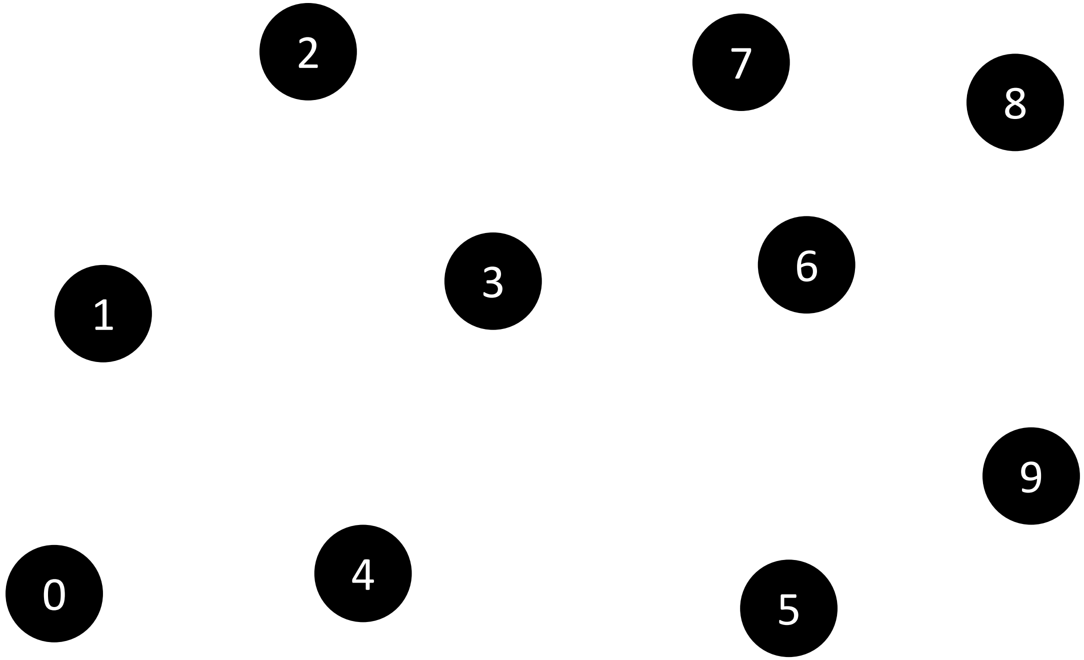
At some point in the algorithm, we want to add an edge
Can we add the red edge?
Can we define what makes an edge "okay?



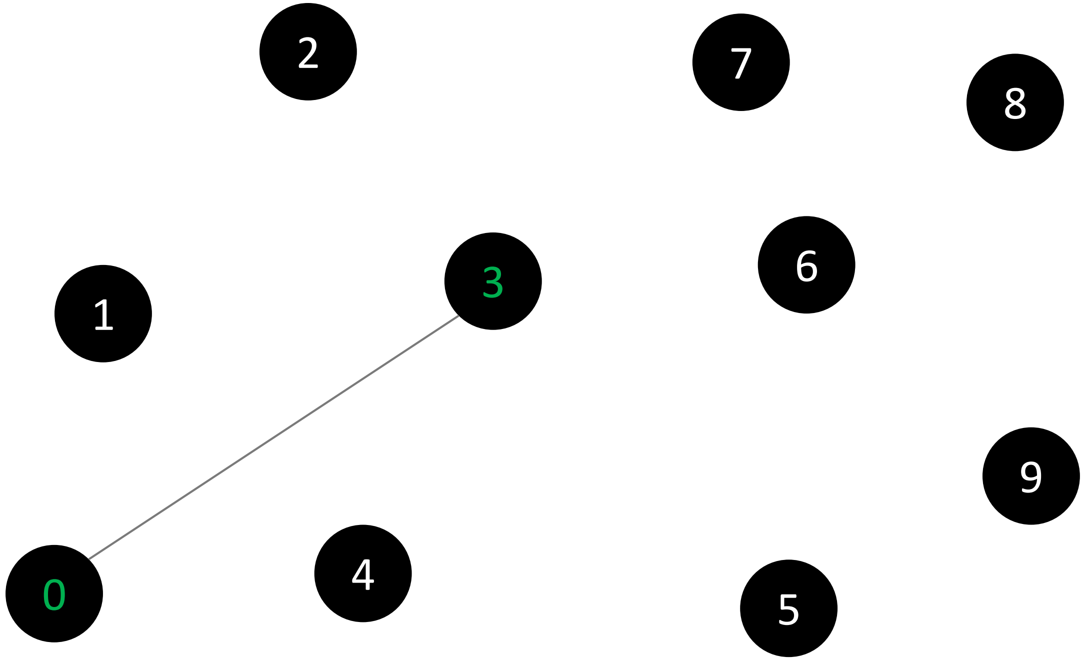
An edge can be added if it does not create a cycle
OR
An edge can be added if it is not between two vertices which belong to the same component

Importantly, adding an edge between two components "merges" those components into one new component.

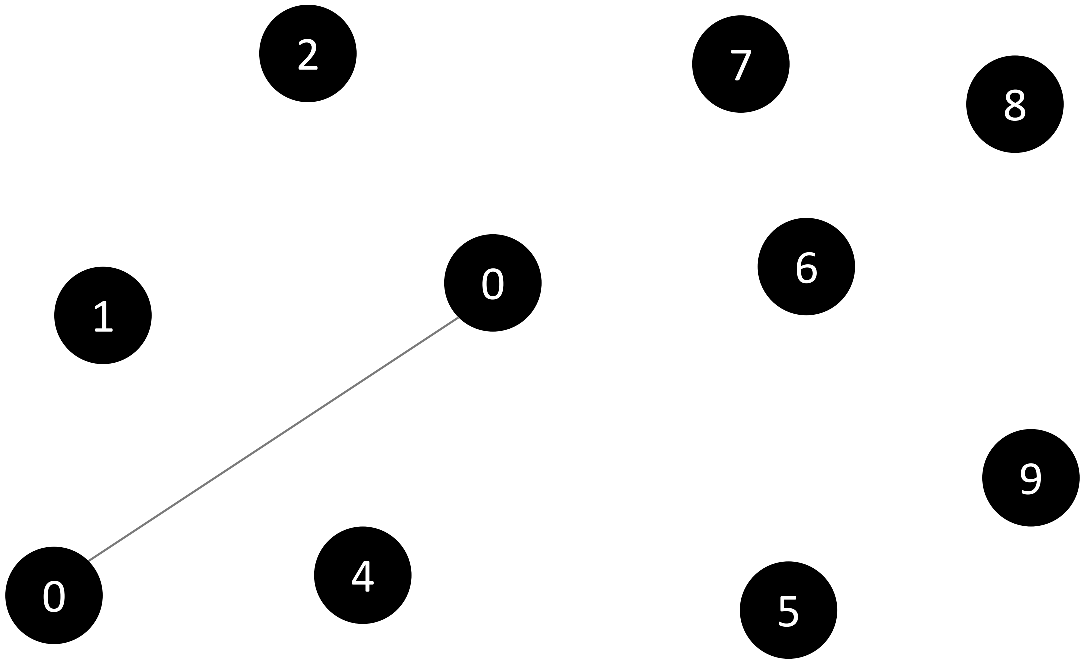
Kruskal’s Algorithm



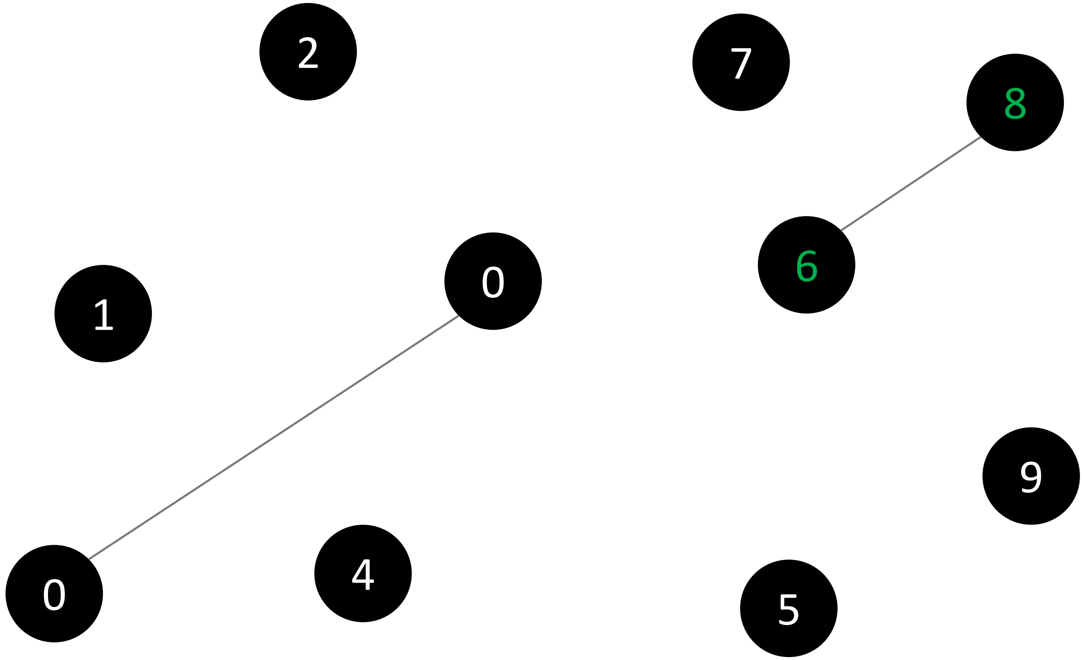
Kruskal's Algorithm



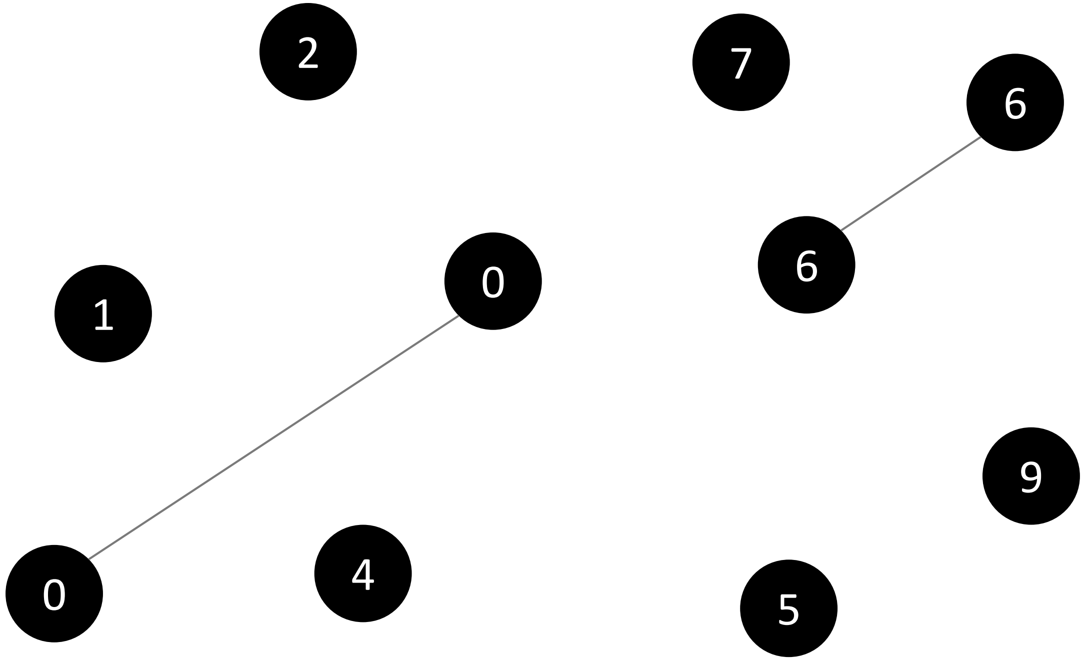
Kruskal's Algorithm



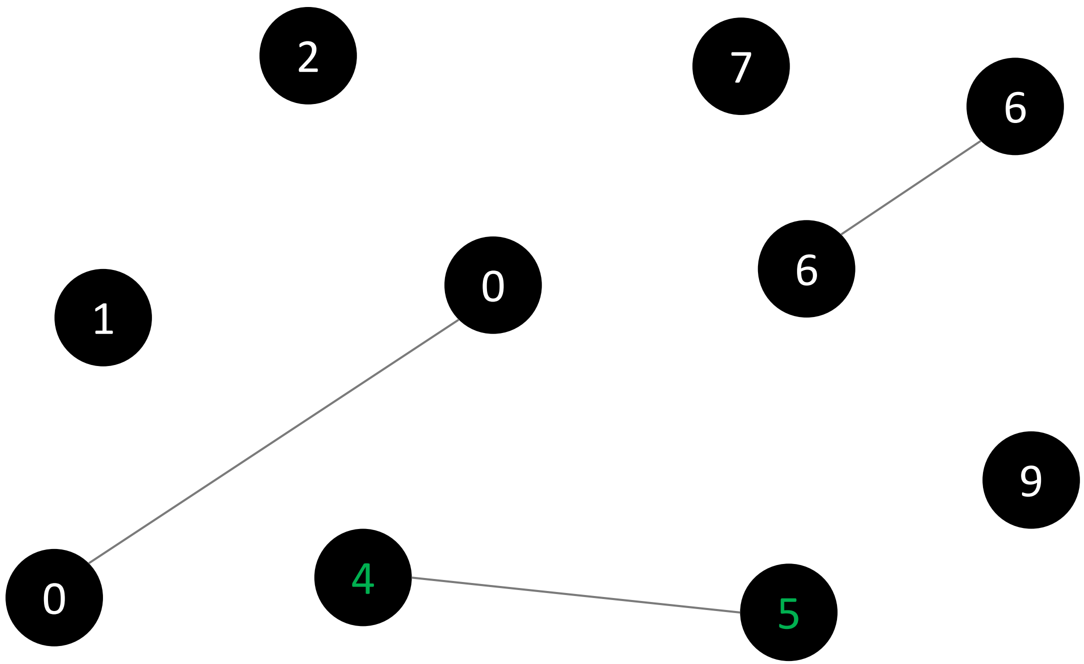
Kruskal's Algorithm



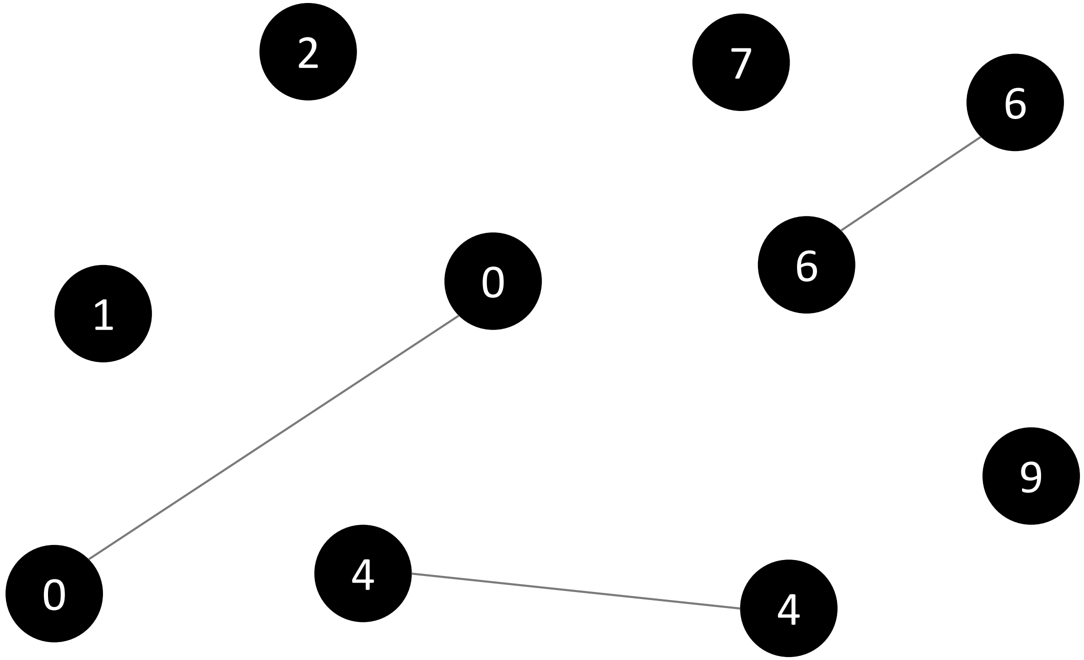
Kruskal’s Algorithm



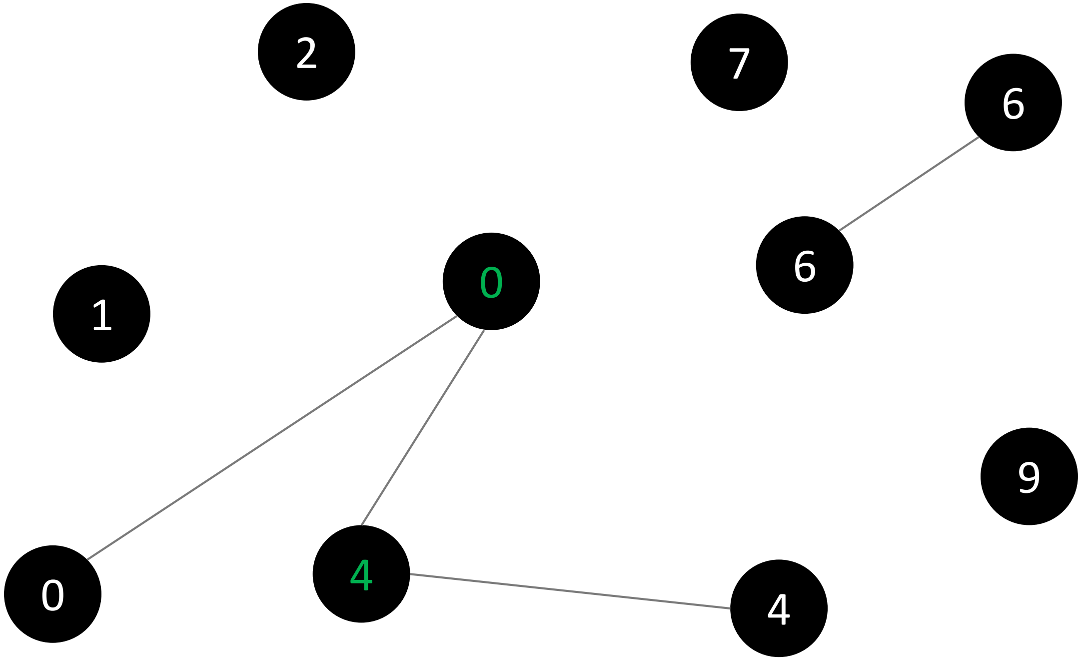
Kruskal's Algorithm



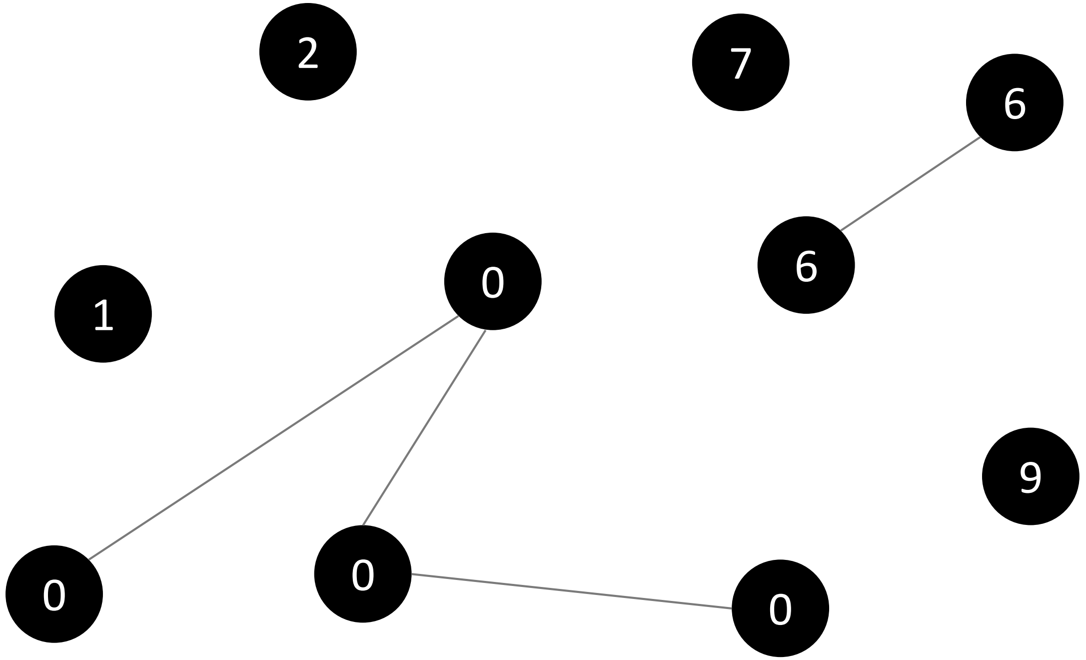
Kruskal’s Algorithm



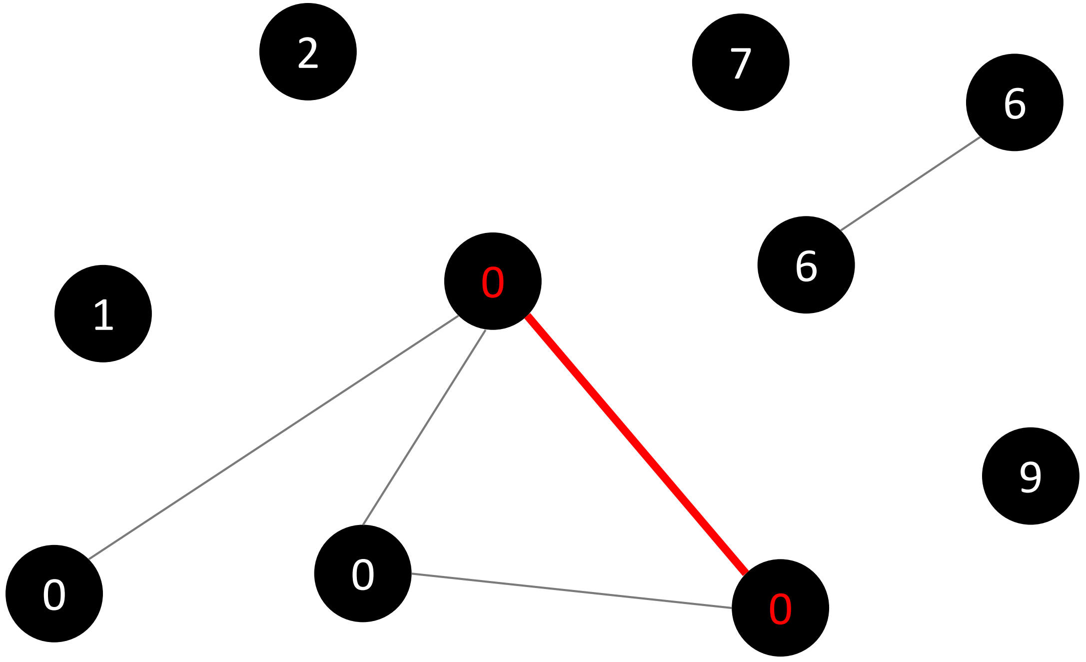
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal’s Algorithm



Kruskal's Algorithm

```
Kruskals(G(V, E))
    Sort the edges in ascending order of weights
    Let each vertex in V be given a unique ID
    Let T be a graph with V as its vertices, and no edges

    For each edge (v, u) in ascending order
        #If adding (v,u) does not create a cycle in T
        If set_lookup(v) != set_lookup(u)
            Add (v,u) to T
            union(set of u, set of v)
    Return T
```

How can we quickly look up the set of a vertex, and also union two sets?

Union-Find Data Structure

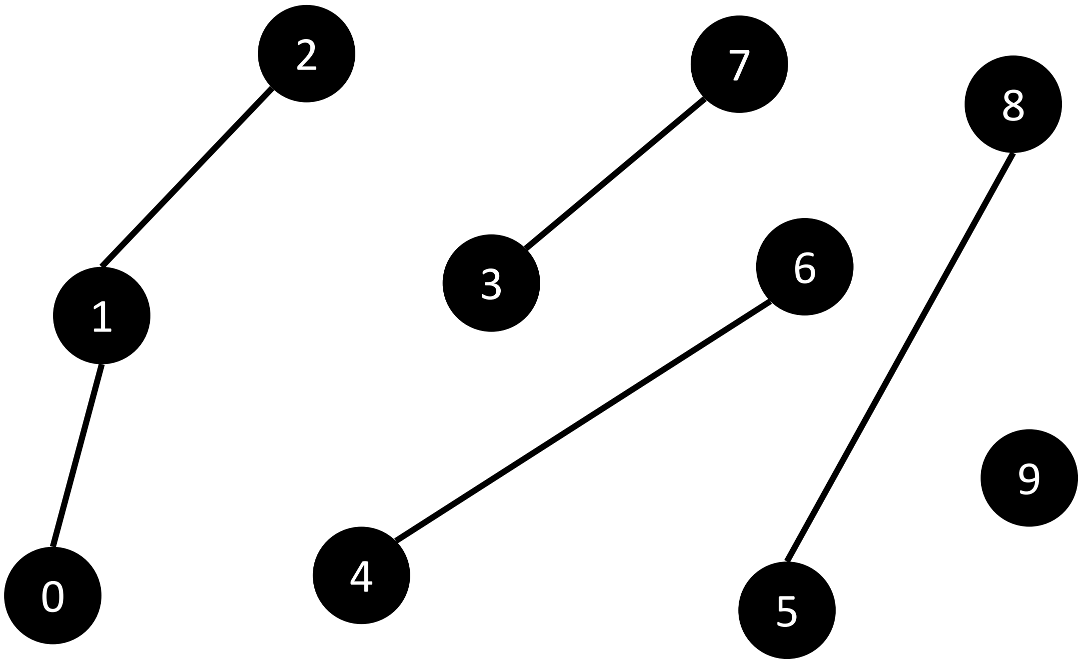
- Define two operations: **find(u)** and **union(u,v)**
- **Find(u):** Given a vertex u, return its set ID
- **Union(u,v):** Given two vertices u and v, if they have different set IDs, union the two sets they belong to (and update all the set IDs of the vertices in one of the sets)

Union-Find Data Structure

- We need both **find(u)** and **union(u,v)** to be fast
- If we just store the set ID of each vertex in an array, then find is O(1)
- Union(u, v) requires us to loop through the whole array, looking for elements of find(u) and changing them to find(v), which is O(V)

Vertex ID	0	1	2	3	4	5	6	7	8	9
Set ID	0	0	0	7	6	5	6	7	5	9

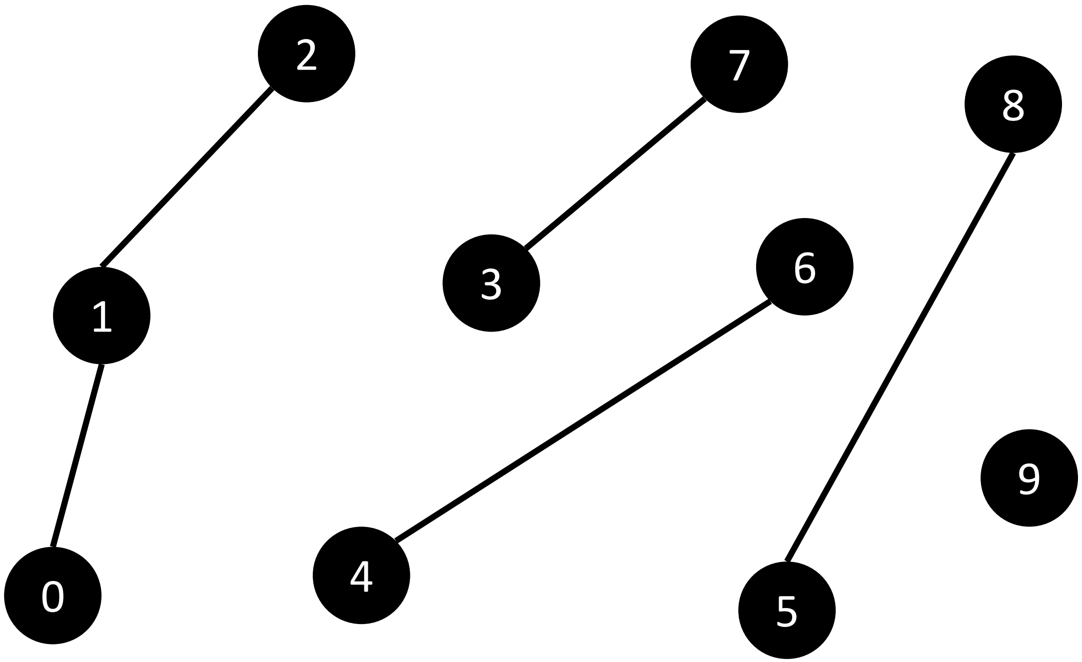
Union-Find Data Structure



Vertex ID	0	1	2	3	4	5	6	7	8	9
Set ID	0	0	0	7	6	5	6	7	5	9

Union-Find Data Structure

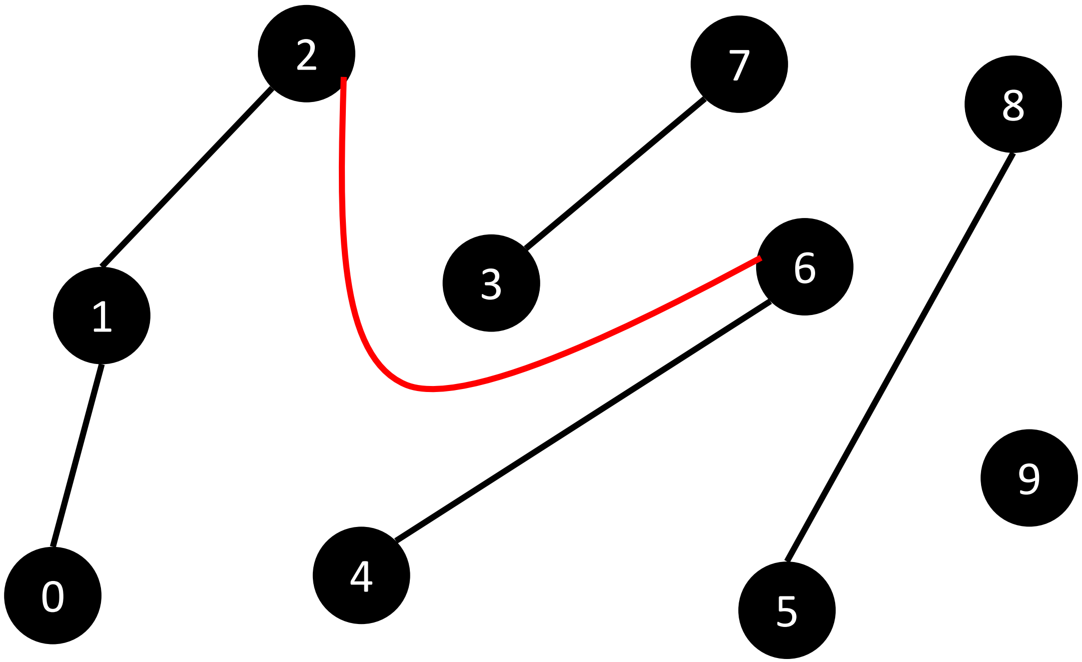
Union(2, 6)



Vertex ID	0	1	2	3	4	5	6	7	8	9
Set ID	0	0	0	7	6	5	6	7	5	9

Union-Find Data Structure

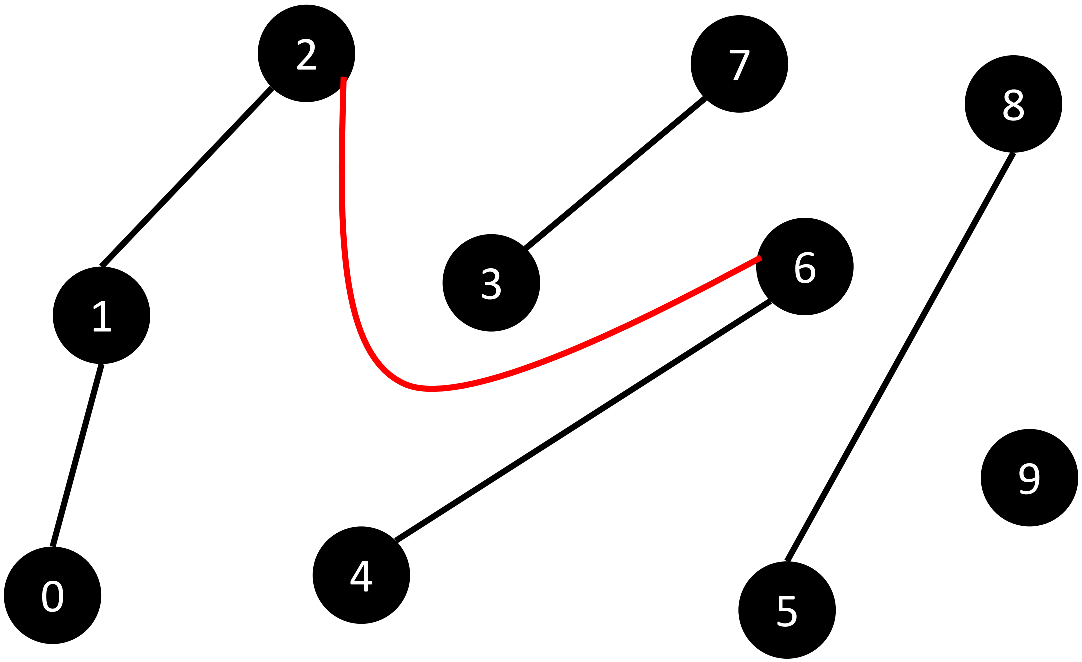
Union(2, 6)



Vertex ID	0	1	2	3	4	5	6	7	8	9
Set ID	0	0	0	7	6	5	6	7	5	9

Union-Find Data Structure

Union(2, 6)

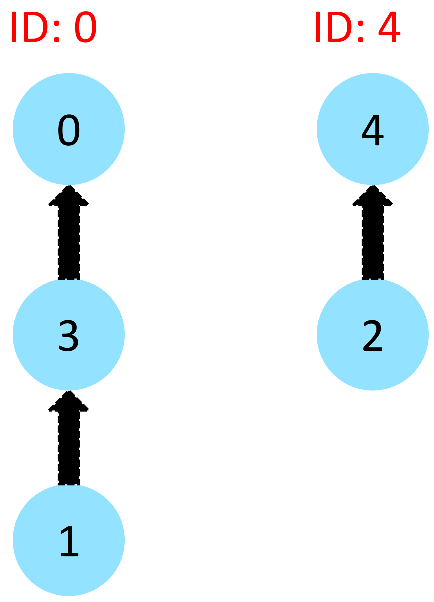


Vertex ID	0	1	2	3	4	5	6	7	8	9
Set ID	0	0	0	7	6	5	6	7	5	9

Need to find all 6 and change to 0… O(V)

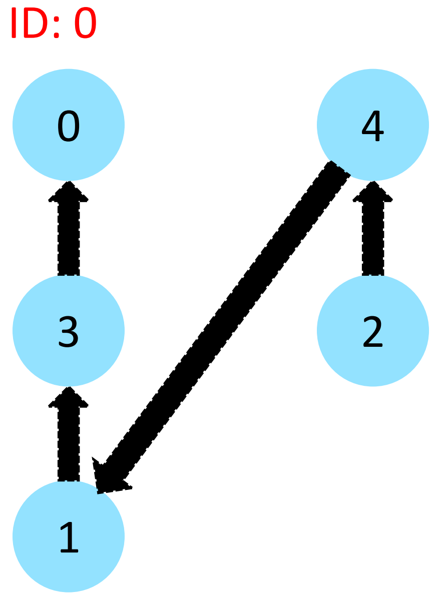
Union-Find Data Structure

- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)



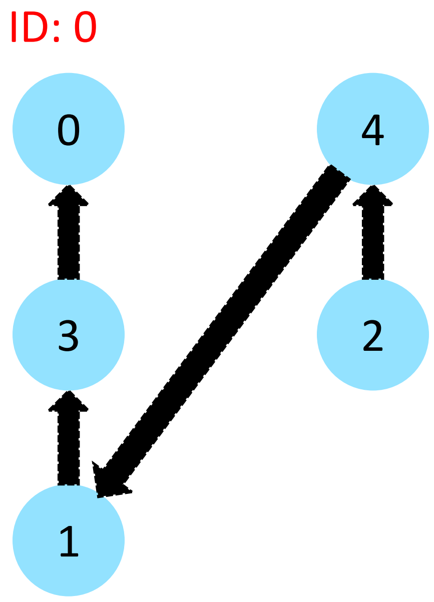
Union-Find Data Structure

- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)



Union-Find Data Structure

- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

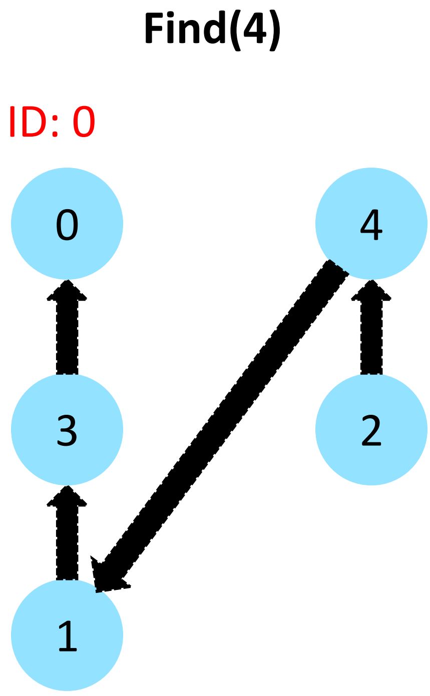


Quiz time!
https://flux.qa - RFIBMB

Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

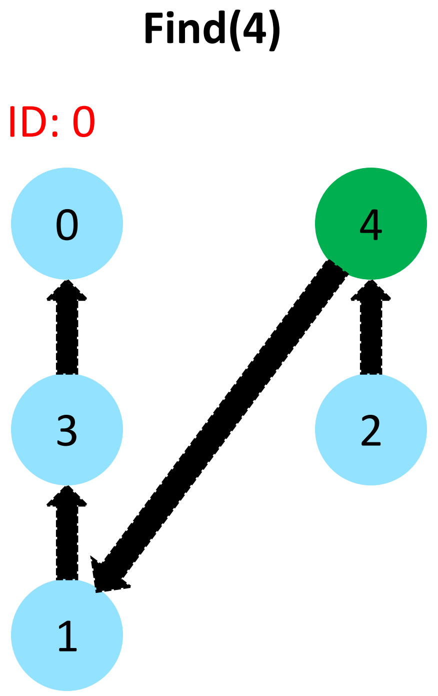
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

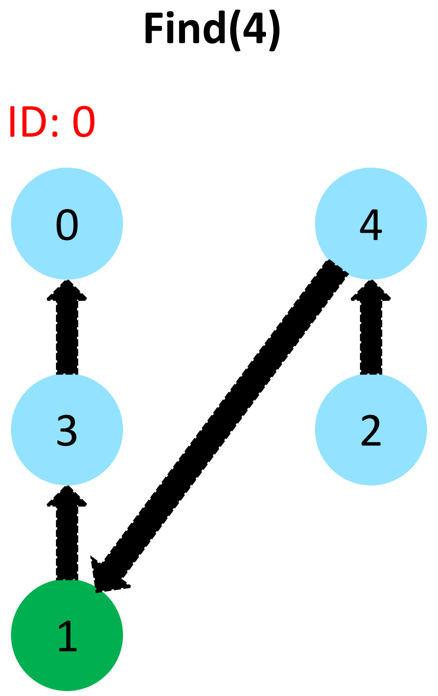
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

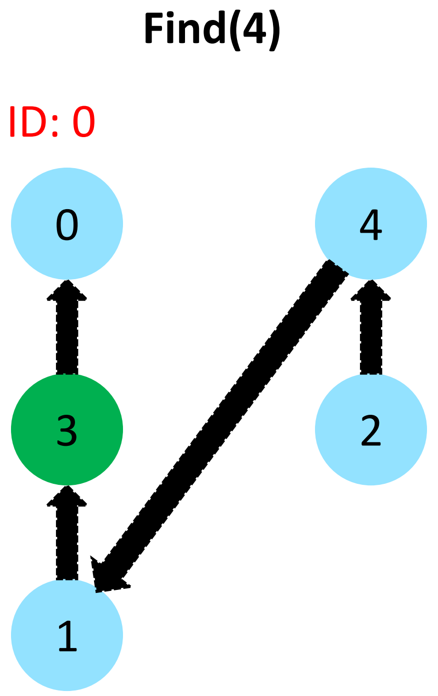
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



Union-Find Data Structure

- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

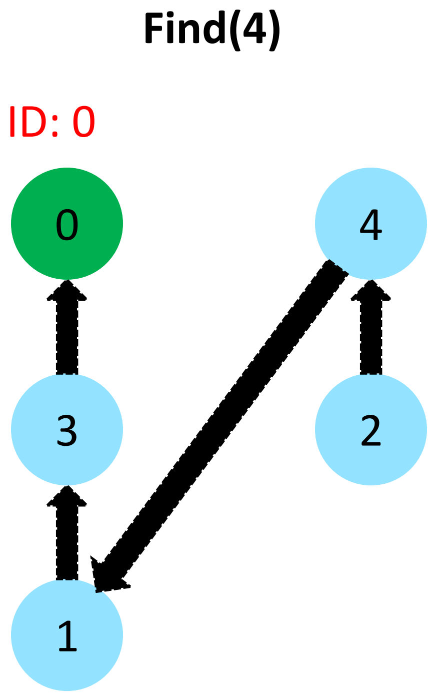
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



Union-Find Data Structure

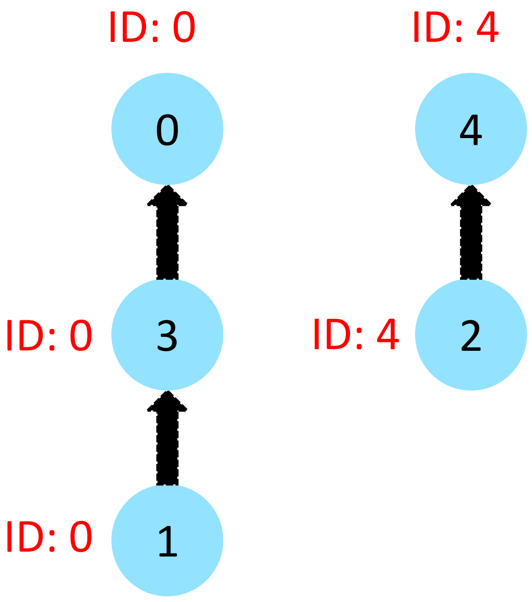
- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



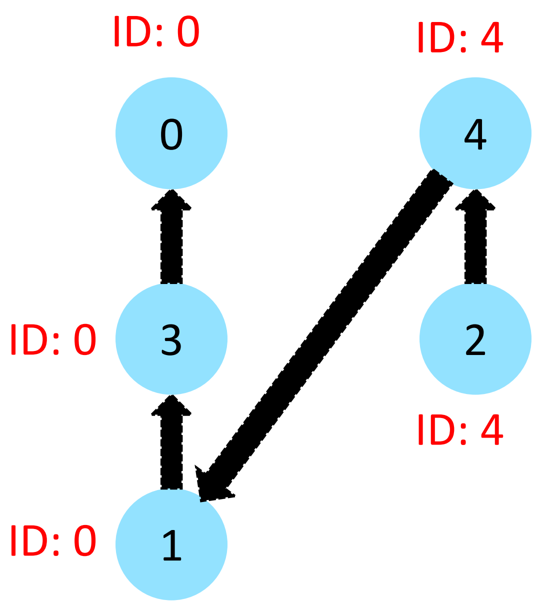
Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is O(1)
- But Union is now slower, we have to update all the IDs



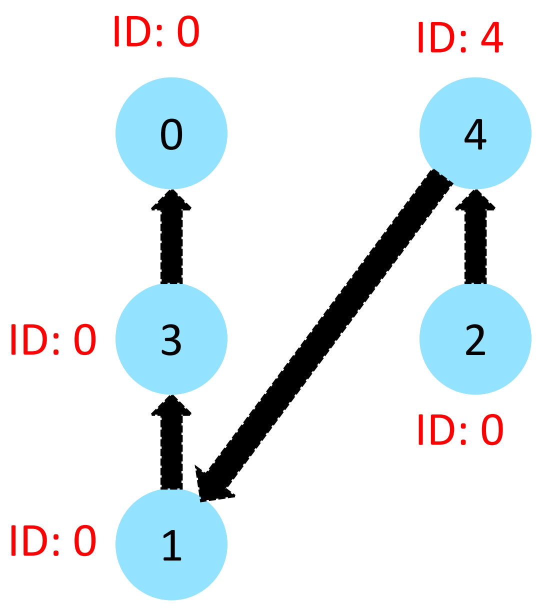
Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is O(1)
- But Union is now slower, we have to update all the IDs



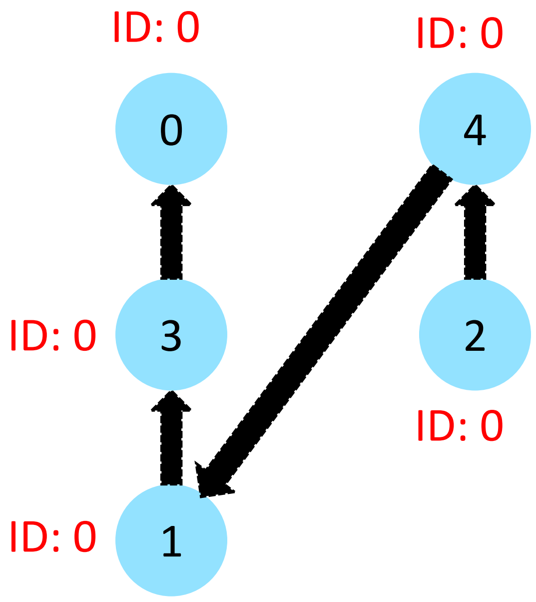
Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is O(1)
- But Union is now slower, we have to update all the IDs



Union-Find Data Structure

- Alternatively, every node could know its ID
- Now find is O(1)
- But Union is now slower, we have to update all the IDs



Union-Find Data Structure

- Where are we?
- We want find and union to be fast
- Linked lists are an improvement, since they stop us looking at items which are not relevant to the union we are currently doing
- Linked lists allows O(1) union
- We can’t make find O(1) because to do that, we have to store the ID at every node which makes union slow (we have to change all the IDs)

- Solution: Change from linked list to linked tree

Union-Find Data Structure

Operations:

Vertex ID	0	1	2	3	4	5
Parent	0	1	2	3	4	5

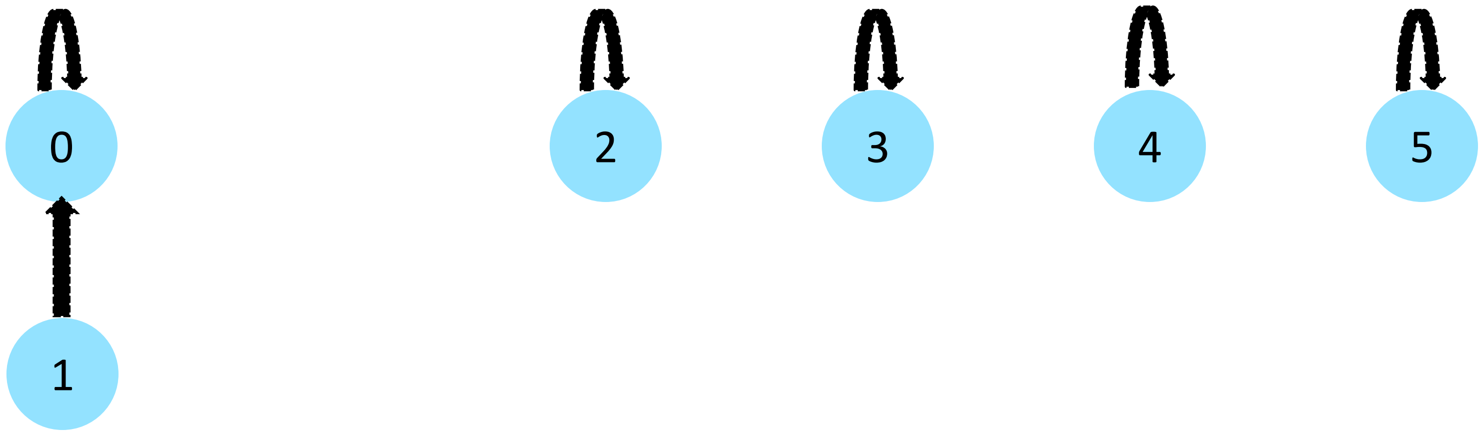
Union-Find Data Structure

Operations:
Union(0,1)
  Find(0) = 0
  Find(1) = 1

Vertex ID	0	1	2	3	4	5
Parent	0	1	2	3	4	5

Union-Find Data Structure

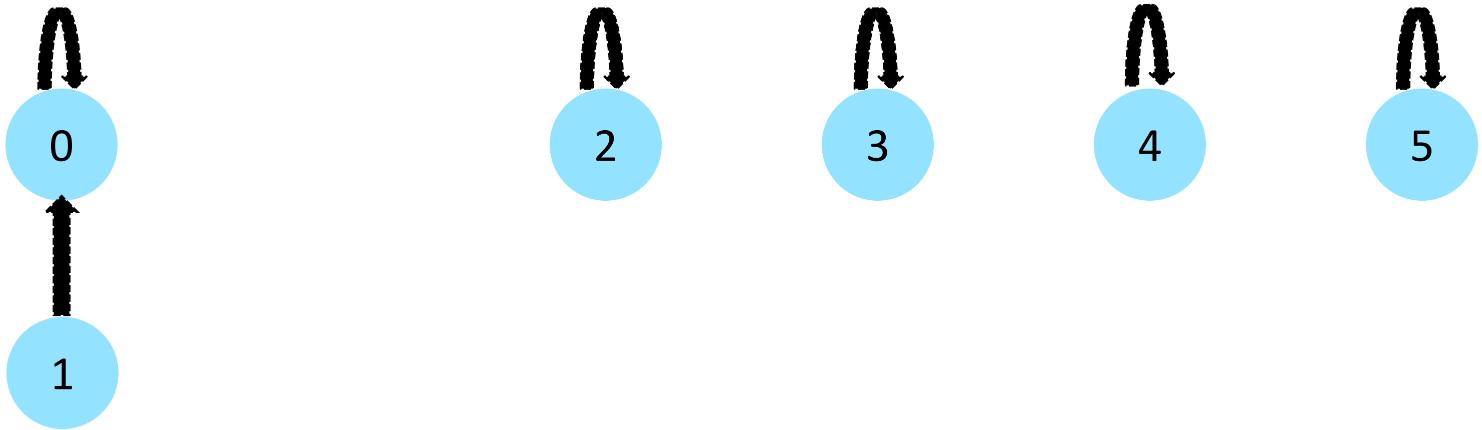
Operations:
Union(0,1)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	3	4	5

Union-Find Data Structure

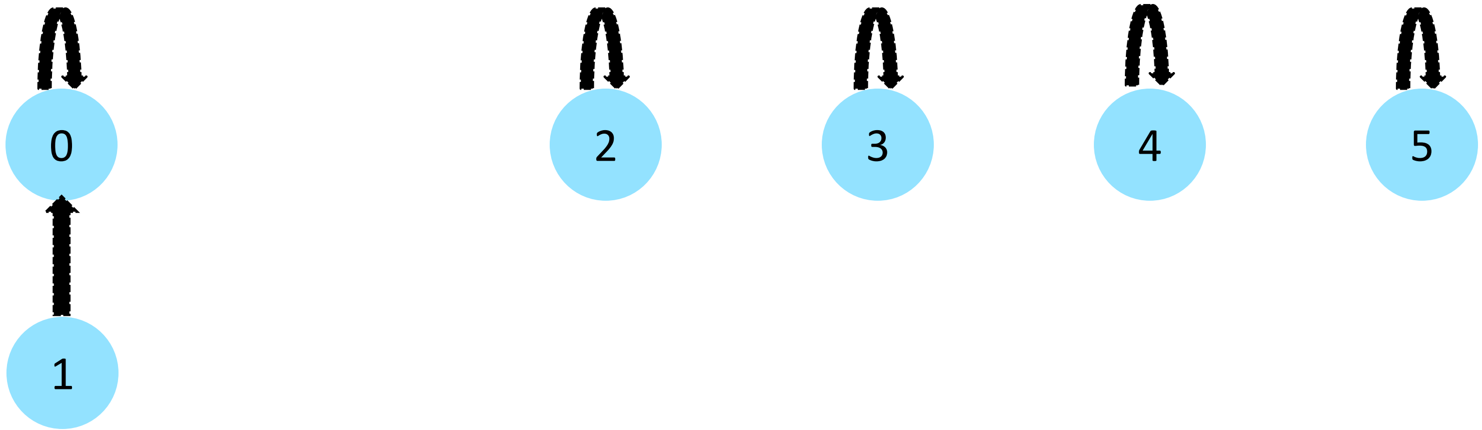
Operations:
Union(0,1)
Union(2,3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	3	4	5

Union-Find Data Structure

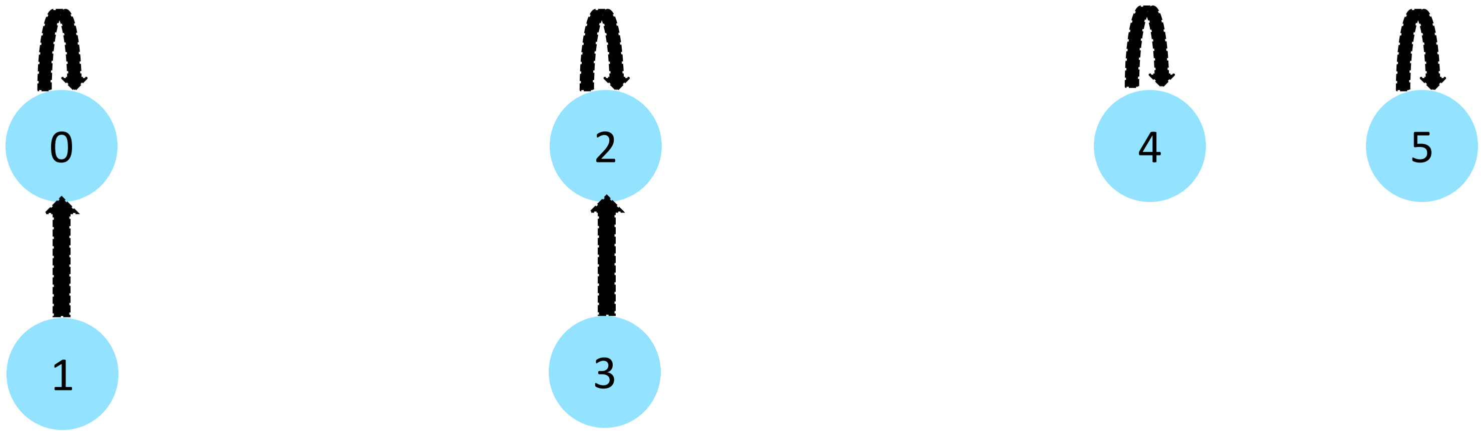
Operations:
Union(0,1)
Union(2,3)
 Find(2) = 2
 Find(3) = 3



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	3	4	5

Union-Find Data Structure

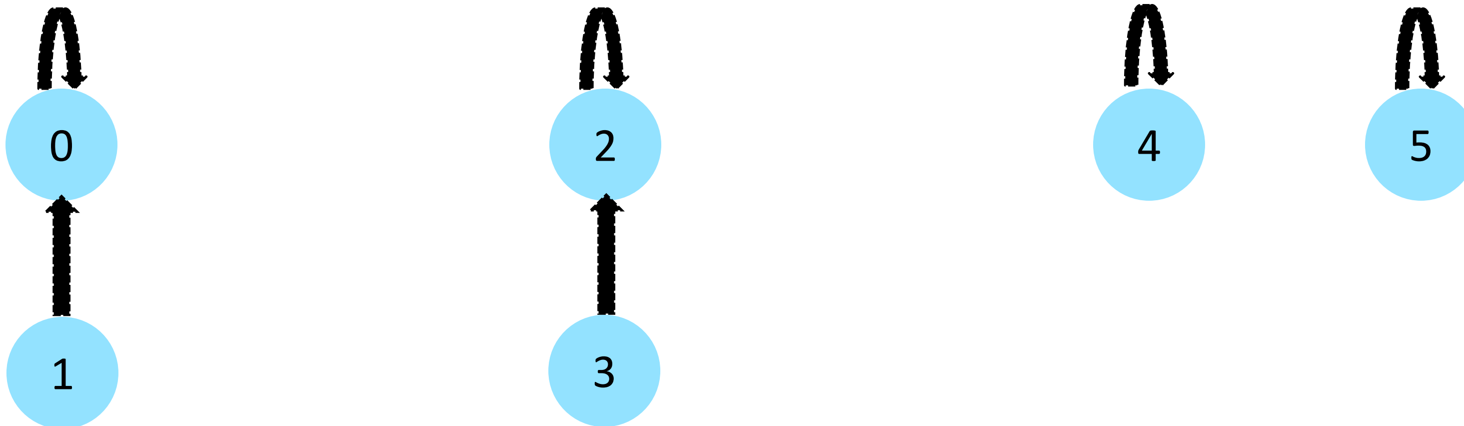
Operations:
Union(0,1)
Union(2,3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

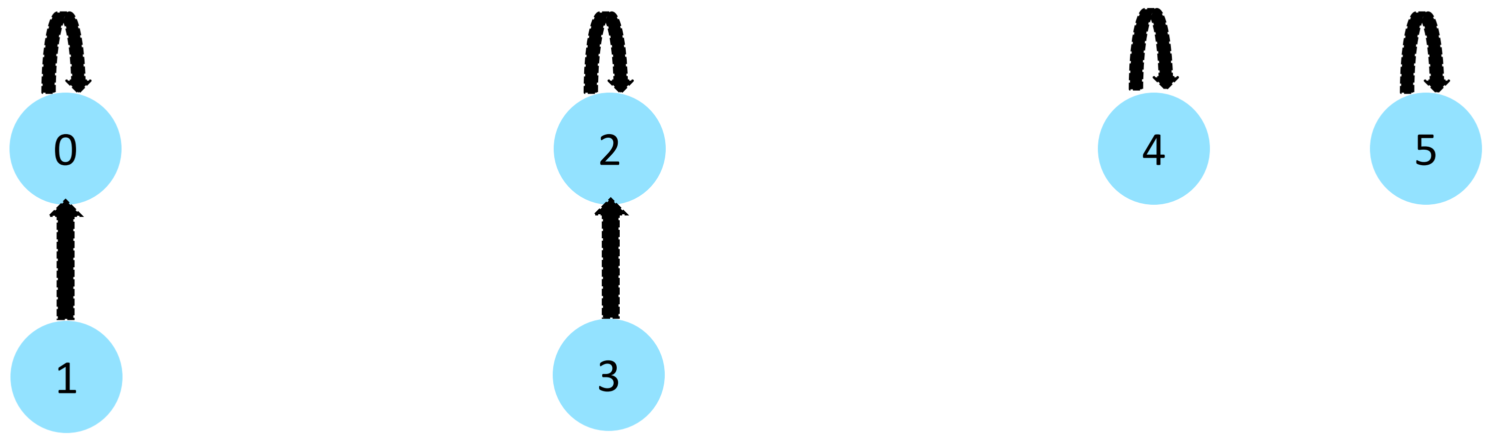
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

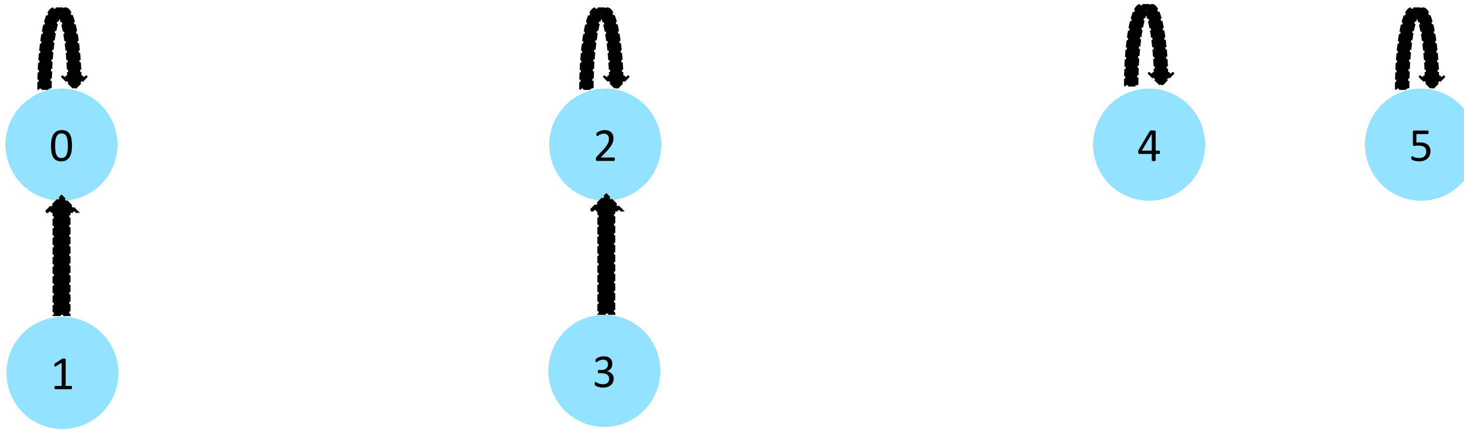
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

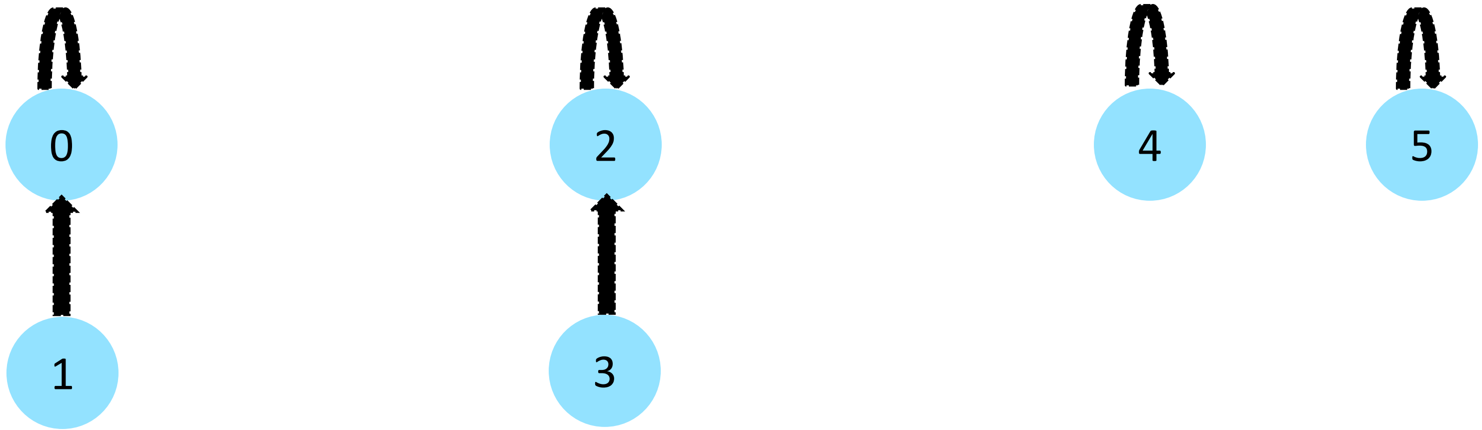
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

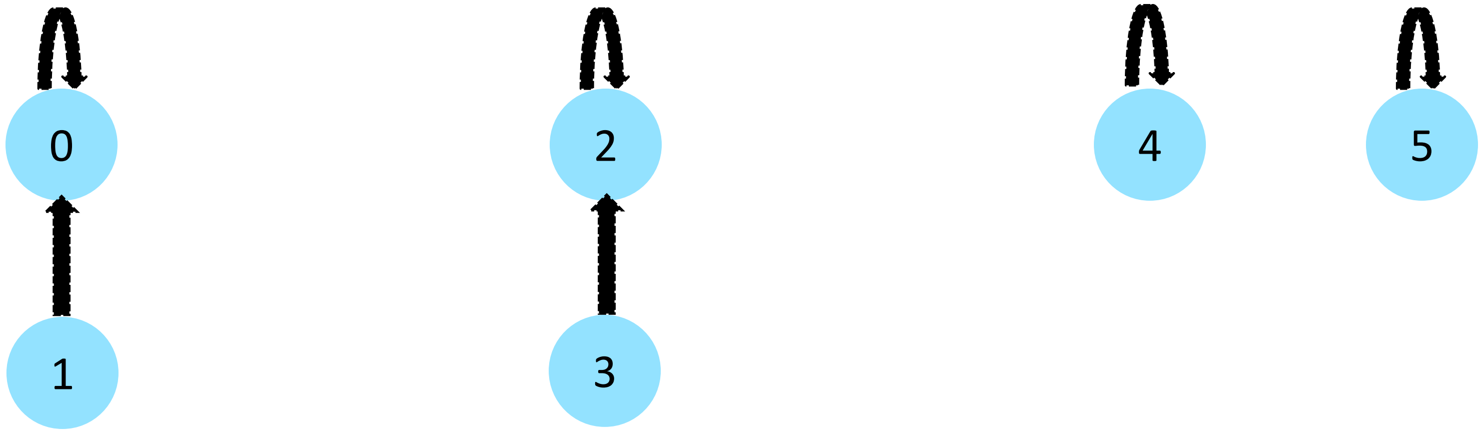
Operations:
Union(0,1)
Union(2,3)
Find(3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

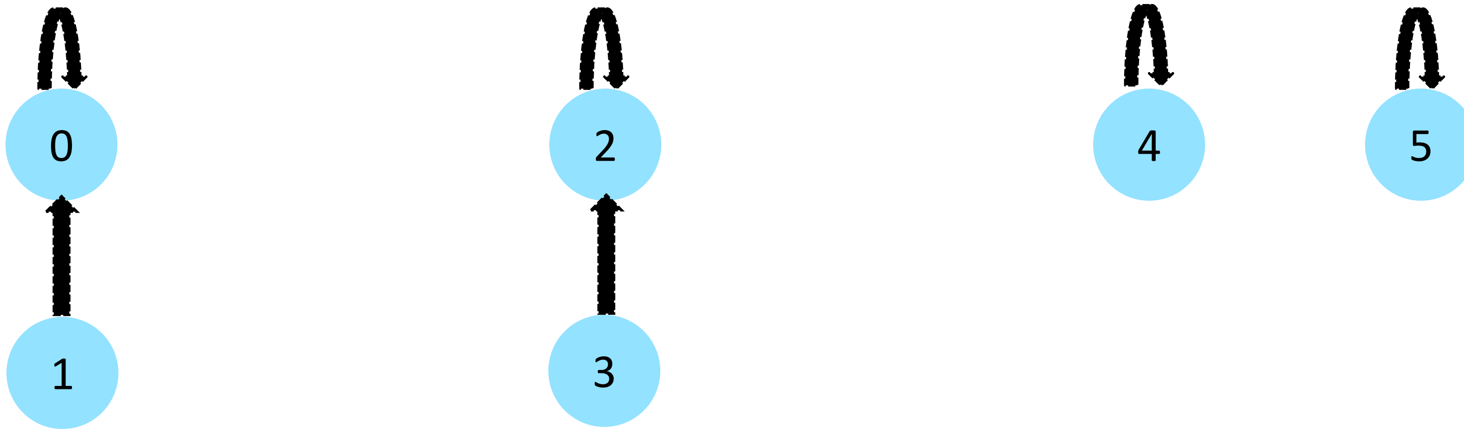
Operations:
Union(0,1)
Union(2,3)
Find(3)=2



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

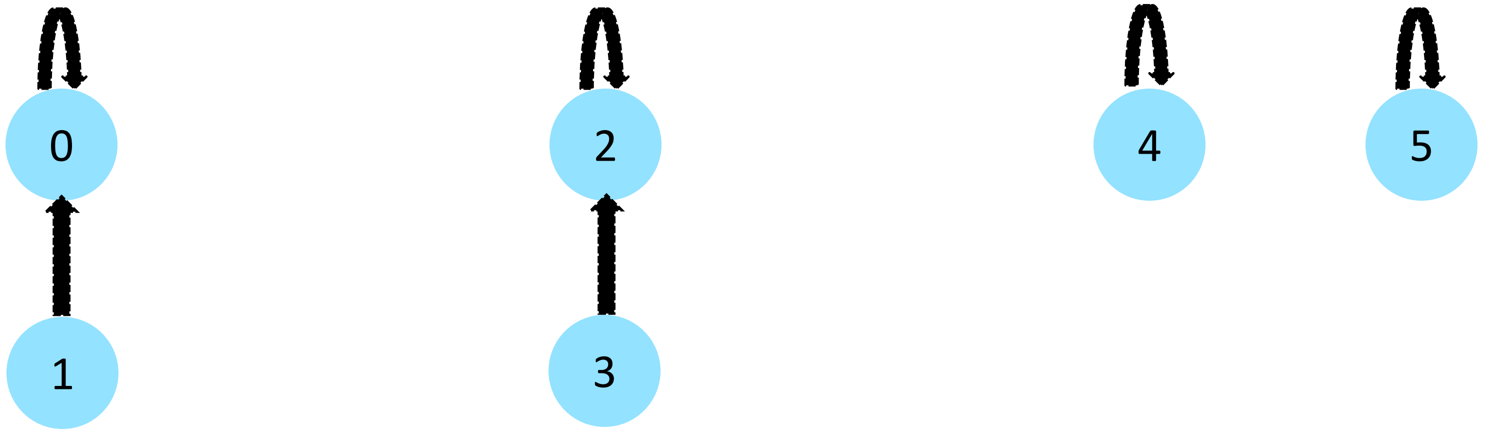
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

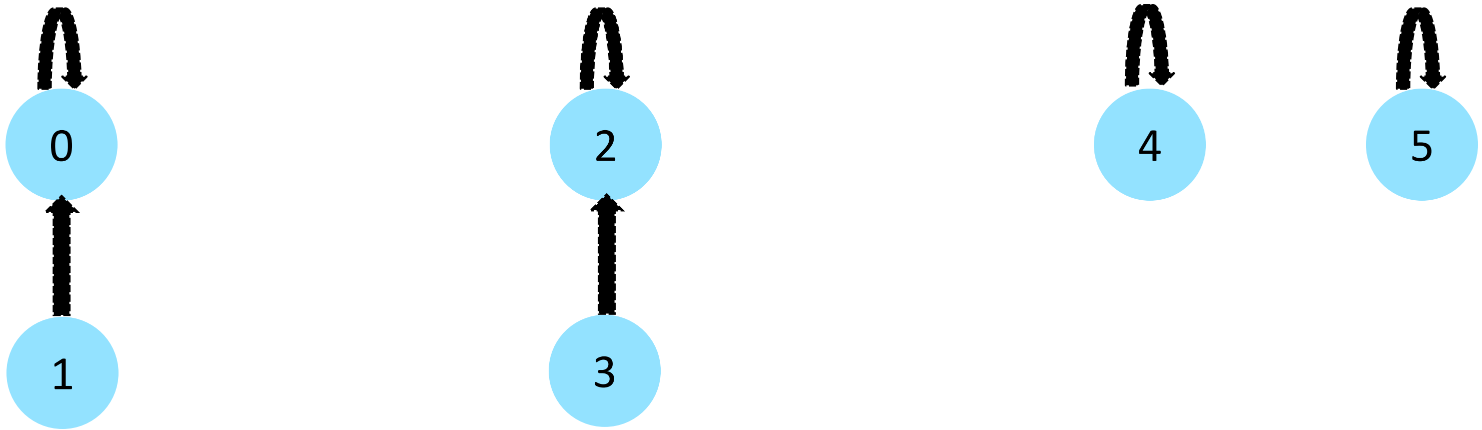
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
  Find(0) = 0
  Find(1) = 0



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

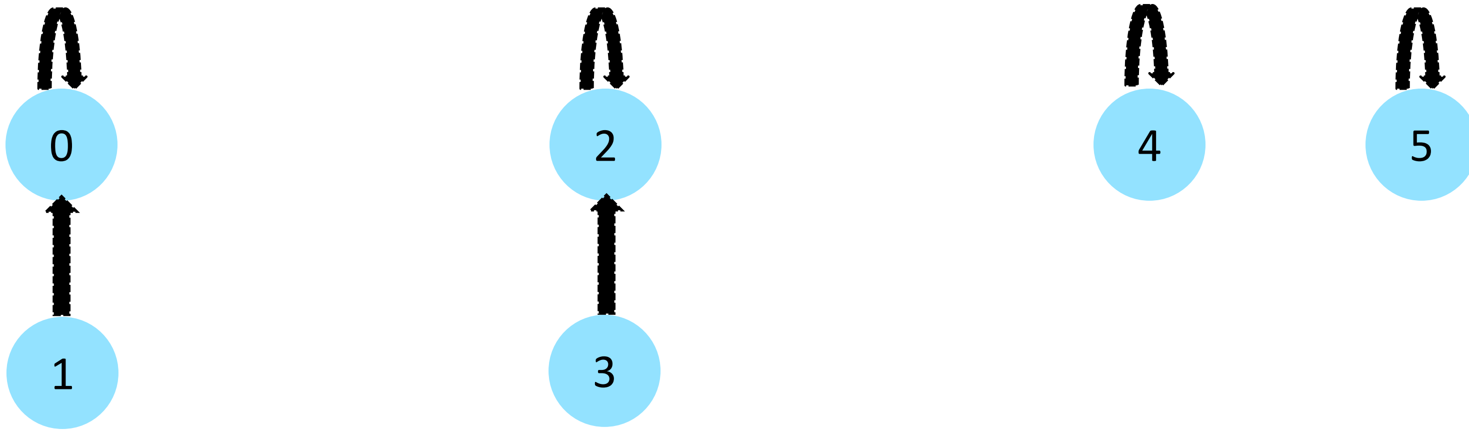
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
 Find(1) = 0
 Find(3) = 2



Vertex ID	0	1	2	3	4	5
Parent	0	0	2	2	4	5

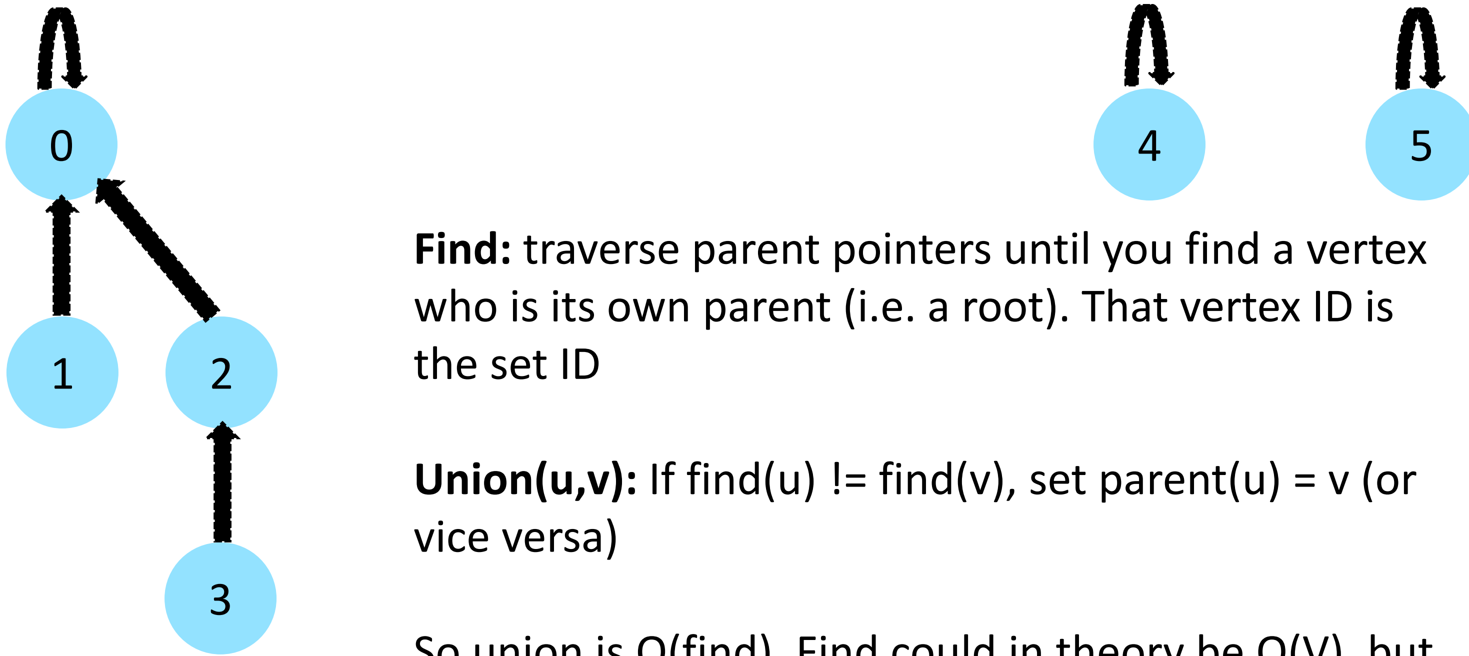
Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2

Vertex ID	0	1	2	3	4	5
Parent	0	0	0	2	4	5

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2



Find: traverse parent pointers until you find a vertex who is its own parent (i.e. a root). That vertex ID is the set ID

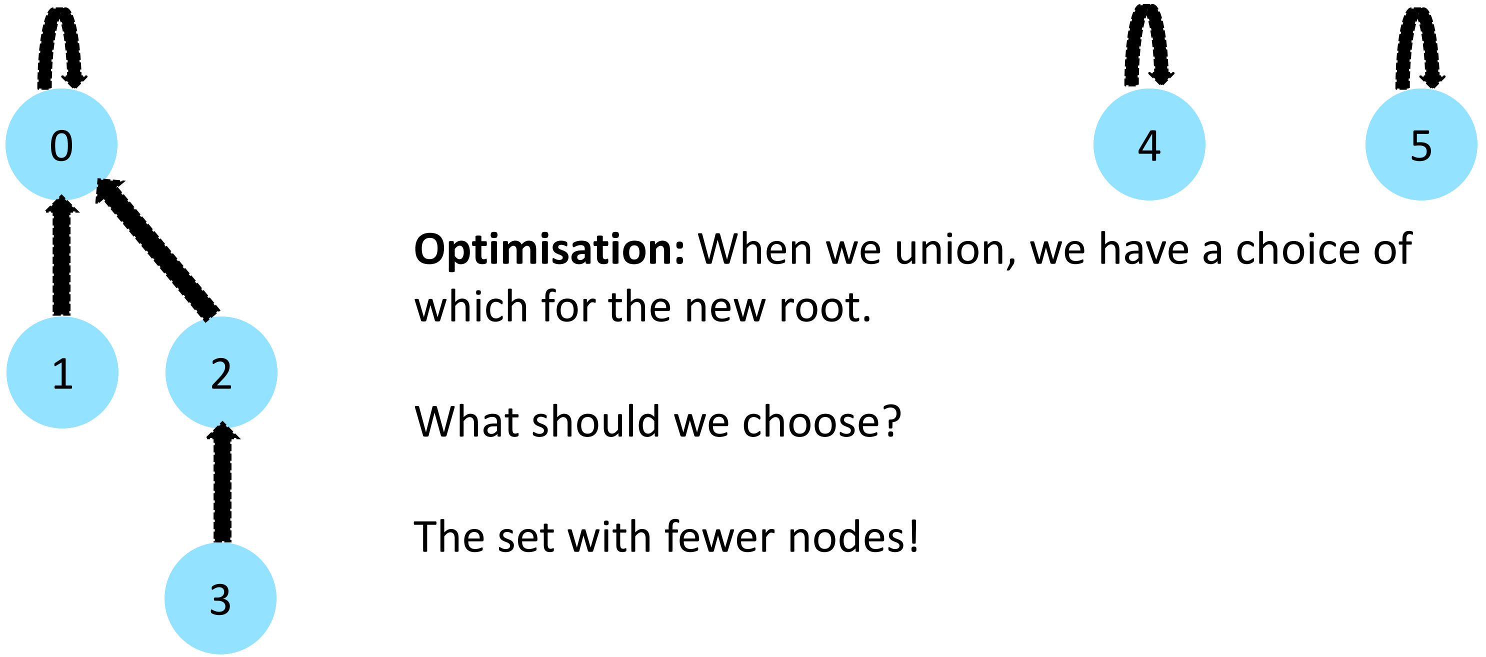
Union(u,v): If find(u) != find(v), set parent(u) = v (or vice versa)

So union is O(find). Find could in theory be O(V), but if we can keep the heights of the trees low, then it will be at most O(max height)

Vertex ID	0	1	2	3	4	5
Parent	0	0	0	2	4	5

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2



Optimisation: When we union, we have a choice of which for the new root.

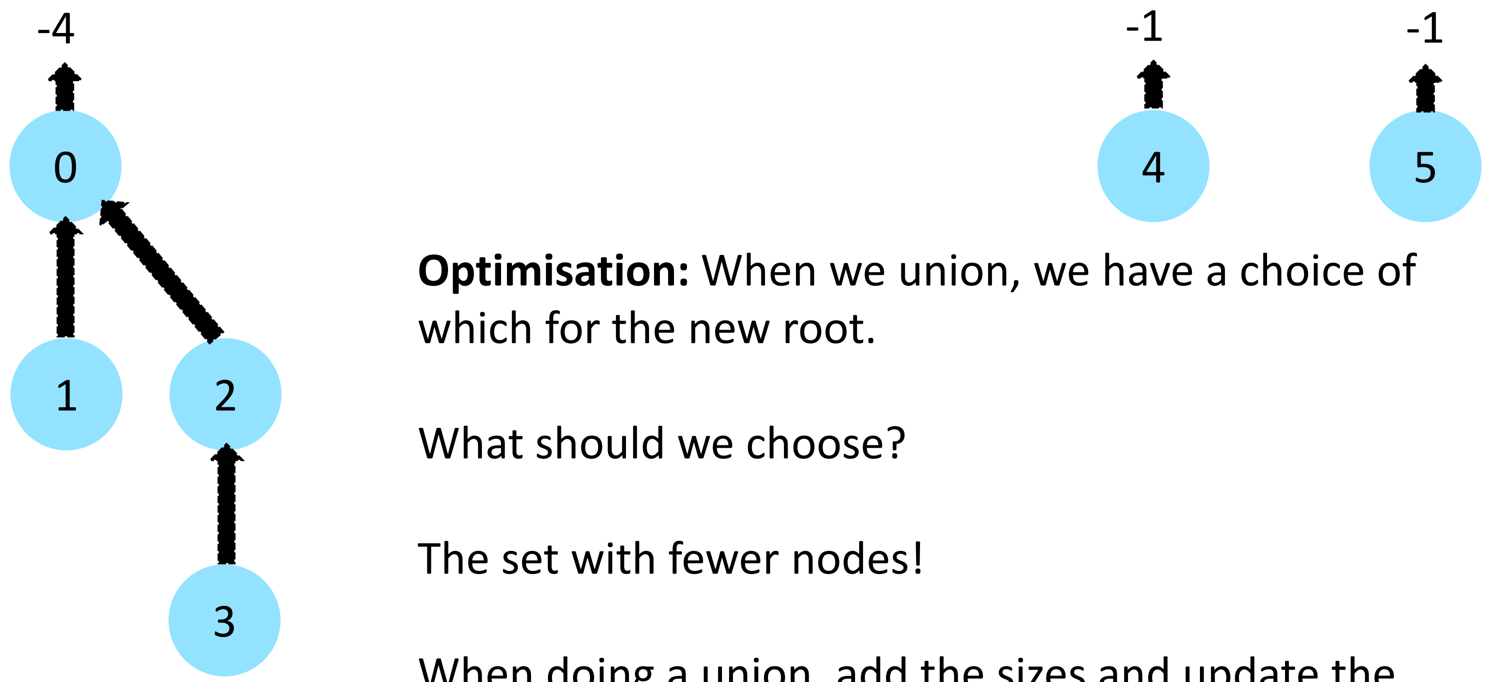
What should we choose?

The set with fewer nodes!

Vertex ID	0	1	2	3	4	5
Parent	0	0	0	2	4	5

Union-Find Data Structure

Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)
  Find(1) = 0
  Find(3) = 2



Optimisation: When we union, we have a choice of which for the new root.

What should we choose?

The set with fewer nodes!

When doing a union, add the sizes and update the parent value of the root appropriately

Vertex ID	0	1	2	3	4	5
Parent	-4	0	0	2	-1	-1

Parent array values are now either parents OR –ve sizes

Kruskal's Algorithm: Complexity

Algorithm 70 Kruskal's algorithm

```
function KRUSKAL(G = (V, E))
    sort(E, key((u, v)) = w(u, v))            // Sort edges in ascending order of weight
    forest = UnionFind.initialise(n)
    T = (V, ∅)
    for each edge (u, v) in E do
        if forest.FIND(u) ≠ forest.FIND(v) then      // Ignore edges that would create a cycle
            forest.UNION(u, v)
            T.add_edge(u, v)
    return T
```

But this is not tight as we assumed the cost of UNION_SETS to be O(V) for each call leading to overall cost of O(EV). A closer look reveals that the total cost of UNION_SETS is O(V log V)

Time Complexity:

- Initialization: O(V)
- Sorting edges: O(E log E)
  - E log E <= E log V^2 = 2 E log V → O(E log V)
- For loop executes O(E) times
  - SET_ID() takes O(x) where x is height of the tree
  - UNION_SET() takes O(1) + 2 finds, so it takes O(x) where x is the height of the **deeper** of the two trees to be unioned (which could be at most V)
- Total cost: O(EV)

Complexity of UNION_SETS

- We can show that, when using the union-by-size rule, the number of elements N in any tree is at least 2^h, where h is the height of the tree
- In other words, $h \leq \log_2 N$

- Unioning takes 2 finds + O(1) effort
- Find takes effort equal to the height of the tree, which is $\leq \log_2 N$
- So union is $\mathrm{O}(\log_2 N)$, where N is the height of the taller tree being unioned.

- We need to do V-1 unions
- Each one has a worst case cost of O(log(V)) (this is a significant over-estimation, but it makes the maths easy)
- So the total cost of all unions is bounded by O(VlogV)

Kruskal's Algorithm: Complexity

Algorithm 70 Kruskal's algorithm

function KRUSKAL($G = (V, E)$)
  sort(E, key((u, v)) $= w(u, v)$) *// Sort edges in ascending order of weight*
  forest = UnionFind.initialise(n)
  $T = (V, \emptyset)$
  for each edge (u, v) **in** E **do**
    if *forest*.FIND(u) $\neq$ *forest*.FIND(v) **then** *// Ignore edges that would create a cycle*
      forest.UNION(u, v)
      T.add_edge(u, v)
  return T

Time Complexity:

- Sorting edges: O(E log E)
- Initialization of union find: O(V)
  - E log E = E log V^2 = 2 E log V → O(E log V)
- For loop executes O(E) times
  - The two finds take the same effort as the union, log(v)
- UNION takes O(V log V) in total
- Total cost: O(E log V + V log V) → O(E log V)

Complexity of UNION_SETS

- We can improve the disjoint sets data structure significantly with 2 other optimisations
- Union by rank
- Path compression

- These are discussed in the notes, but are not examinable

- The complexity can be improved from VlogV to $V\alpha(V)$, where α denotes the inverse Ackermann function, an **extremely** slow growing function

- Note: α(any number which can be represented using the matter in the universe) < 5, so $V\alpha(V)$ is effectively $O(V)$

Kruskal's Algorithm: Correctness

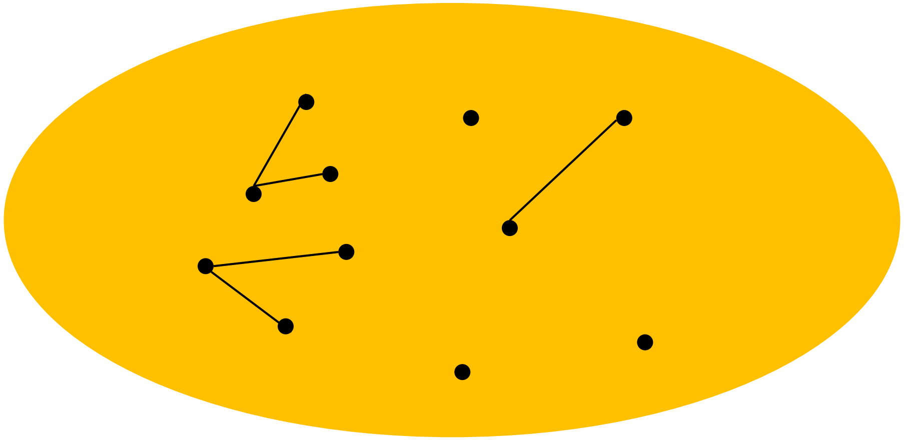
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**



Kruskal's Algorithm: Correctness

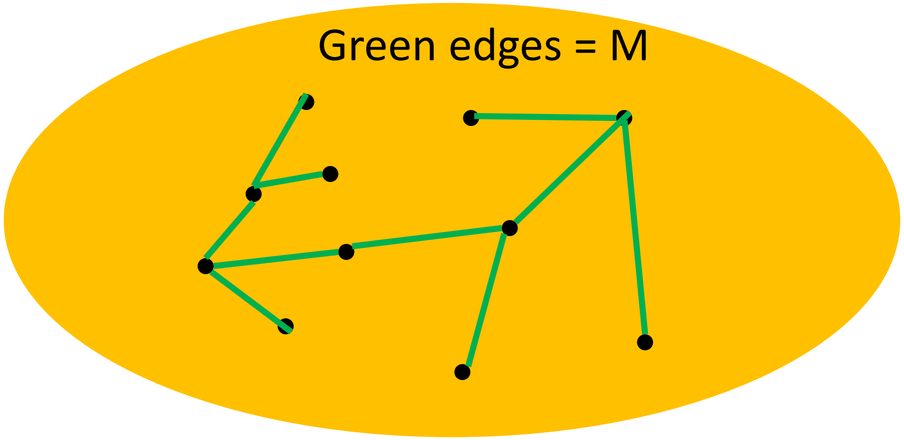
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**



Kruskal's Algorithm: Correctness

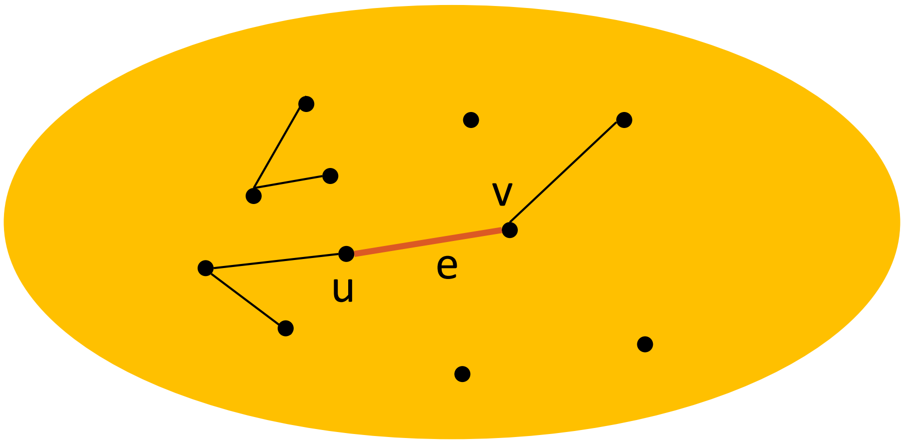
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects two vertices in different components of **T** (i.e this is the edge Kruskal's will choose in this iteration)
- If **e** is in **M**, then **T U {e}** is a subset of **M**, which is an MST, so the invariant holds



Kruskal's Algorithm: Correctness

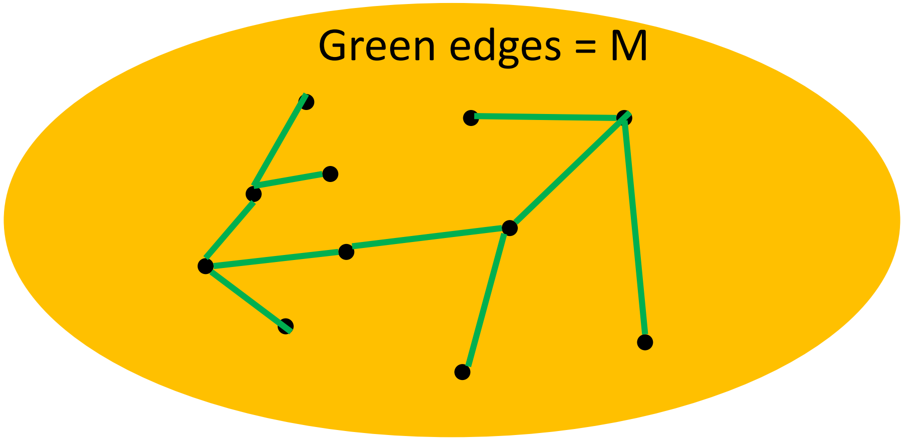
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects two vertices in different components of **T** (i.e this is the edge Kruskal's will choose in this iteration)
- If **e** is in **M**, then **T U {e}** is a subset of **M**, which is an MST, so the invariant holds



Kruskal's Algorithm: Correctness

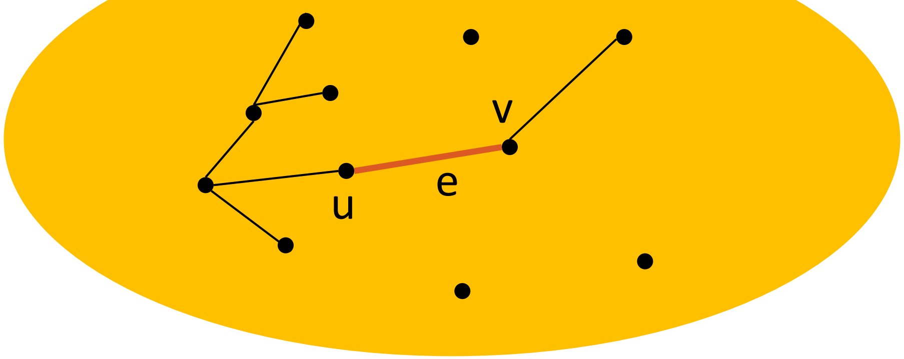
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects two vertices in different components of **T** (i.e this is the edge Kruskal's will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds
- The interesting case is where **e** is not in **M**. In this case we have to show that there is some other MST which contains **T ∪ {e}**



Kruskal's Algorithm: Correctness

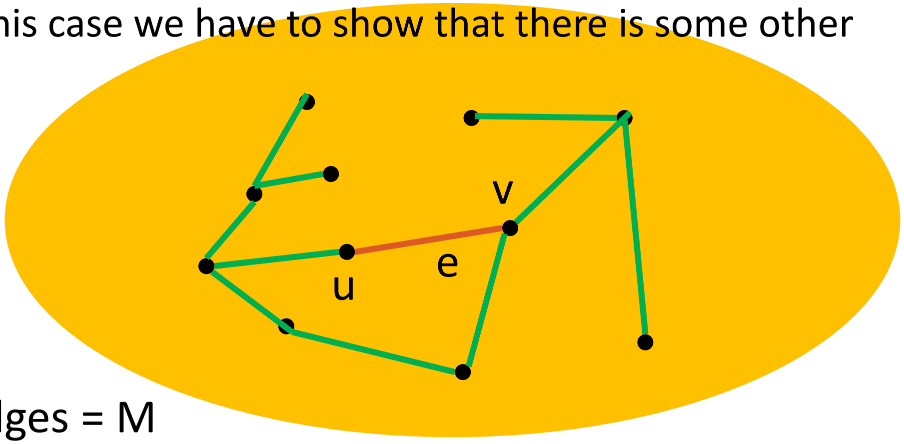
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects two vertices in different components of **T** (i.e this is the edge Kruskal's will choose in this iteration)
- If **e** is in **M**, then **T ∪ {e}** is a subset of **M**, which is an MST, so the invariant holds
- The interesting case is where **e** is not in **M**. In this case we have to show that there is some other MST which contains **T ∪ {e}**

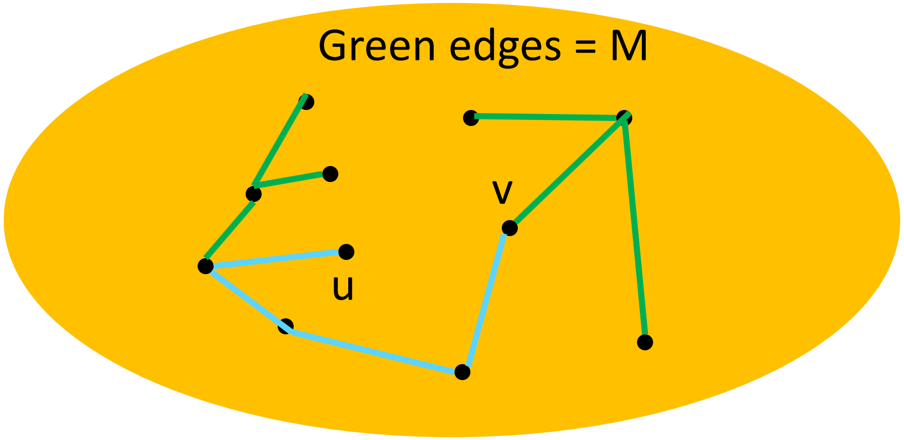


Green edges = M

Kruskal's Algorithm: Correctness

Inductive step:

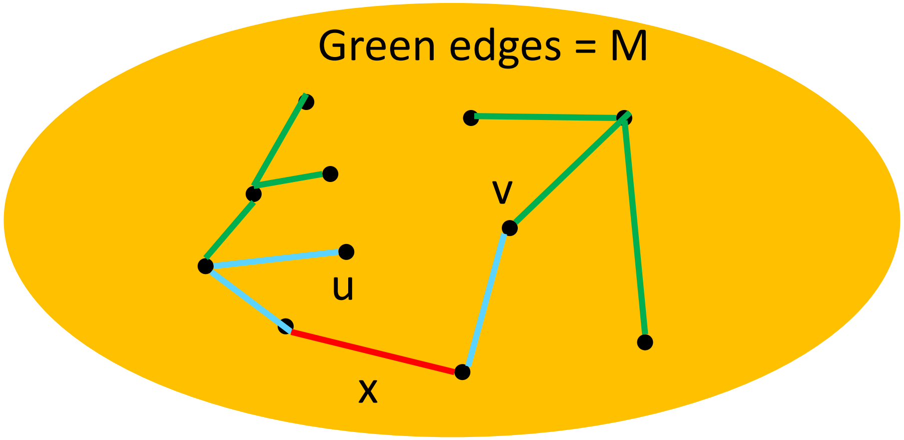
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- **u** and **v** are not connected in **T** (since we are adding **(u,v)**, and we never create a cycle)
- Consider the first edge on the blue path which is **not** contained in **T** (call this edge **x**).



Kruskal’s Algorithm: Correctness

Inductive step:

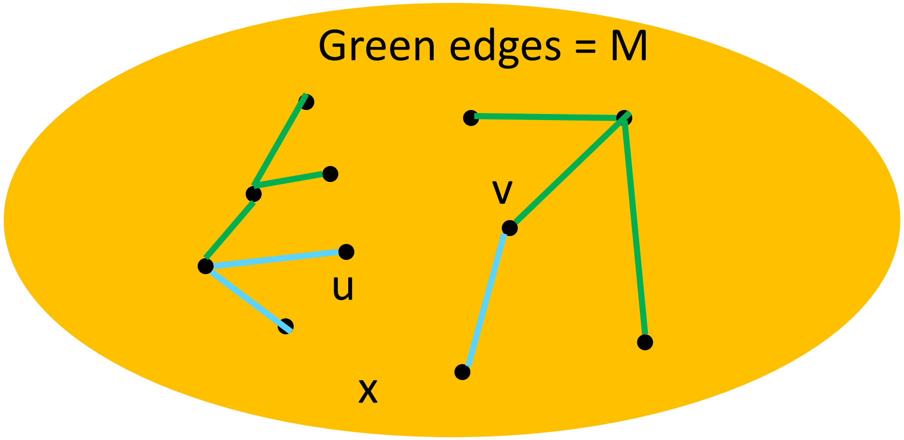
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since we are adding (u,v), and we never create a cycle)
- At least one edge on the blue path is not currently in **T** (otherwise **u** and **v** would be connected in **T**)
- Removing this edge would disconnect **M**



Kruskal’s Algorithm: Correctness

Inductive step:

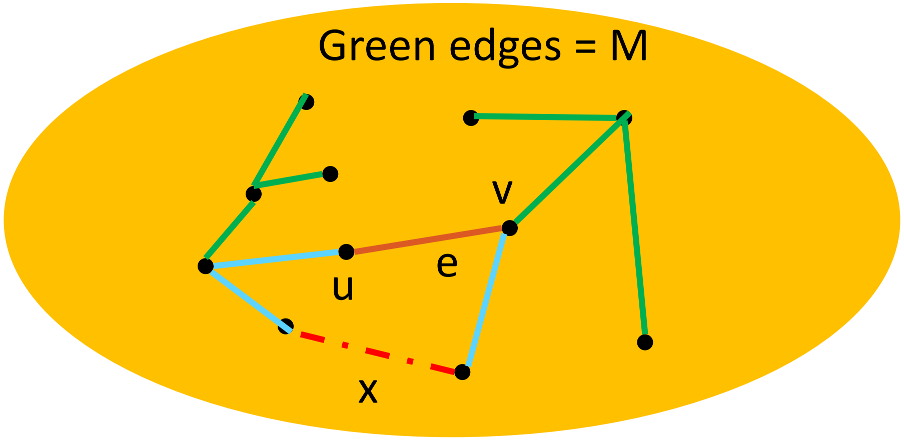
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since we are adding (u,v), and we never create a cycle)
- At least one edge on the blue path is not currently in **T** (otherwise **u** and **v** would be connected in **T**)
- Removing this edge would disconnect **M**



Kruskal's Algorithm: Correctness

Inductive step:

- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since we are adding (u,v), and we never create a cycle)
- At least one edge on the blue path is not currently in **T** (otherwise **u** and **v** would be connected in **T**)
- Removing this edge would disconnect **M**
- Removing this edge would disconnect **M**
- Adding the edge **(u,v)** would form a new spanning tree, **M'**
- Since the algorithm always selects the lightest edge incident to **T**, we know that **w(e) ≤ w(x)**
- So the weight of **M'** is no greater than the weight of **M**, therefore choosing e is correct



Summary

Take home message

- Prim's Algorithm and Kruskal's algorithm both are greedy algorithm that correctly determine minimum spanning trees.

Things to do (this list is not exhaustive)

- Make sure you understand
  - the two algorithms especially how to implement Union-Find data structure for Kruskal's algorithm
  - the proofs of correctness for each of the two algorithms
- Start preparing for the final exam

Coming Up Next

- Network Flow