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.

Recommended reading

- Lecture Notes: Chapters 14 and 15

- Cormen et al. Introduction to Algorithms.
  - Chapter 23

Outline

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

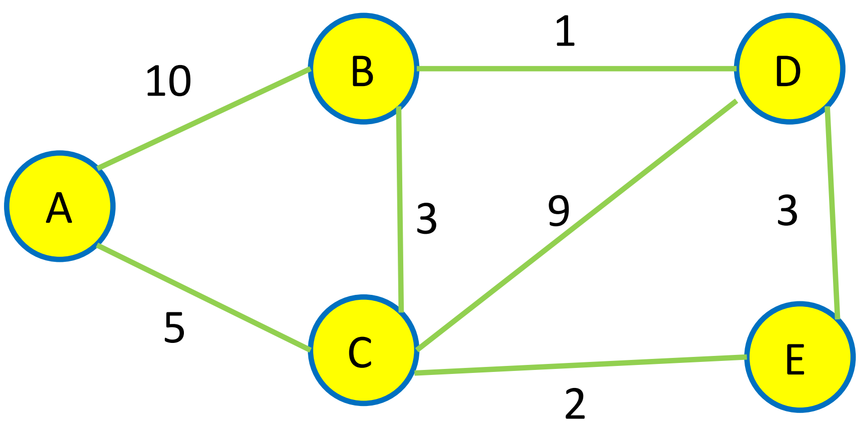
What is a Spanning Tree

Tree:

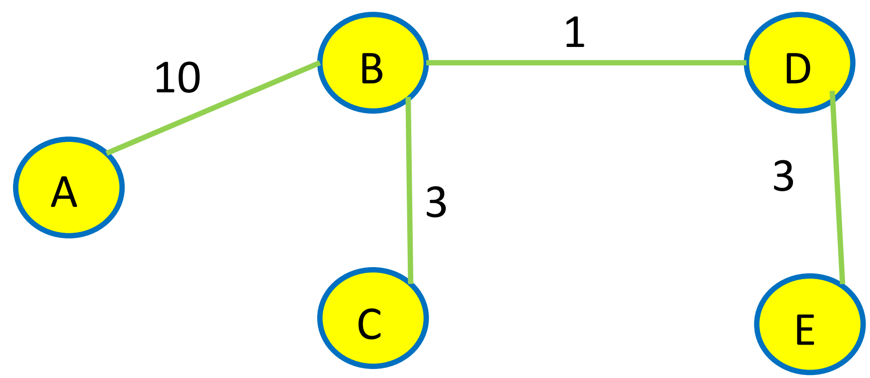
A tree is a connected undirected graph with no cycles in it.

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).

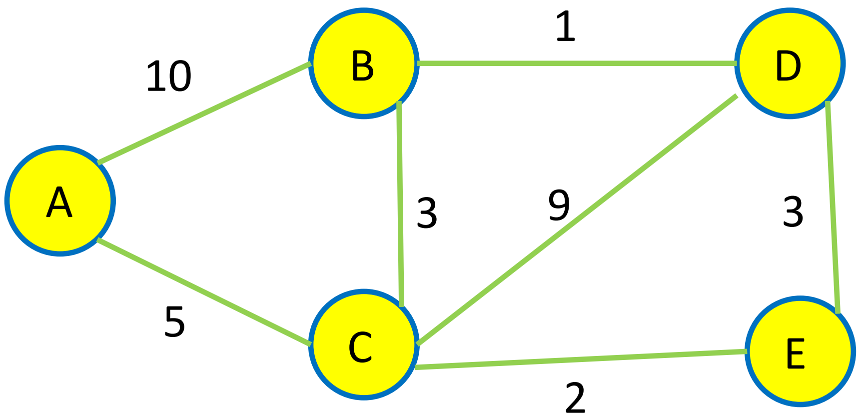


An undirected graph

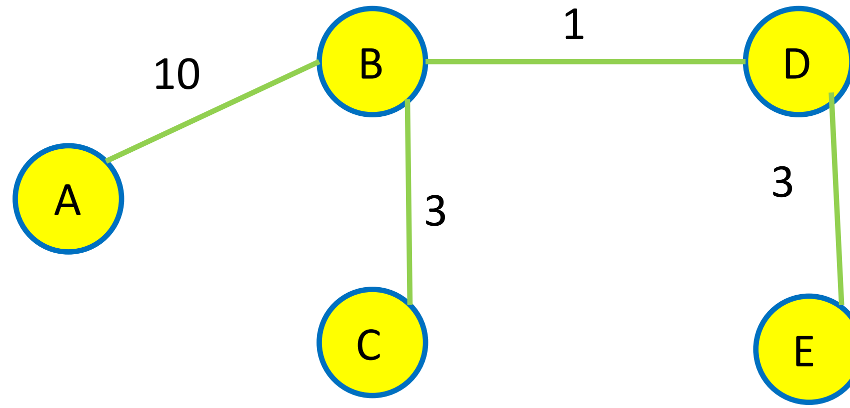


Spanning Tree

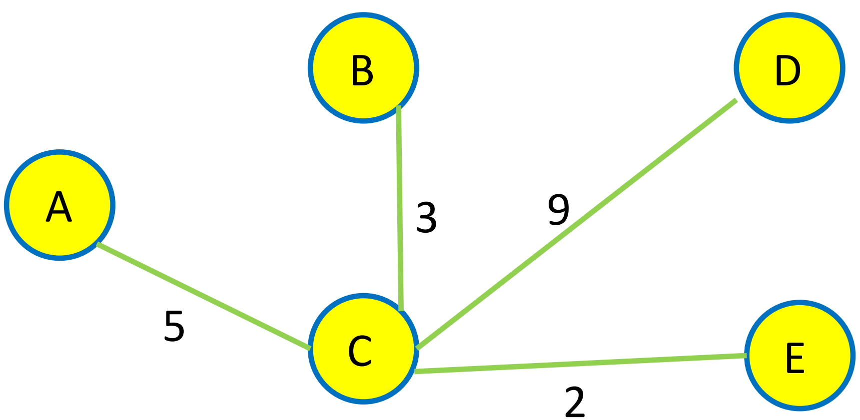
Spanning Tree Examples



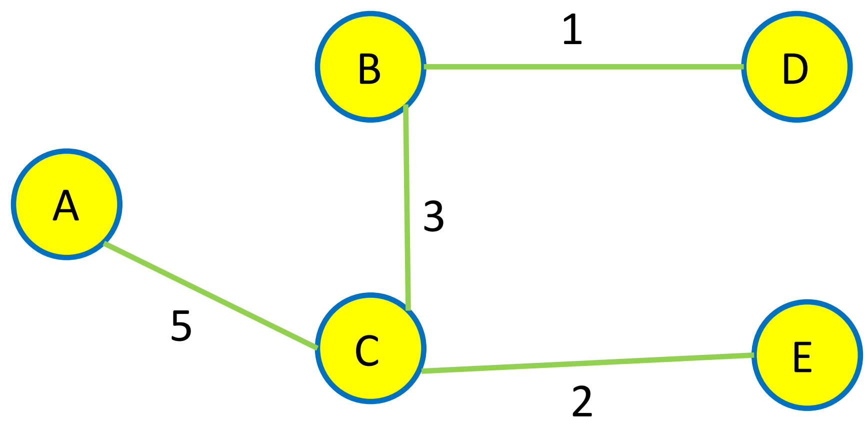
Graph



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:

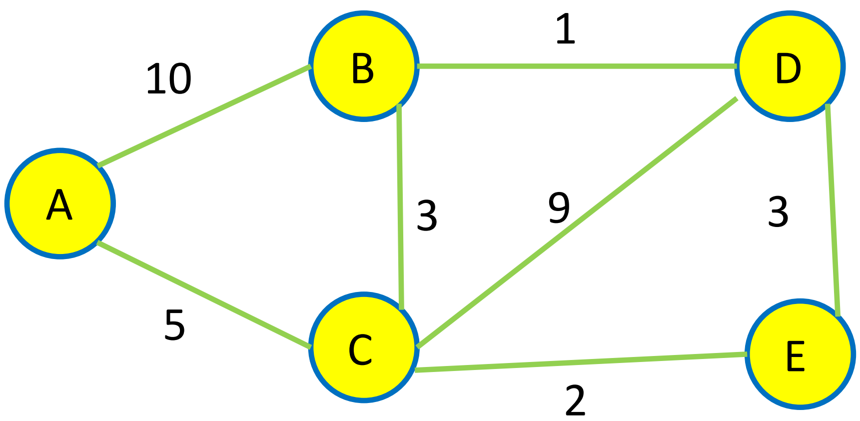
- A spanning tree of a general undirected weighted graph G is a tree that spans G (i.e., a tree that includes every vertex of G) and is a subgraph of G (i.e., every edge in the spanning tree belongs to G).
- A maximal set of edges of G that contains no cycles
- A minimal set of edges that connects all vertices

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

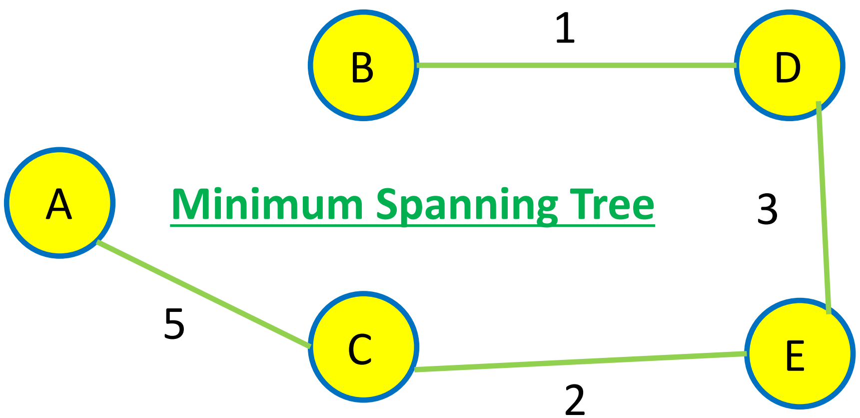
Minimum Spanning Tree (MST)

- 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).

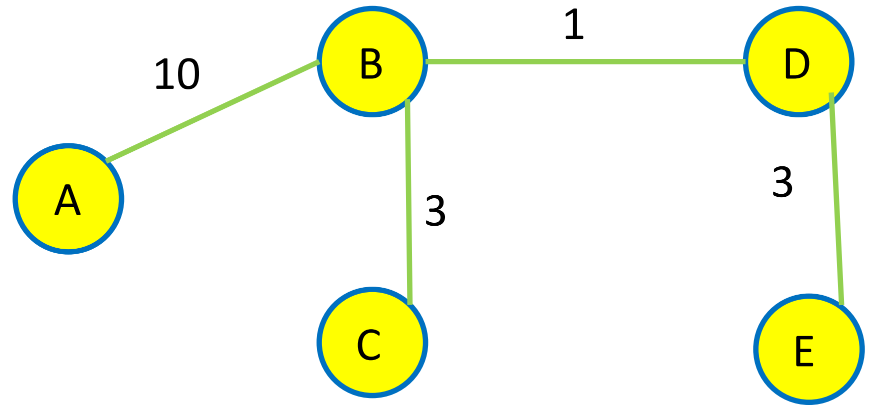
Quiz time!
https://flux.qa - YTJMAZ



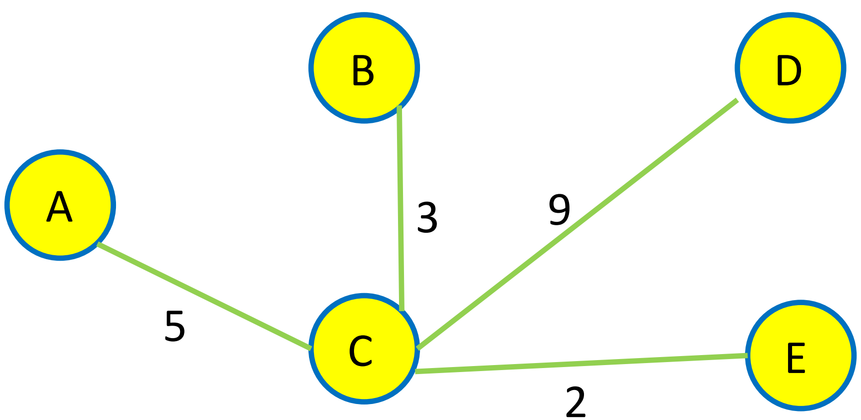
Spanning Trees and MSTs



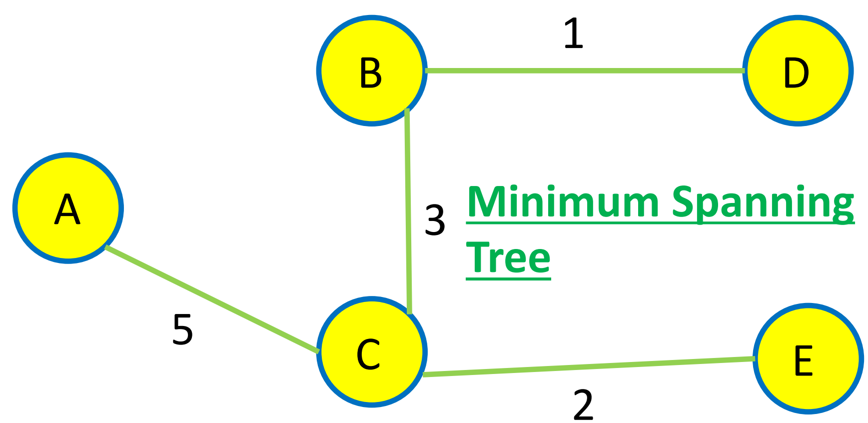
Spanning Tree 1: Weight 11



Spanning Tree 2: Weight 17



Spanning Tree 3: Weight 19



Spanning Tree 4: Weight 11

MST Algorithms

The main MCST algorithms work by incrementally adding edges

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

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

General Strategy:

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

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

MST Algorithms

- Prim's Algorithm (very similar to Dijkstra's Algorithm)
  - We start with no edges in T.
  - We select one node as the source.
  - At each iteration, we add to T the shortest edge that connects T to a node outside of T
  - In each iteration, the number of nodes not connected decreases by 1 and the number of nodes in T increases by 1.
  - Cycles are avoided as no edges linking two nodes that are already in T are ever added.
  - Time complexity: O(E log V)

MST Algorithms

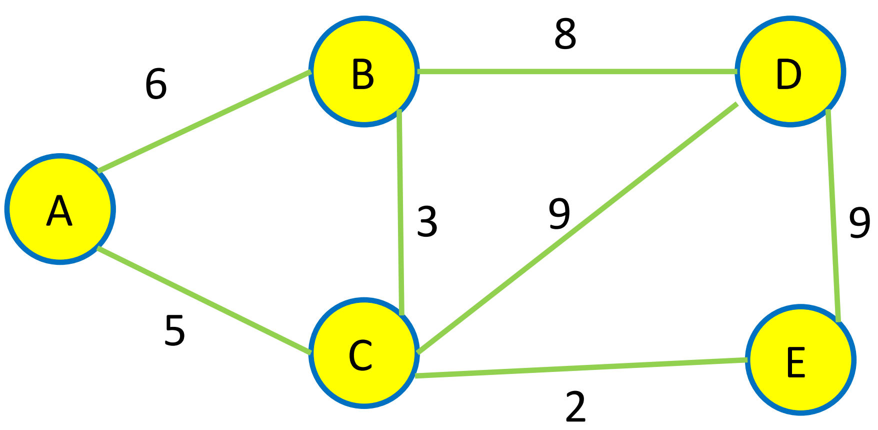
- Kruskal's Algorithm
  - We do not just grow a tree but a forest (a set of trees)
  - We start with an empty forest, so there are V connected components in T (each with a single node).
  - We add edges in ascending order of weight, provided they do not introduce a cycle.
  - Each time that one edge e is added to T, e is the smallest edge that links two distinct connected components of T.
  - Cycles are avoided as no edges linking two nodes that are already in the same connected component are ever added to T.
  - Time complexity: O(E log V)

Outline

1. Introduction
2. Prim's Algorithm
   - it's really Jarnik's Algorithm (1930)
   - but commonly called "Prim's" (1957)
3. Kruskal's Algorithm

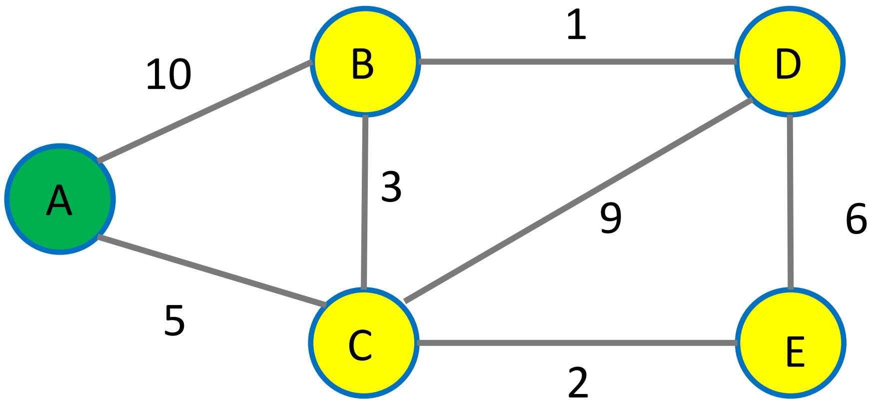
Prim's Algorithm: Overview

- Start by picking any vertex v to be the source of T.
- While T does not contain **all** vertices in the graph:
  - Find shortest edge e that goes from T to a different connected component.
  - add e to the tree T.



Prim's Algorithm

u:



Q:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

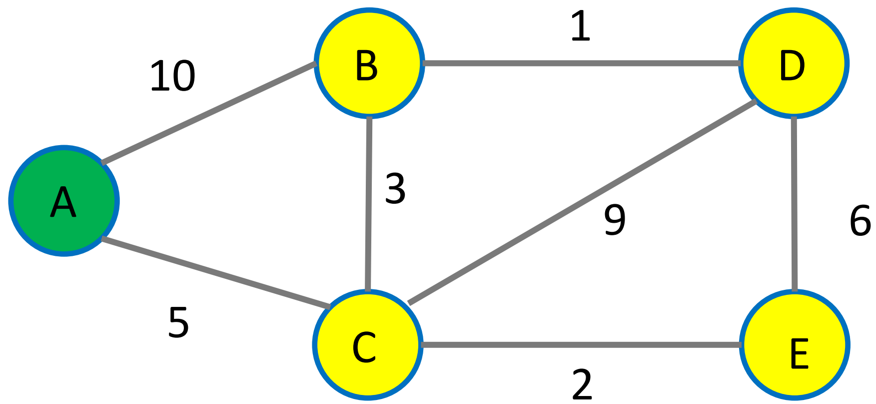
A	B	C	D	E
0	Inf	Inf	Inf	inf

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
-	-	-	-	-

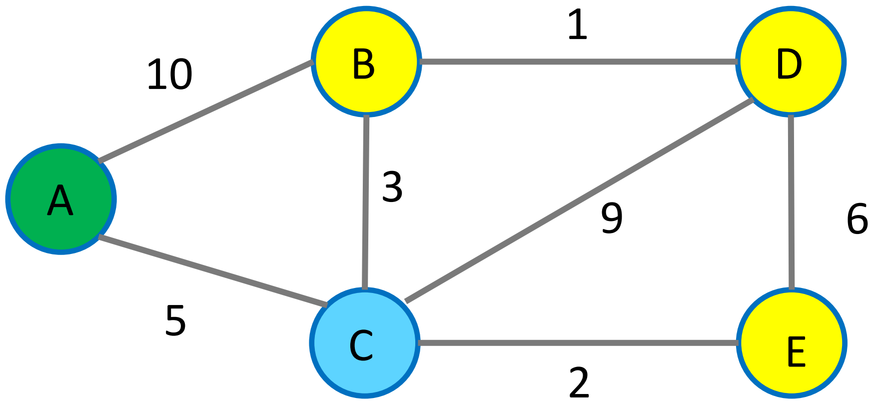
Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

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

For each neighbour v of u, try to update distance

5 < inf

Dist[C] = 5

Pred[C] = A

Pred:

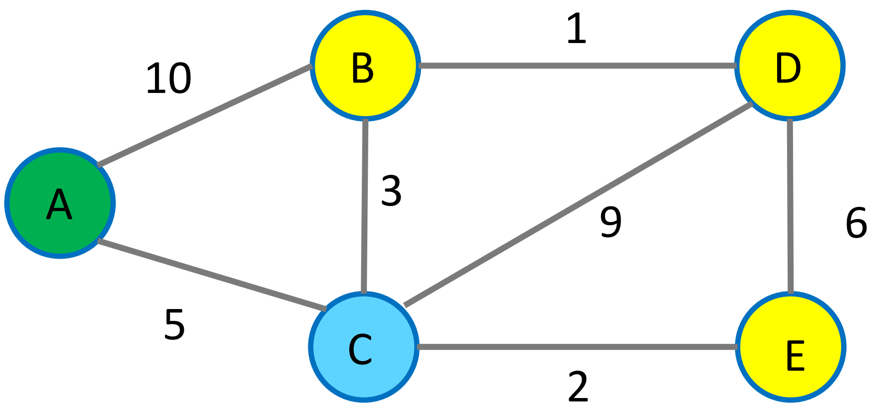
A	B	C	D	E
-	-	-	-	-

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:

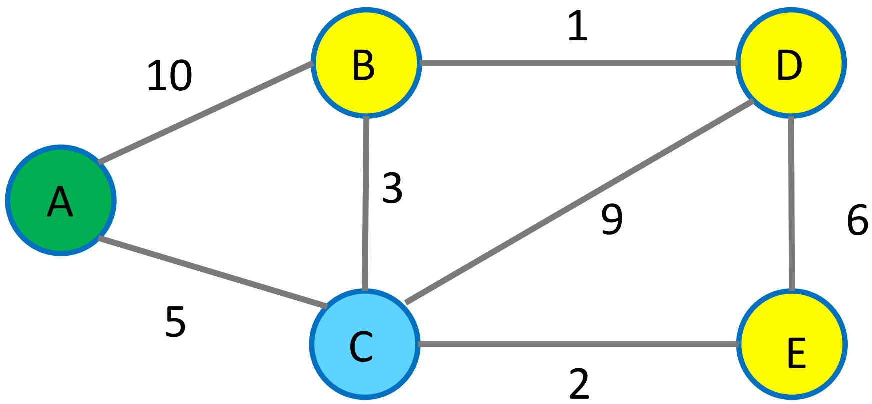
A	B	C	D	E
0	Inf	Inf	Inf	inf

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:

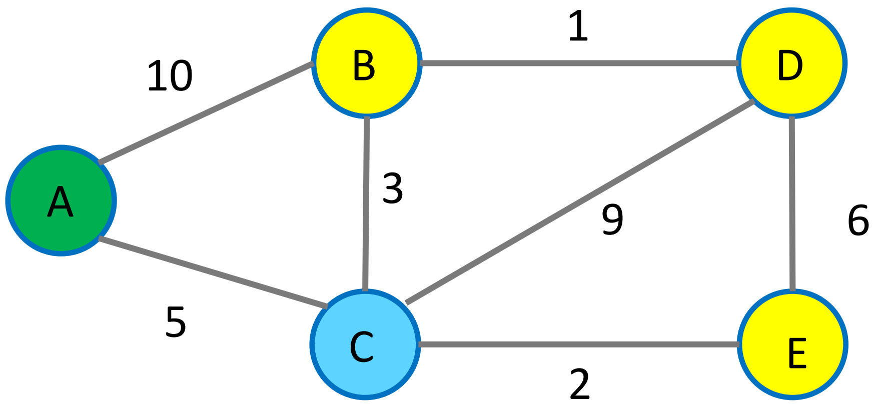
A	B	C	D	E
0	Inf	Inf	Inf	inf

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:

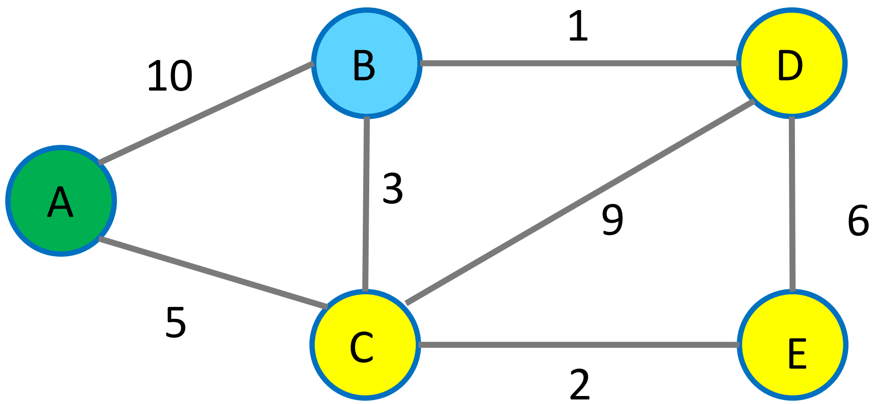
A	B	C	D	E
-	-	A	-	-

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

Pred:

A	B	C	D	E
-	-	A	-	-

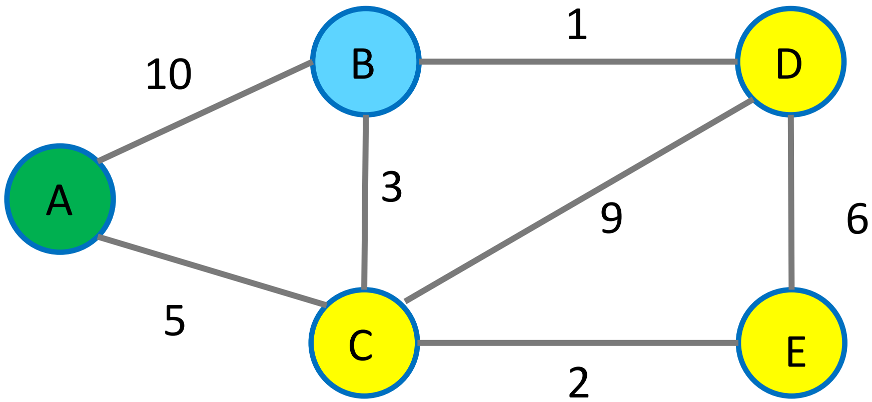
Dist:

A	B	C	D	E
0	10	5	Inf	inf

Doing the same for B

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

Prim's Algorithm



u: A

Q:

C	B	D	E
5	10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

Dist:

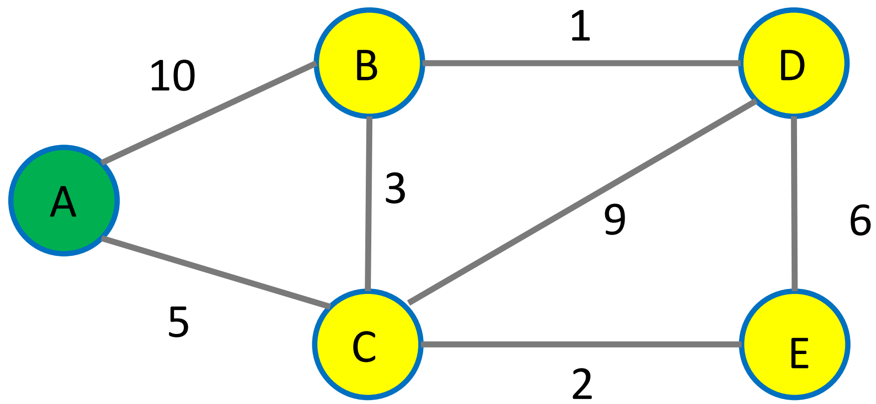
A	B	C	D	E
0	10	5	Inf	inf

Doing the same for B

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

Prim's Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

Pred:

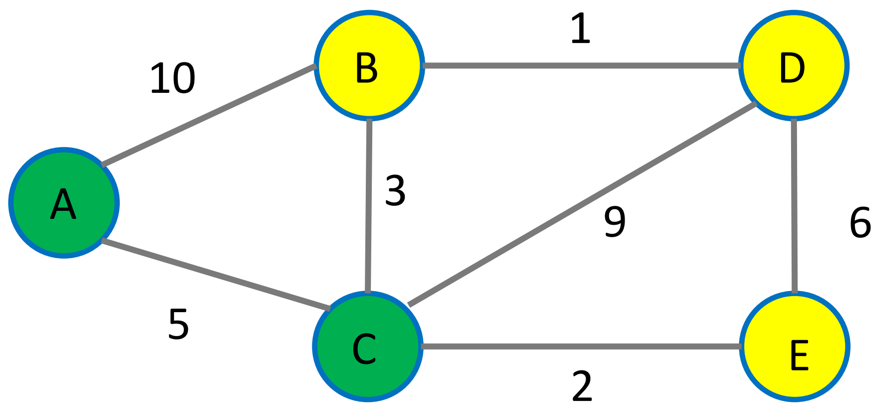
A	B	C	D	E
-	A	A	-	-

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	-	-

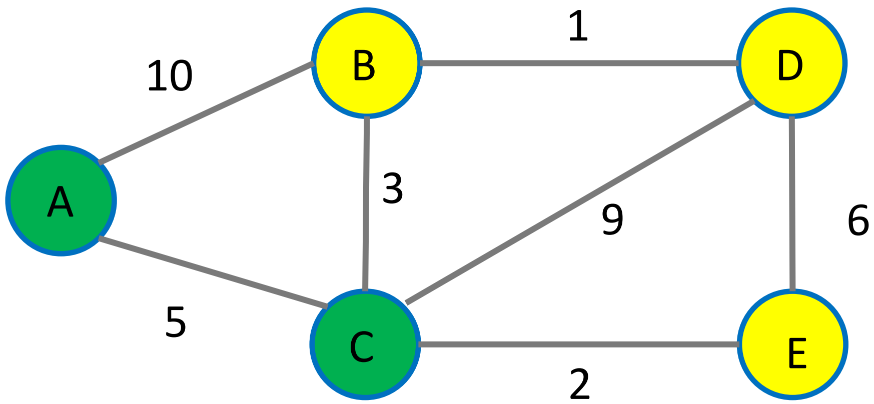
Dist:

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	-	-

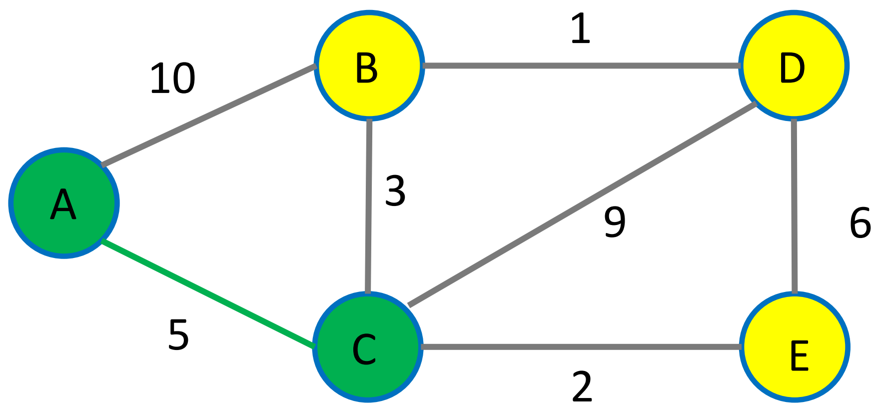
Dist:

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:

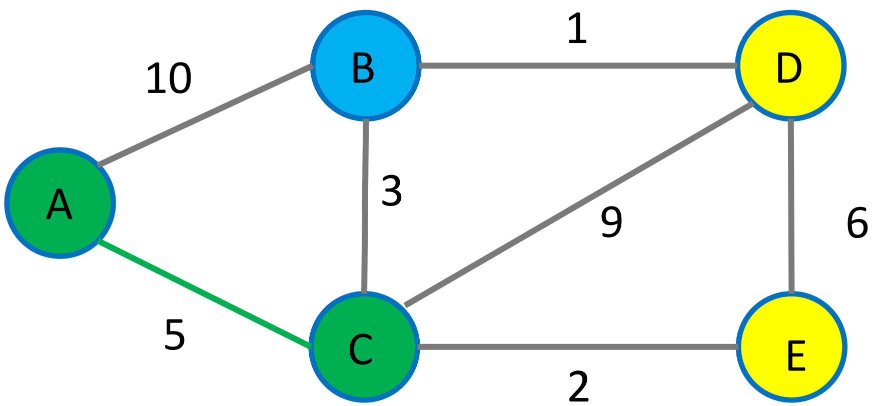
A	B	C	D	E
-	A	A	-	-

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

Pred:

A	B	C	D	E
-	A	A	-	-

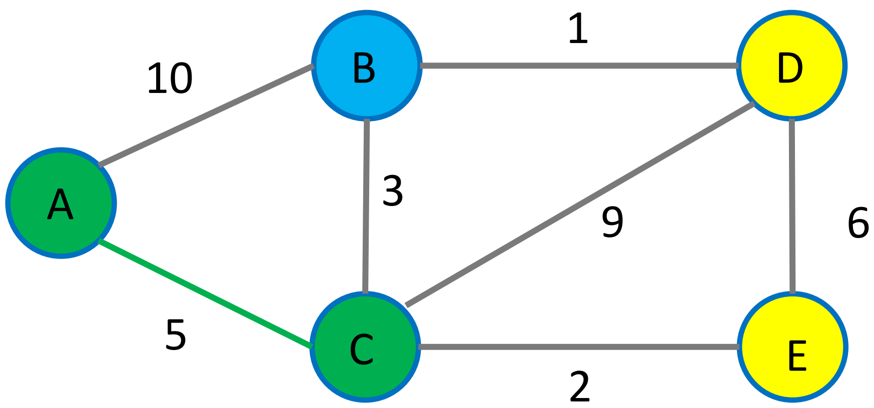
Dist:

A	B	C	D	E
0	10	5	Inf	inf

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

Prim's Algorithm

u: C



Q:

B	D	E
3	Inf	Inf

Pred:

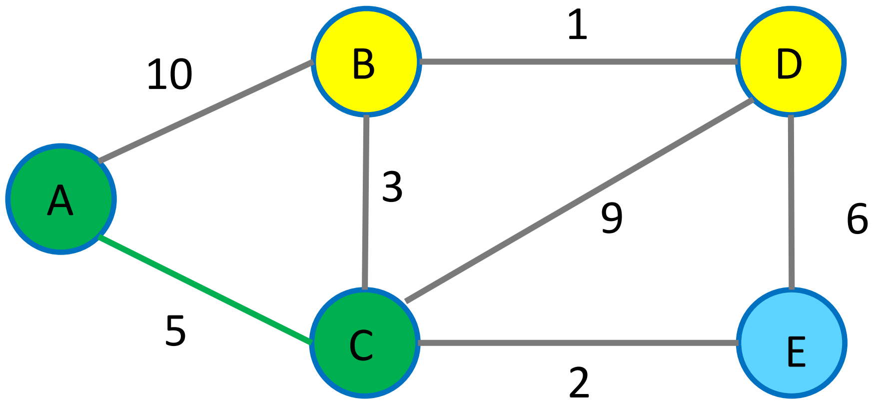
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	3	5	Inf	inf

Prim's Algorithm

u: C



- update E from C

Q:

B	D	E
3	Inf	Inf

Pred:

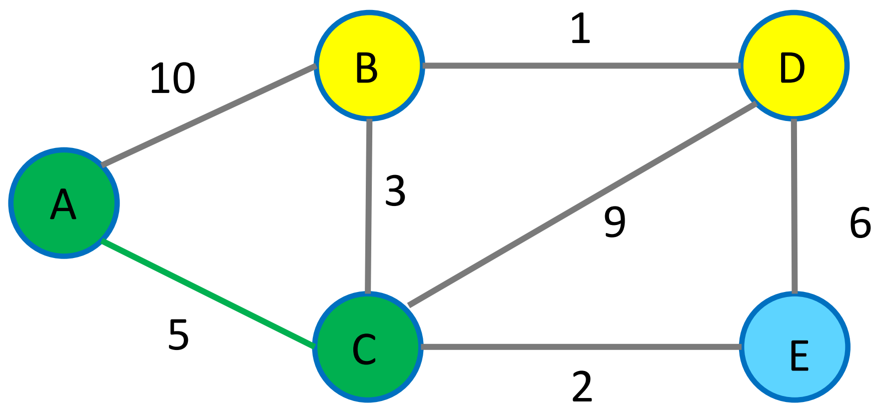
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	3	5	Inf	inf

Prim’s Algorithm

u: C



Q:

E	B	D
2	3	inf

Pred:

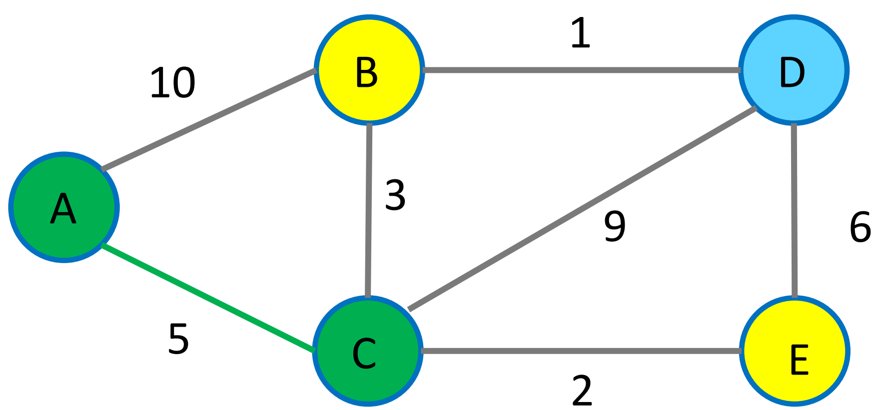
A	B	C	D	E
-	C	A	-	C

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:

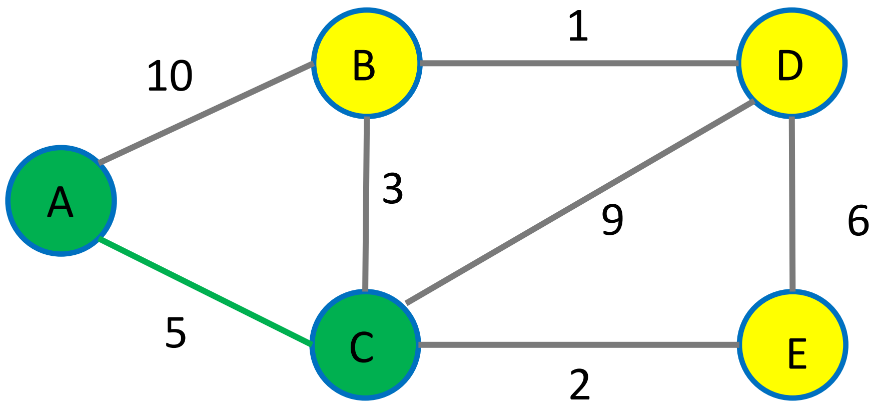
A	B	C	D	E
-	C	A	-	C

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:

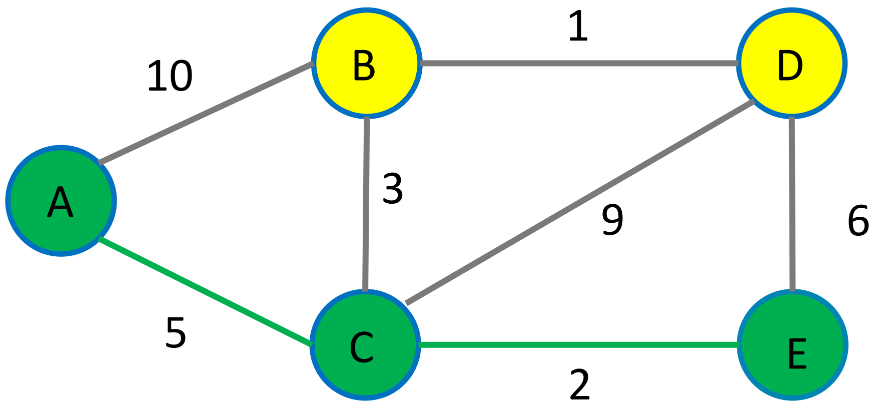
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	3	5	9	2

Prim's Algorithm

u: E



Q:

B	D
3	9

Pred:

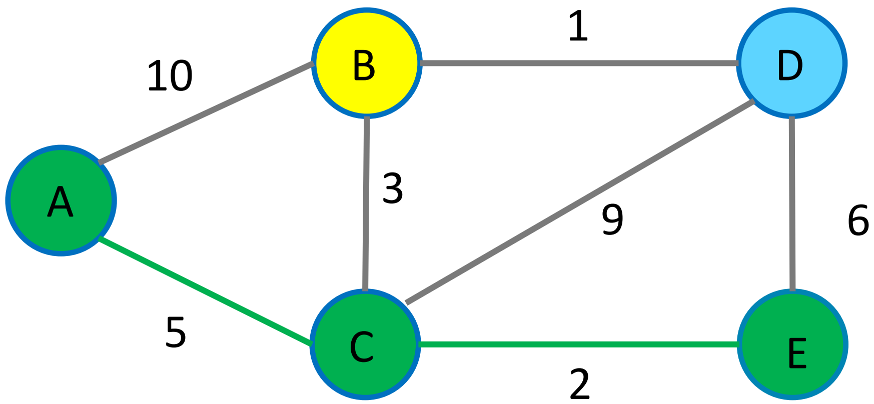
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	3	5	9	2

Prim’s Algorithm

u: E



Q:

B	D
3	6

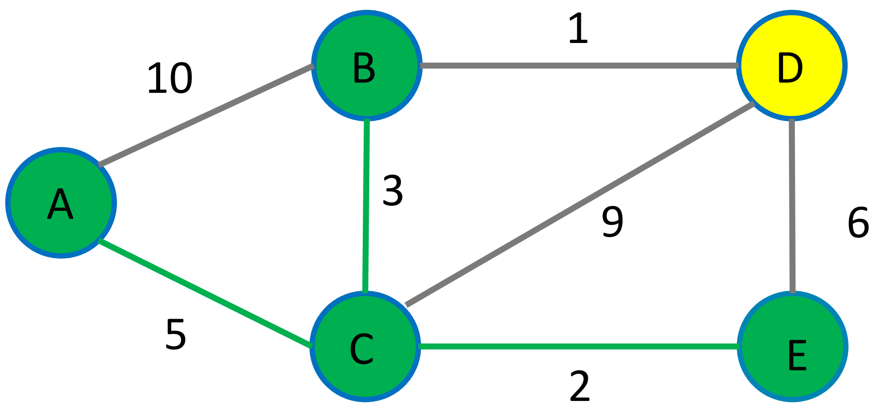
Pred:

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

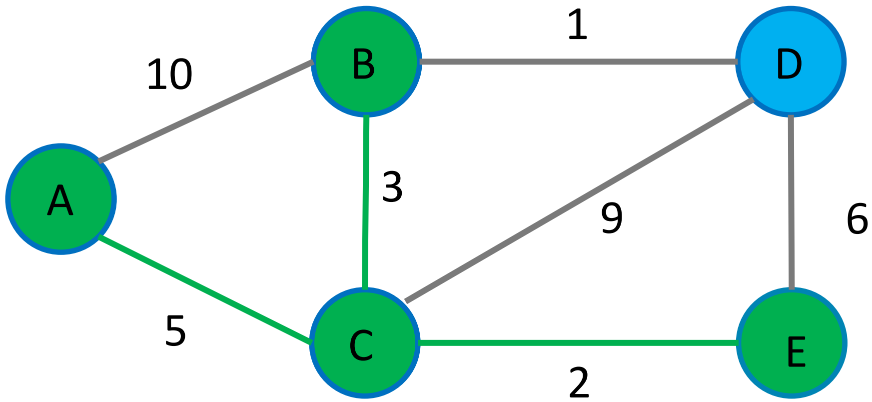
Pred:

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:

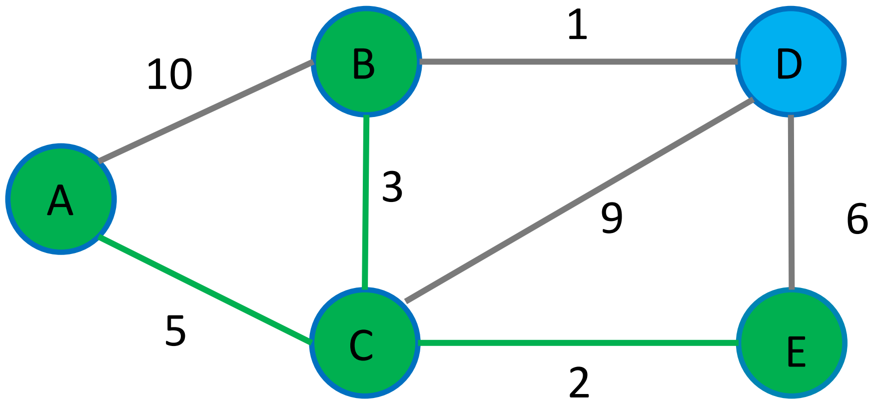
A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	3	5	6	2

Prim's Algorithm

u: B



Q:

D
1

Pred:

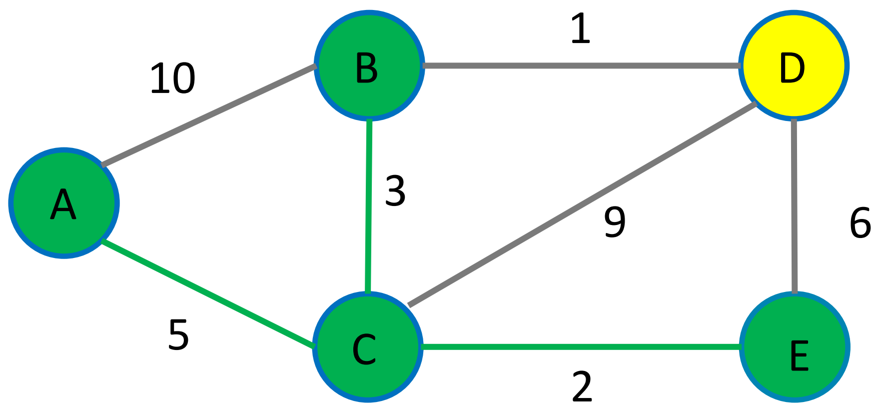
A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	3	5	1	2

Prim’s Algorithm

u: B



Q:

D
1

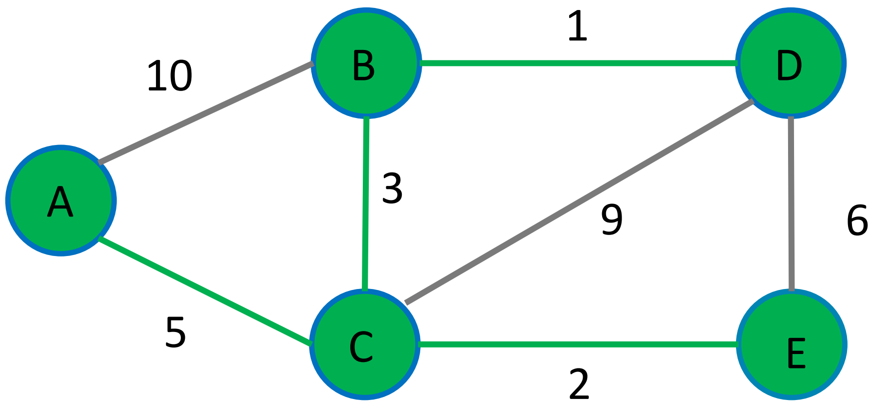
Pred:

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:

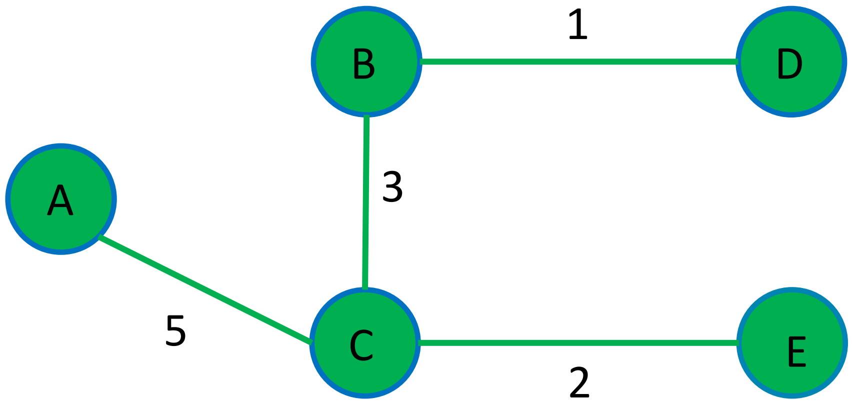
Pred:

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 and adjacency list is used

- Init
  - O(V + deg(start)) constant time operations = O(V)
- Main loop
  - Graph Operations
    - Incident edges retrieved once for each vertex: $O(\Sigma_w \deg(w)) = O(2E) = O(E)$
  - Priority queue operations
    - Each vertex is inserted once and removed once
      - Insertion and removal take O(log V) time: O(V log V)
    - Each vertex is updated at most deg(w) times in the queue
      - vertex priority update takes O(log V) time: $O(\Sigma_w \deg(w) \log V)) = O(2E \log V) = O(E \log V)$
  - Vertex information updates
    - Vertex updates apart from priority queue take contant time and happen deg(w) times: O(E)
- Total: O(V)+O(E)+O(V log V) + O(E log V)=O((V+E) log V))
  - Since the input graph G is connected, E ≥ V-1. Hence, the complexity can be simplified to O(E log V).

O

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, $E \geq V-1$. Hence, the complexity can be simplified to $O(E \log V)$.

Prim’s Algorithm: Correctness

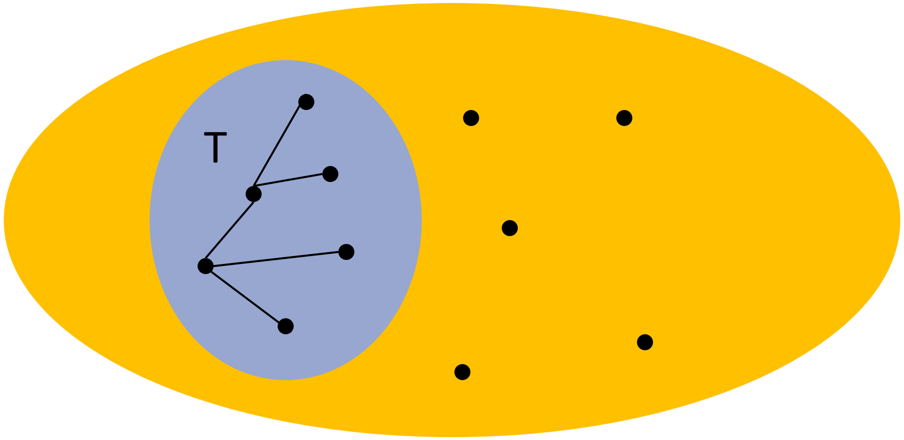
#INV: Every iteration of Prim’s algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

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

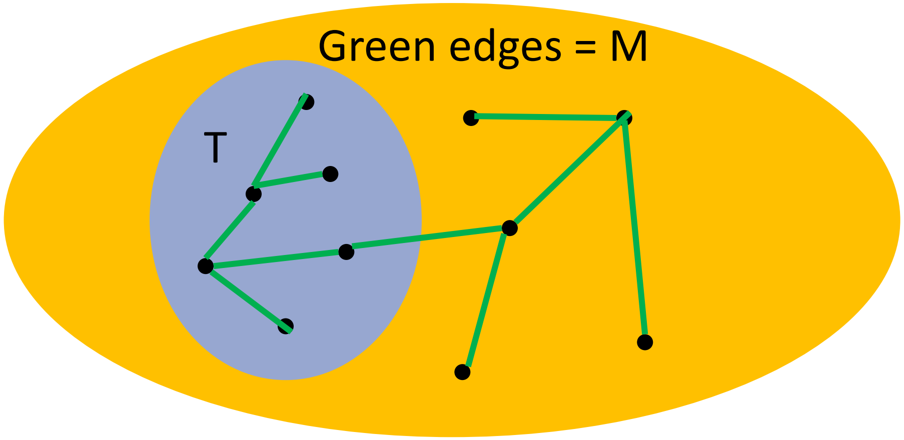
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

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

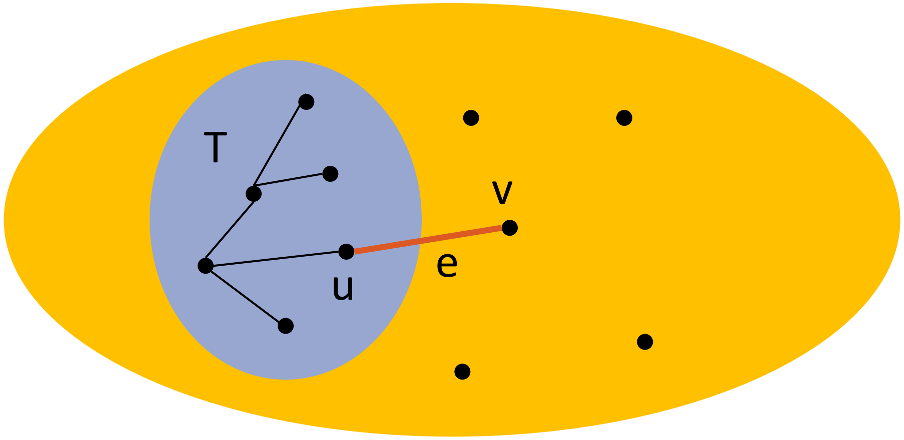
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

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

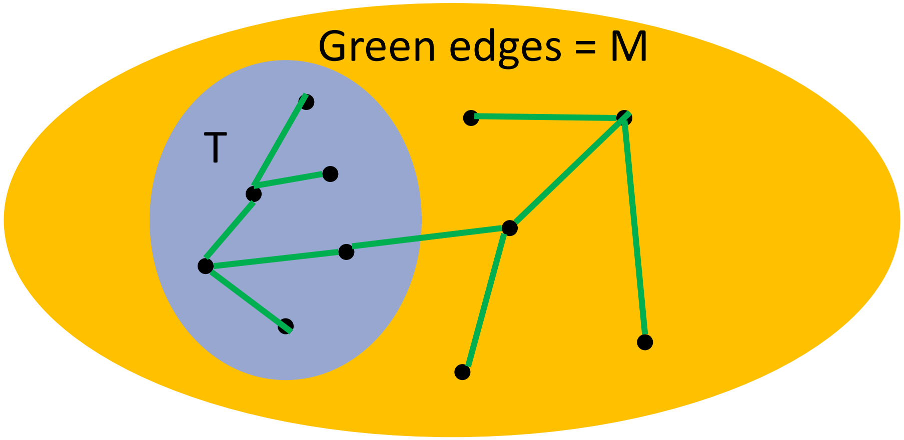
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

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

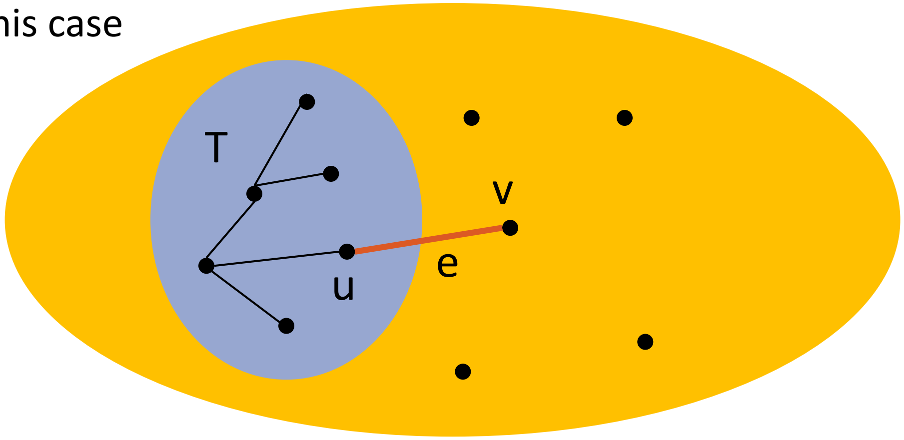
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

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

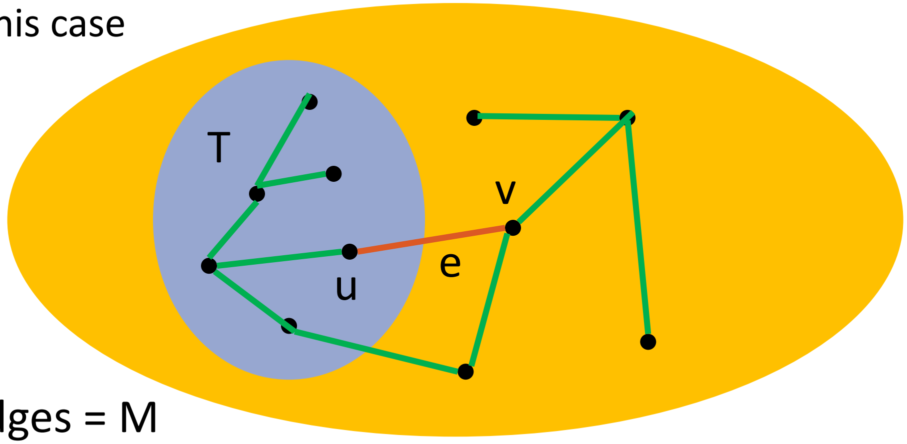
#INV: Every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

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}**

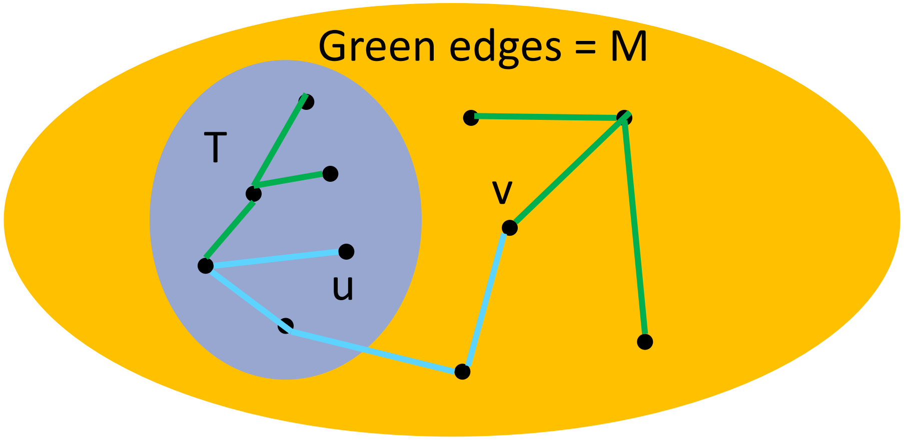


Green edges = M

Prim’s Algorithm: Correctness

Inductive step:

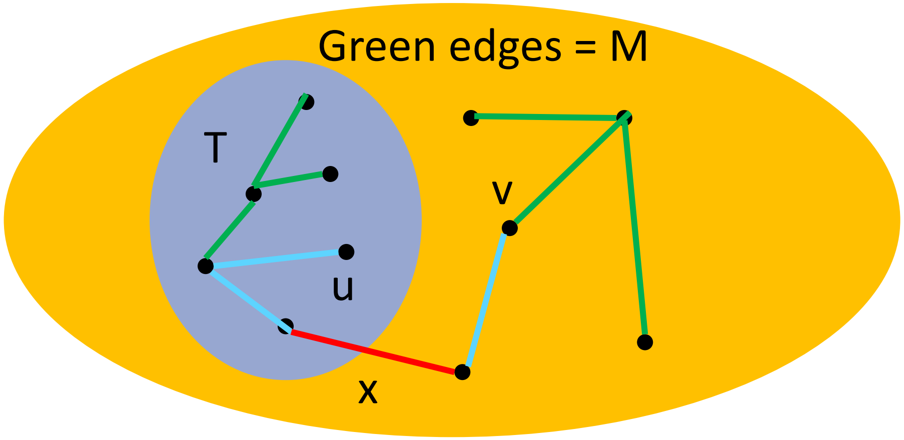
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since **v** is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



Prim's Algorithm: Correctness

Inductive step:

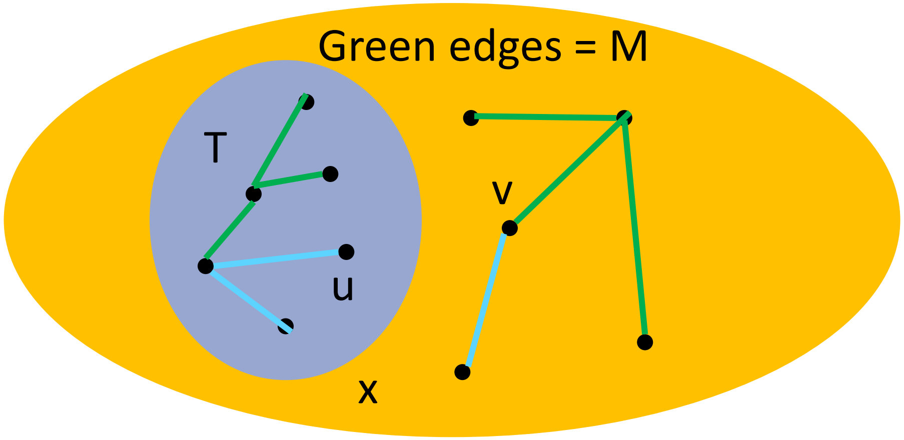
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since **v** is 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:

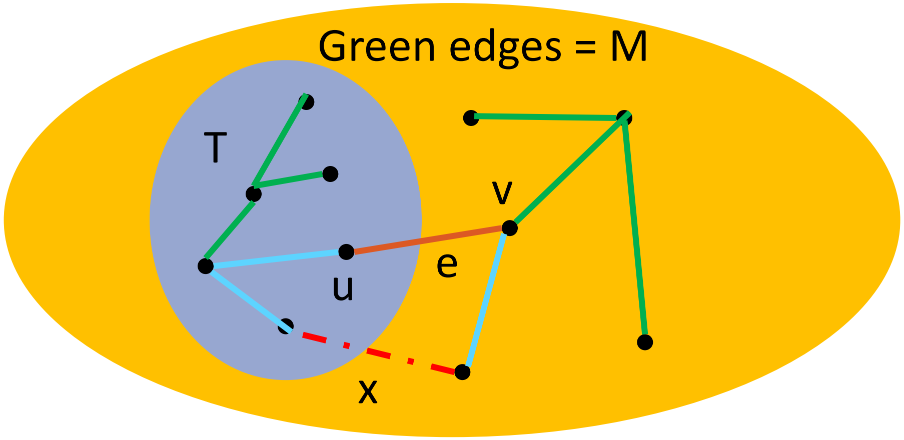
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue)
- u and v are not connected in **T** (since **v** is 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 shortest 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

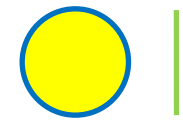


Quiz time!
https://flux.qa - YTJMAZ

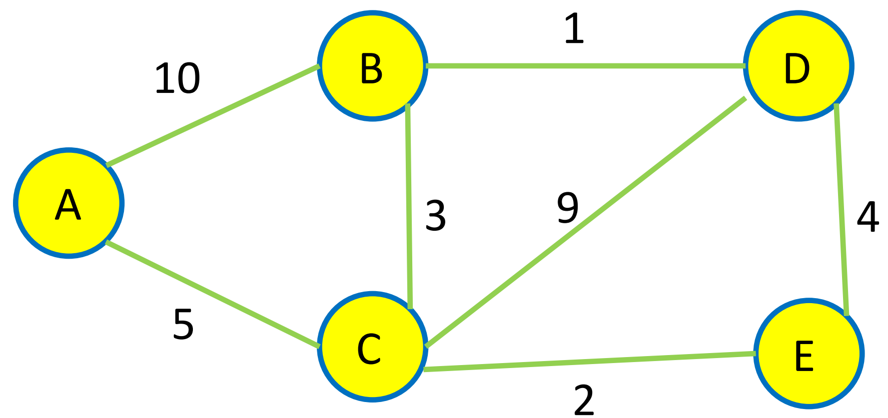
Outline

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???



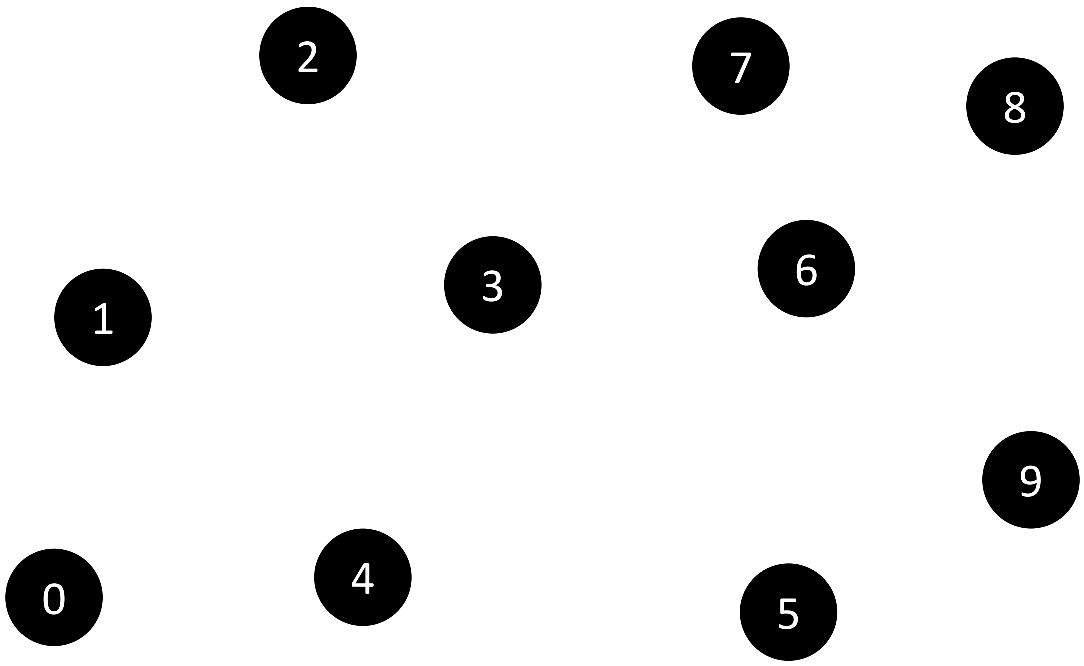
Sorted Edges:

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

Finalized (in MST):

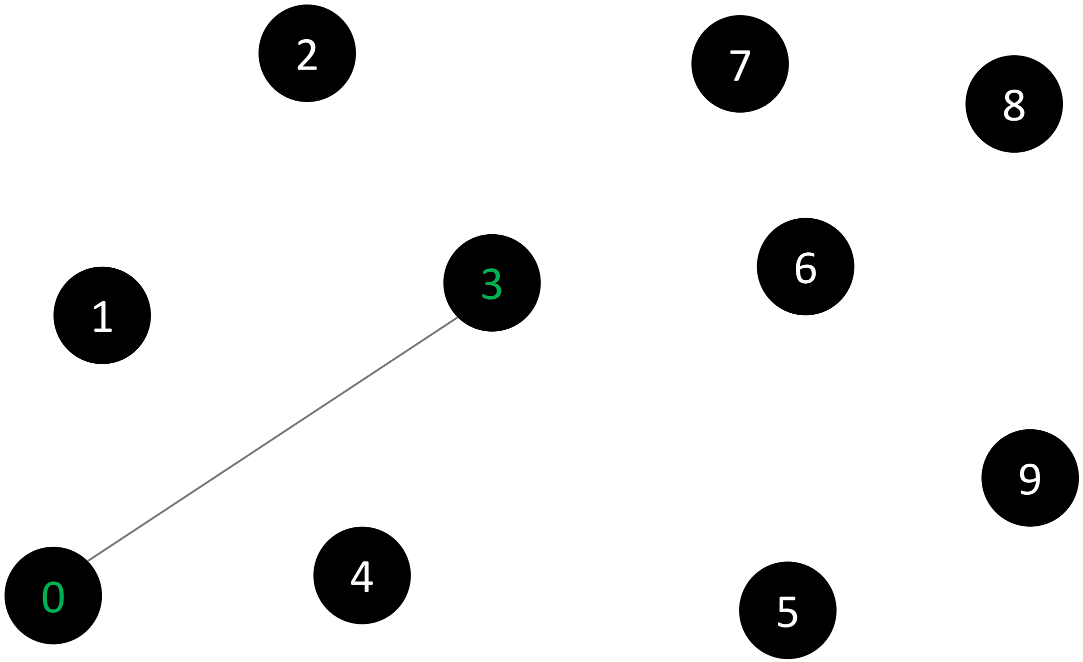
B→D	C→E	C→B	A→C

Kruskal's Algorithm



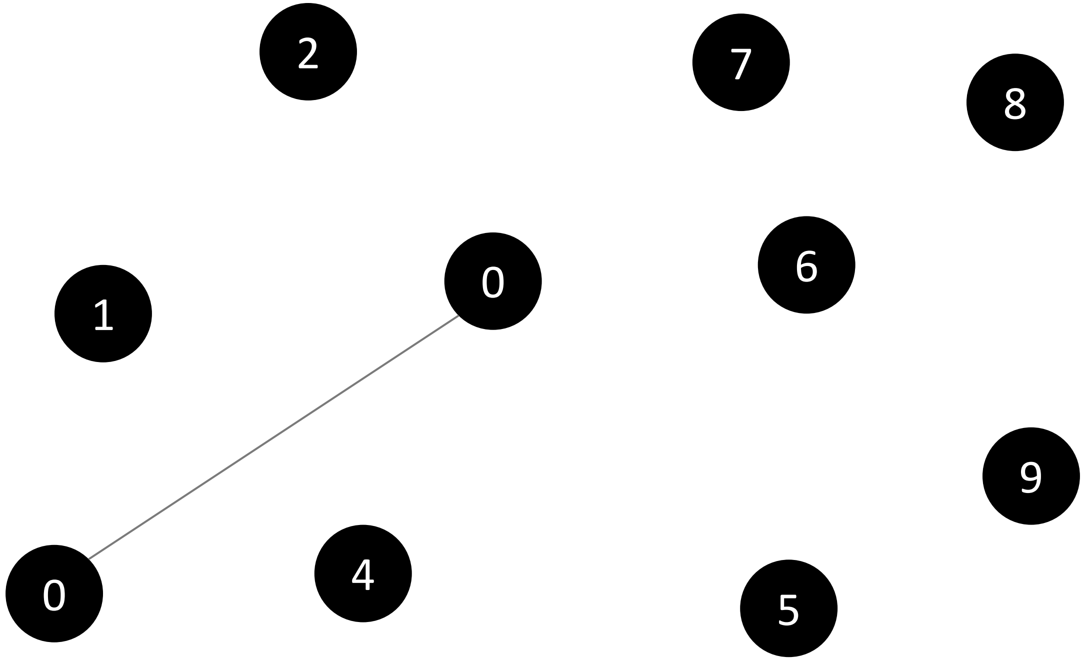
Intuitively, Kruskal's algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal's Algorithm



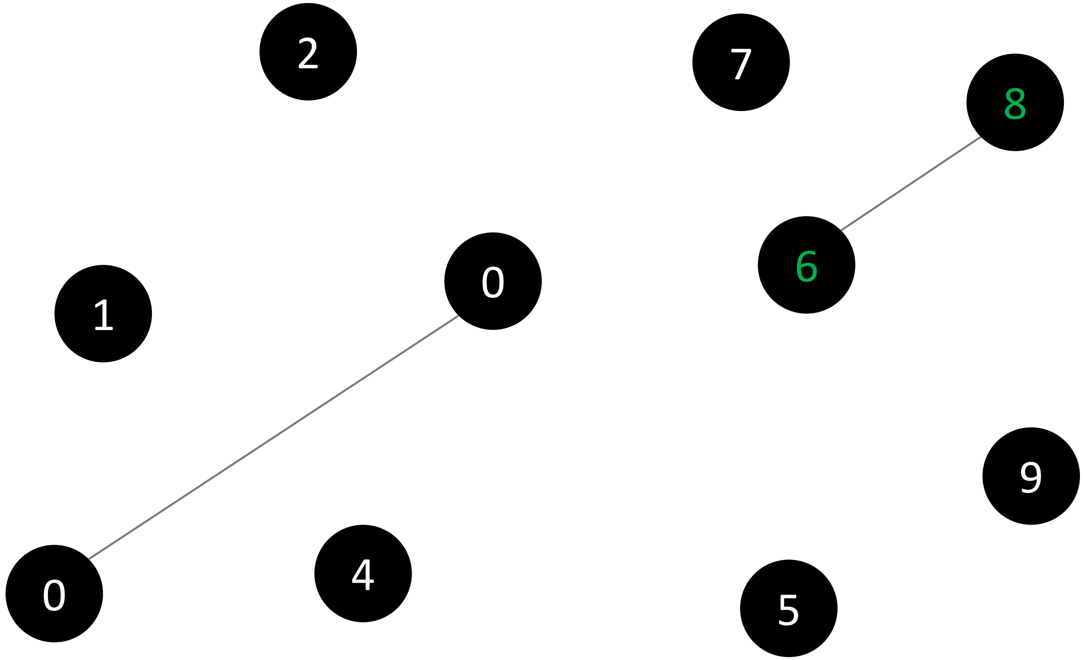
Intuitively, Kruskal's algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal's Algorithm



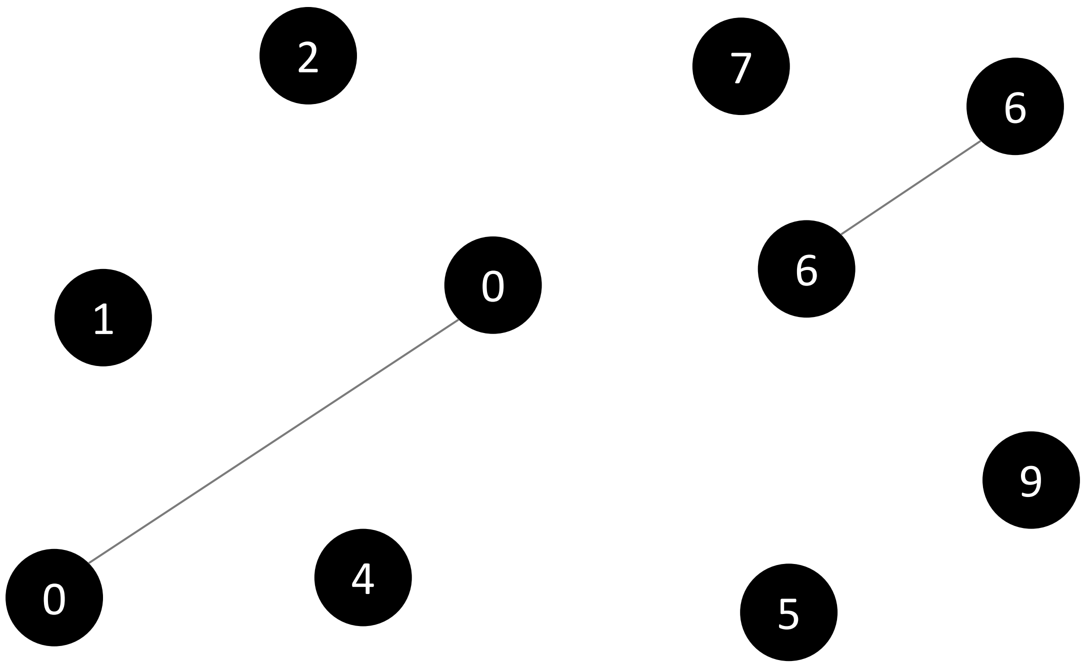
Intuitively, Kruskal's algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal's Algorithm



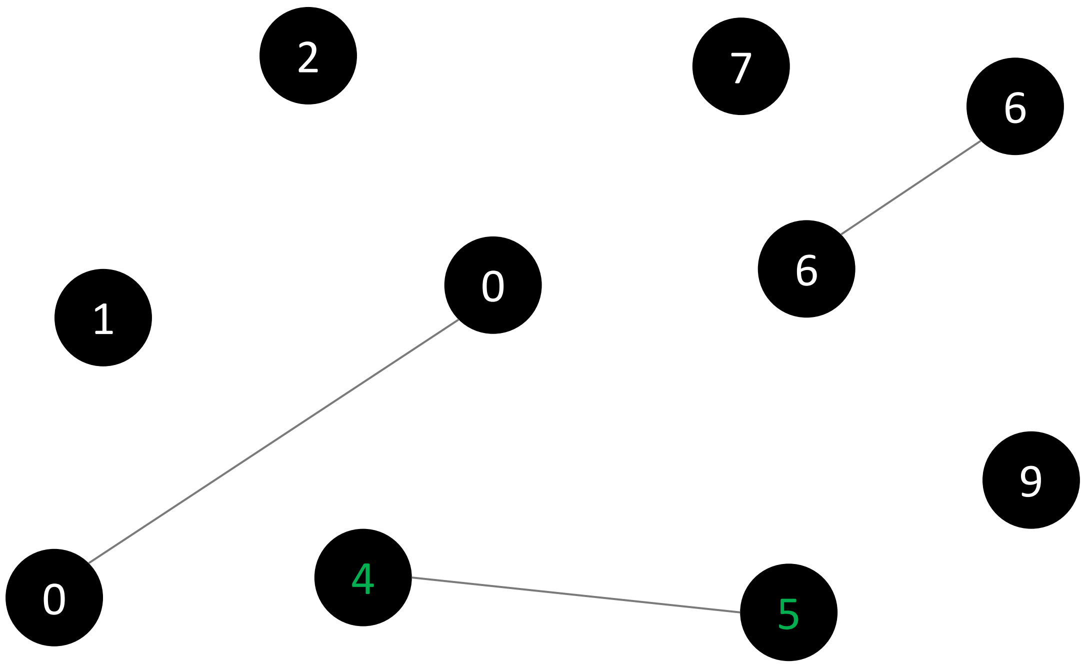
Intuitively, Kruskal's algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal’s Algorithm



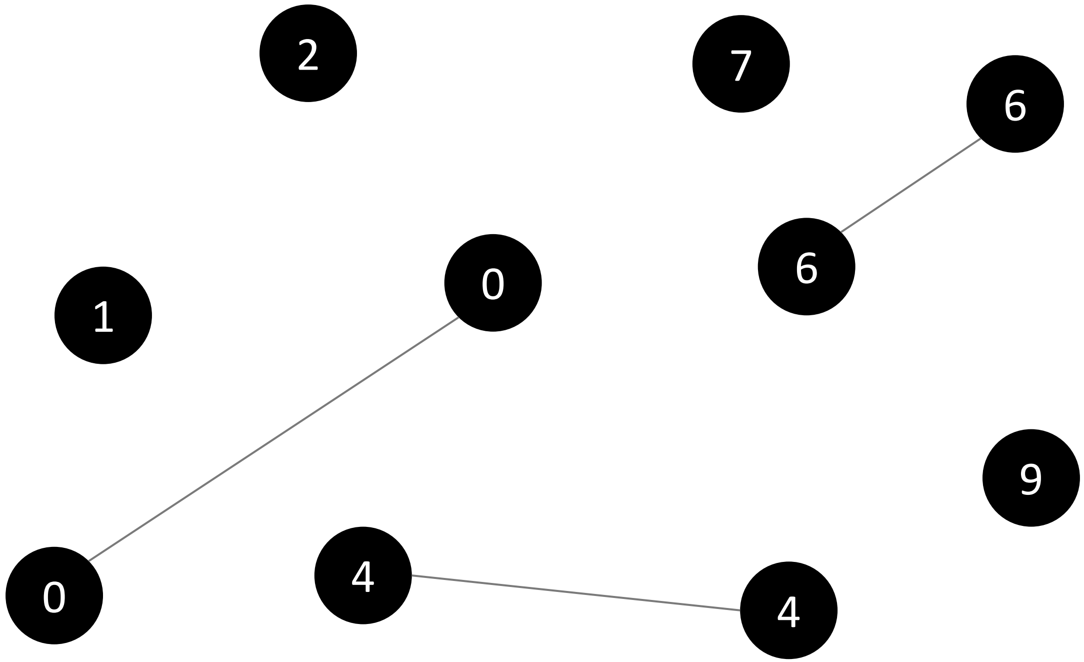
Intuitively, Kruskal’s algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal’s Algorithm



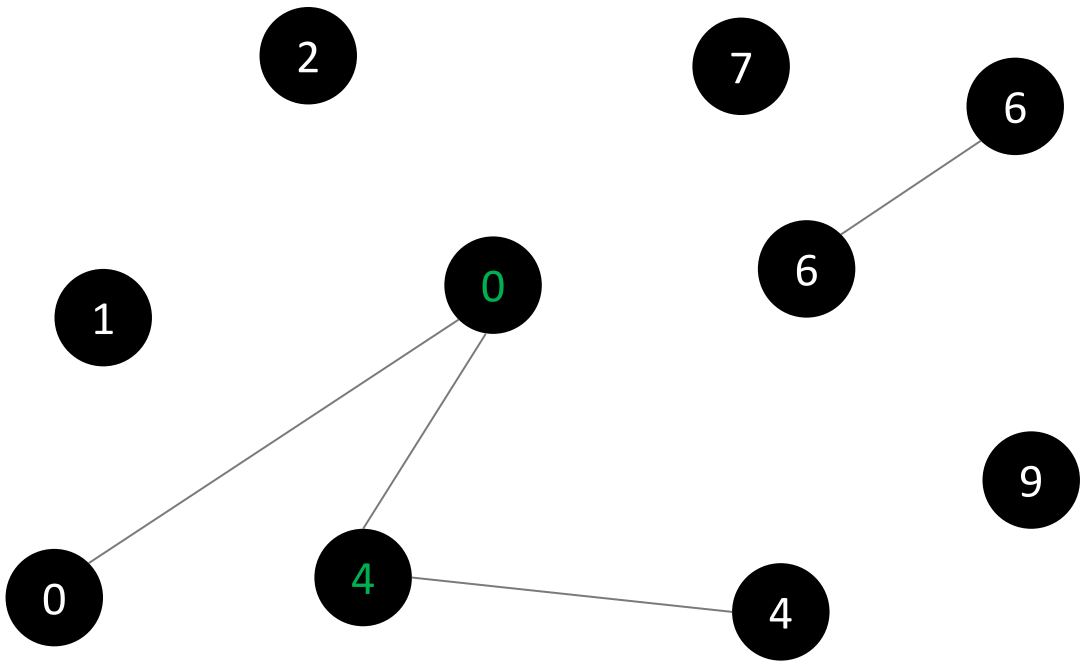
Intuitively, Kruskal’s algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal’s Algorithm



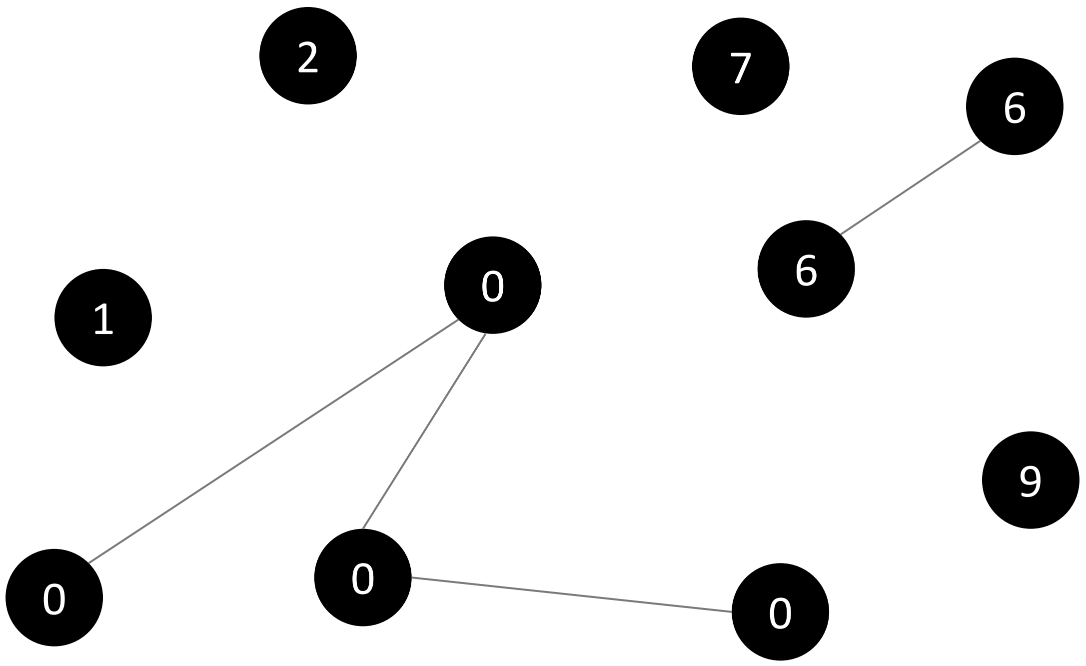
Intuitively, Kruskal’s algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal's Algorithm



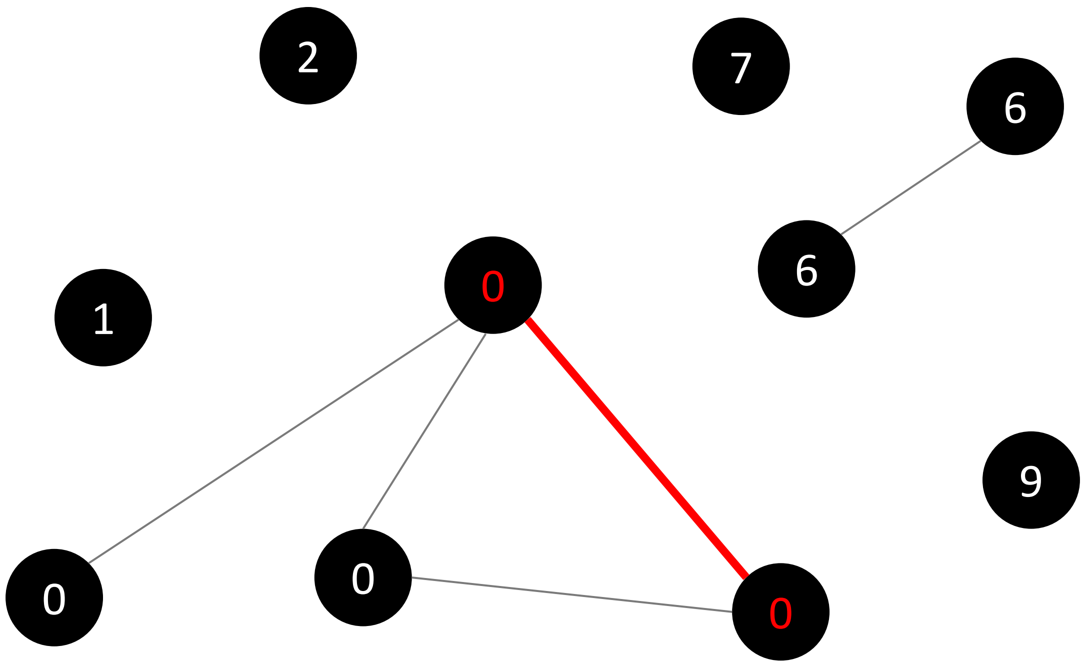
Intuitively, Kruskal's algorithm runs for V-1 iterations, and in each iteration it merges the two distinct connected components that have the smallest distance between themselves.

Kruskal’s Algorithm



We mustn’t add an edge that connects tow vertices in the same component. This is the only way a cycle could be generated.

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
```

We use a special data structure that allows us to keep track of which connected component a node belongs to and supports merging them efficiently

Union-Find Data Structure

- Define two operations: **find(u)** and **union(u,v)**

- **Find(u):** Given a vertex u, return its set ID

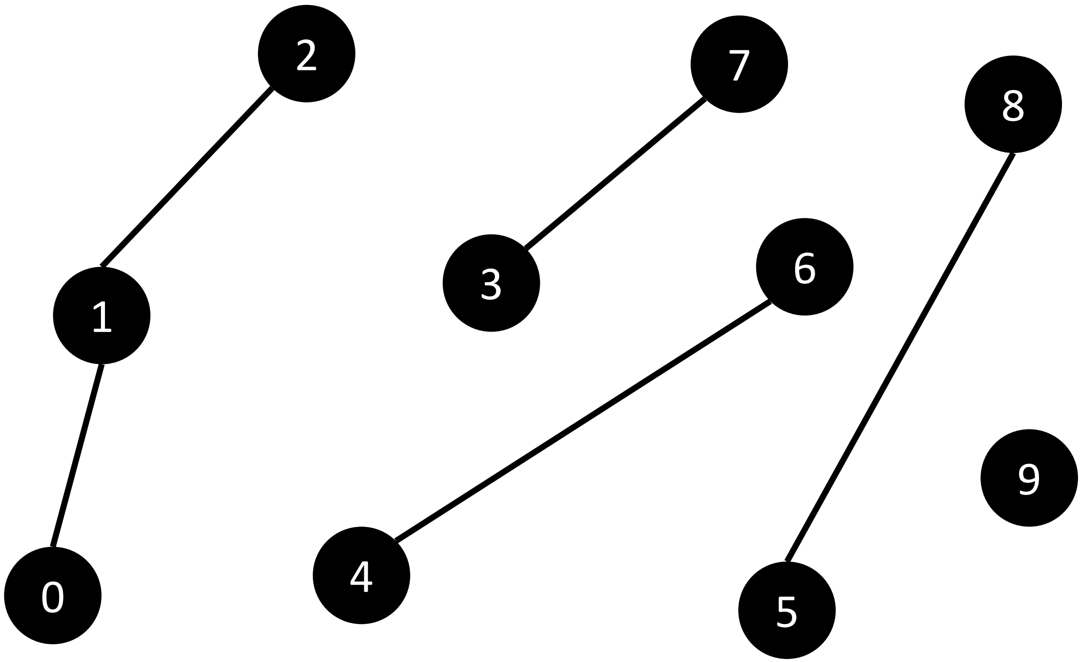
- **Union(u,v):** Given two vertices u and v, if they have different set IDs, union the two sets they belong to (and update all the set IDs of the vertices in one of the sets)

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

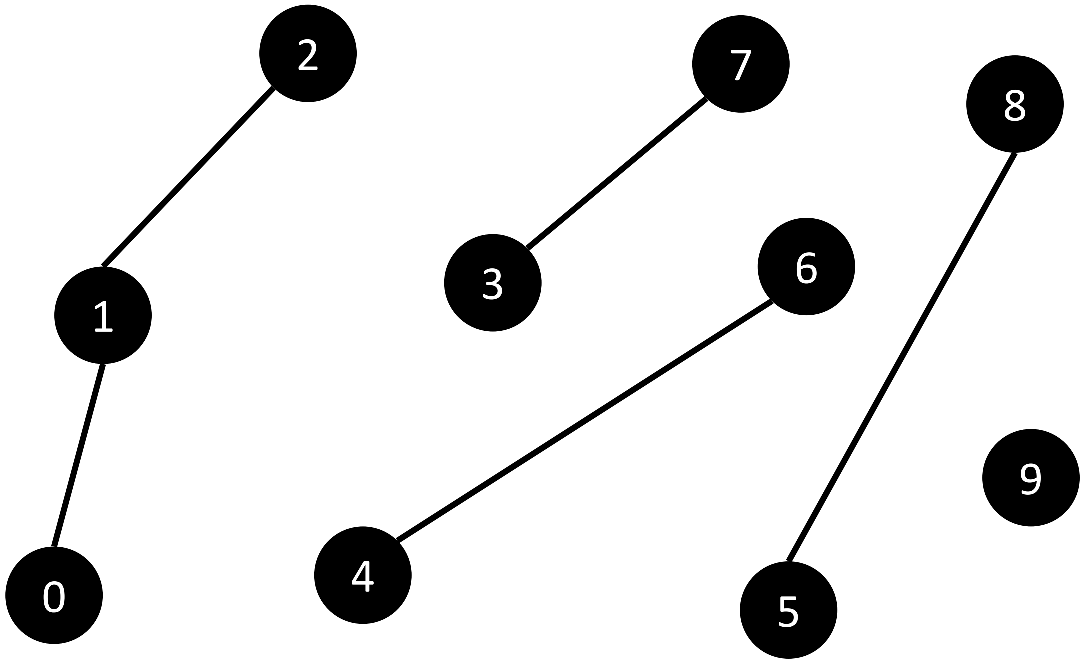
Union-Find Data Structure



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

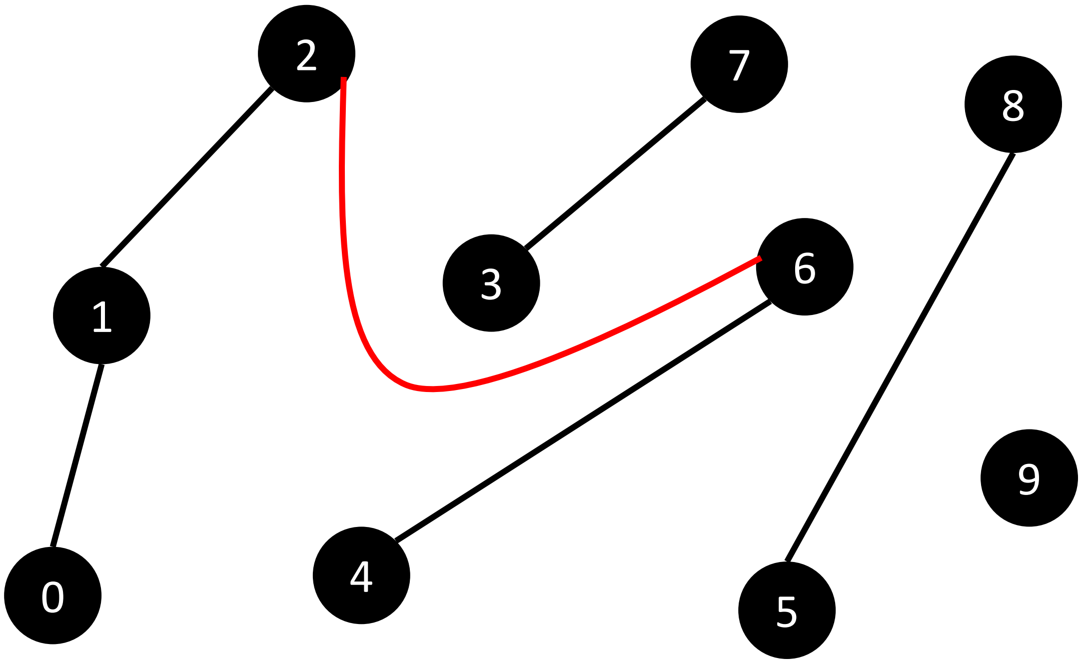
Union(2, 6)



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

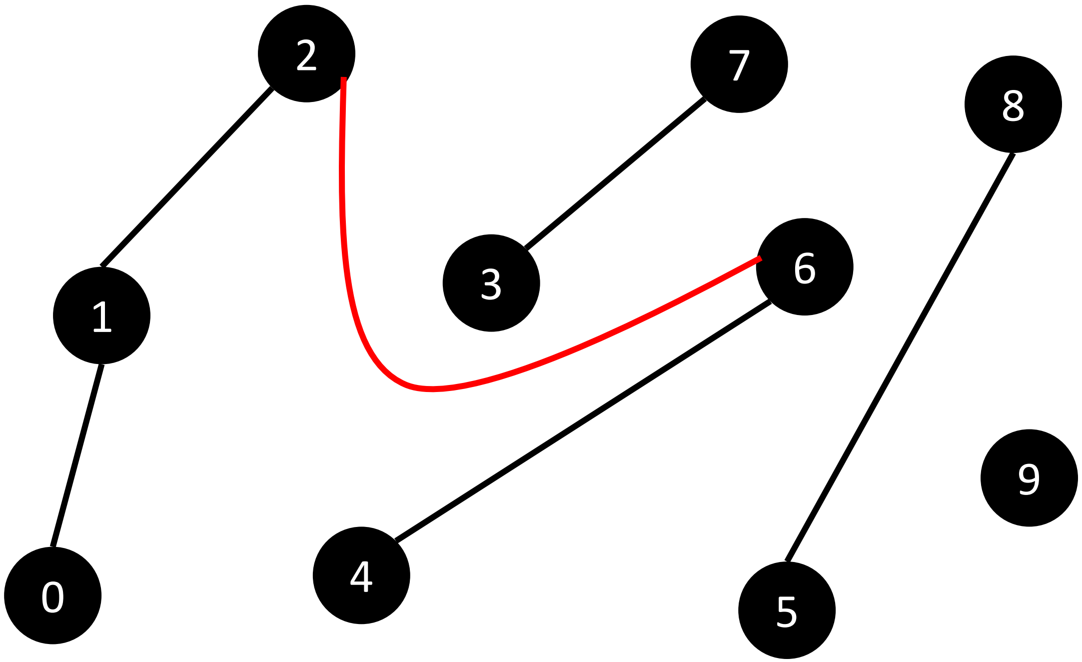
Union(2, 6)



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)

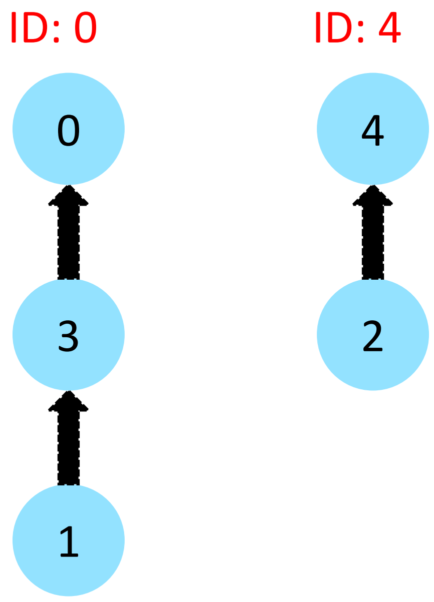


Vertex ID	0	1	2	3	4	5	6	7	8	9
Set ID	0	0	0	7	6	5	6	7	5	9

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

Union-Find Data Structure

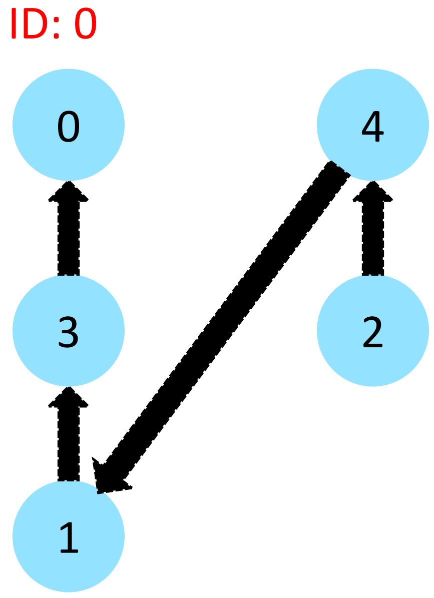
- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)
- Use one of the node IDs as the set ID (this is the head)



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

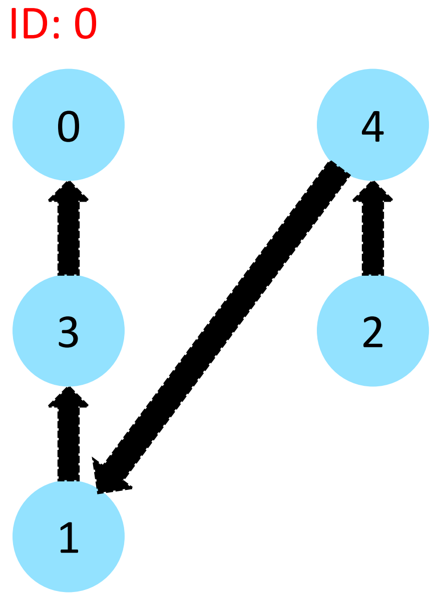
Union-Find Data Structure

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



Union-Find Data Structure

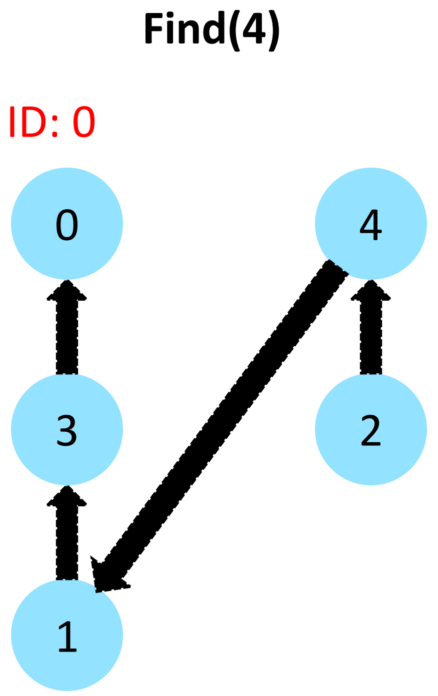
- We want to union faster
- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?



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?

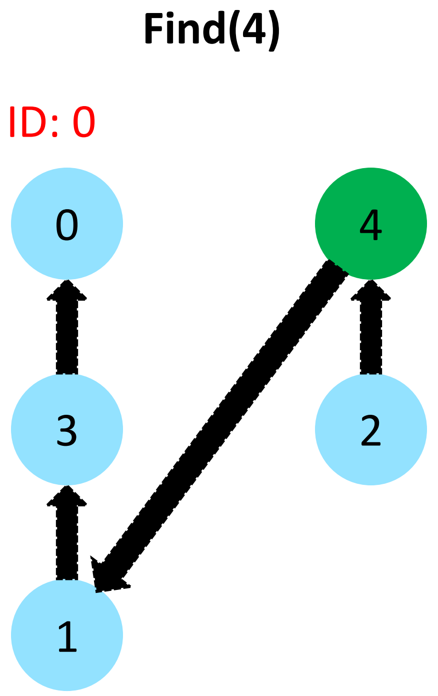
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



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?

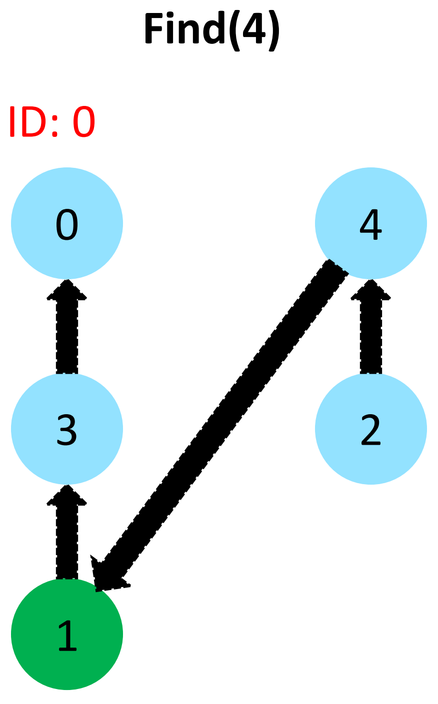
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



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?

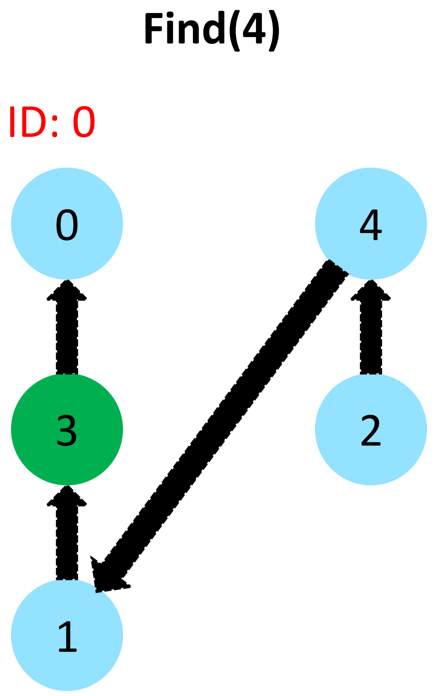
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



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?

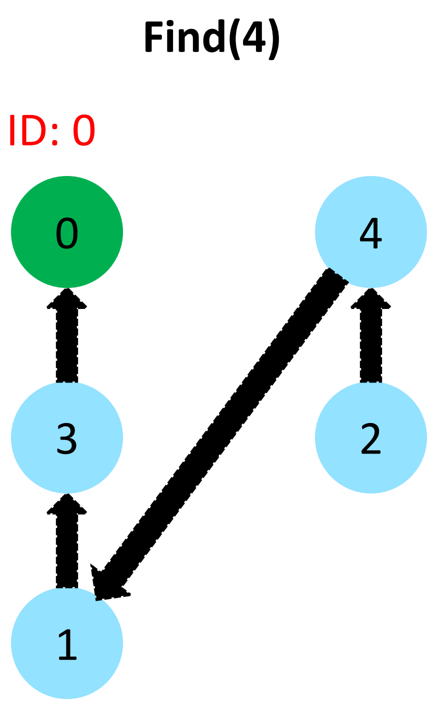
- Heads know their set ID
- Traverse to the head to find the ID of an element in the list
- This is O(size of the linked list)



Union-Find Data Structure

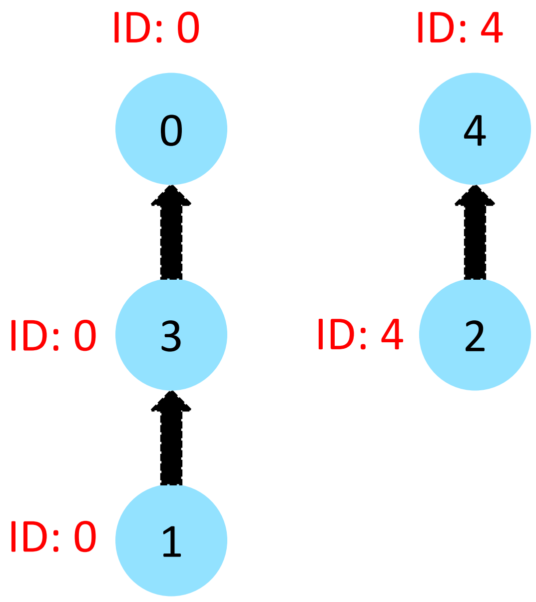
- Linked lists are fast to union (i.e. append one linked list to another)
- How do we find, with linked lists?

- 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)



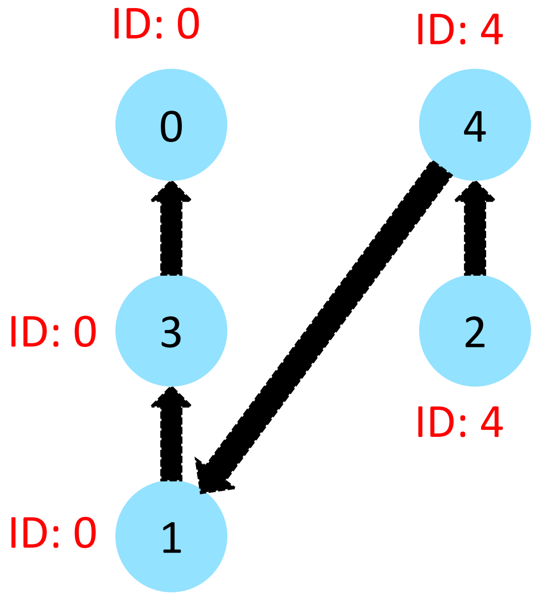
Union-Find Data Structure

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



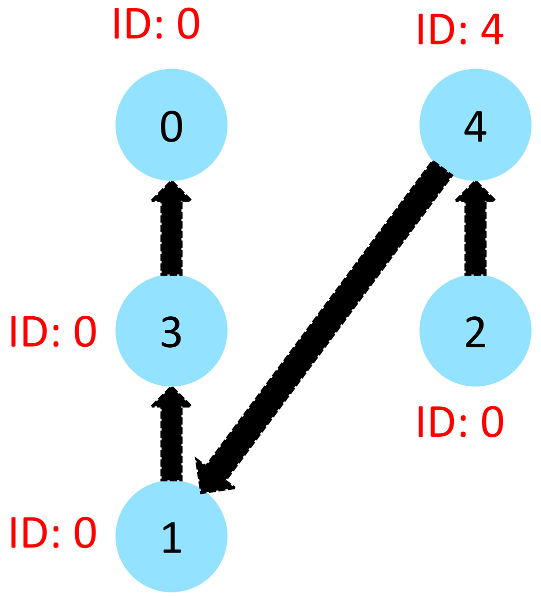
Union-Find Data Structure

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



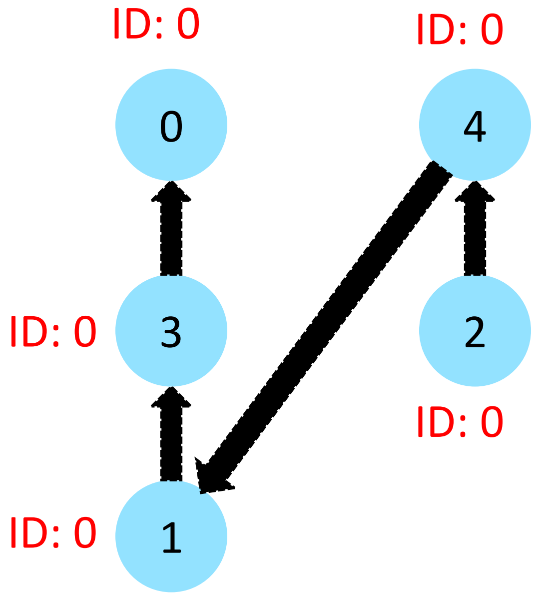
Union-Find Data Structure

- 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

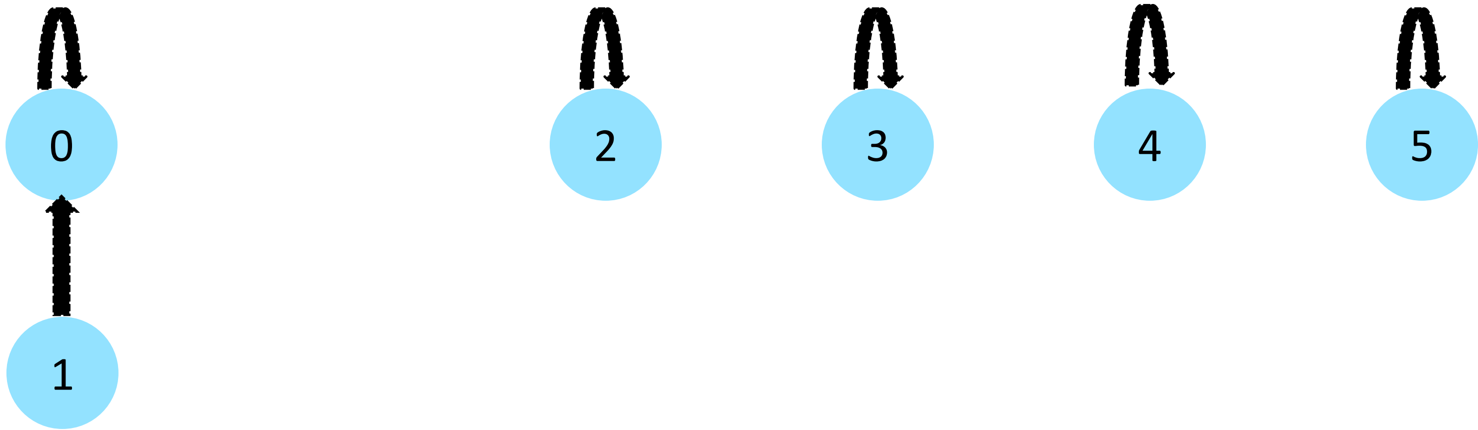
Union-Find Data Structure

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

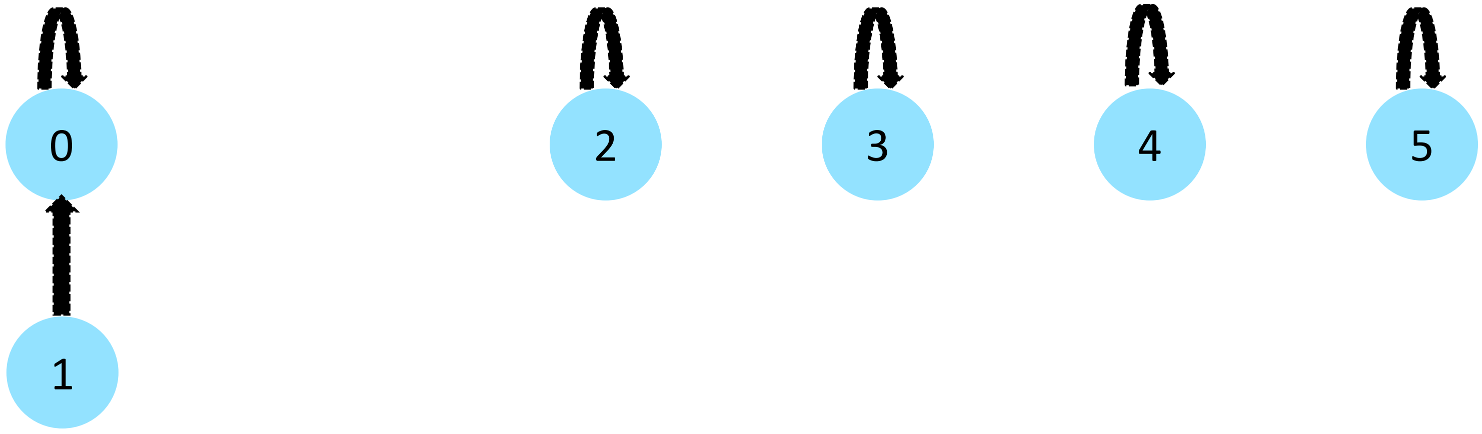
Operations:
Union(0,1)



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

Union-Find Data Structure

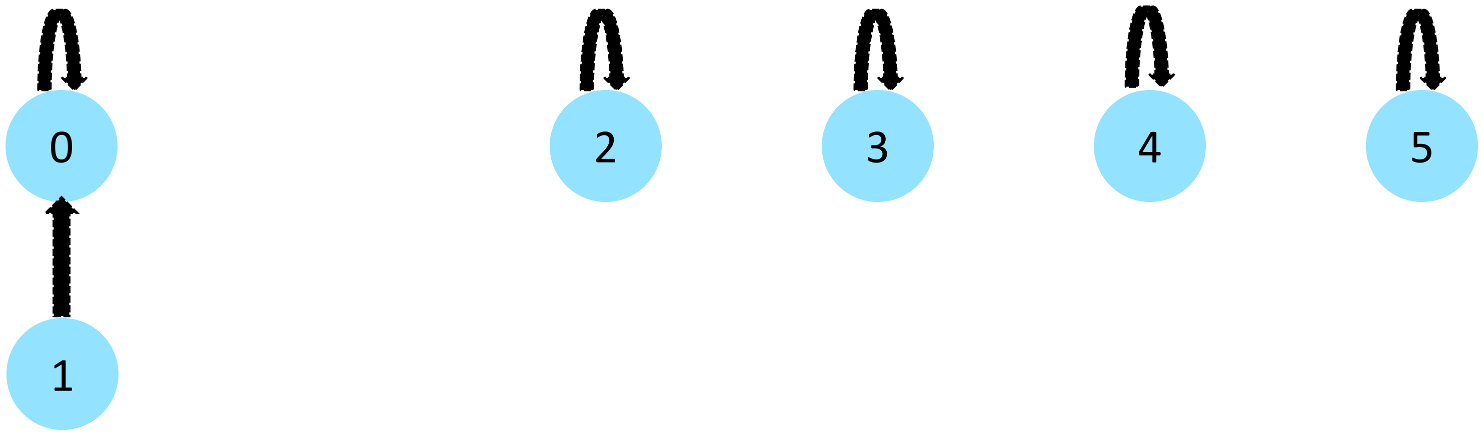
Operations:
Union(0,1)
Union(2,3)



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

Union-Find Data Structure

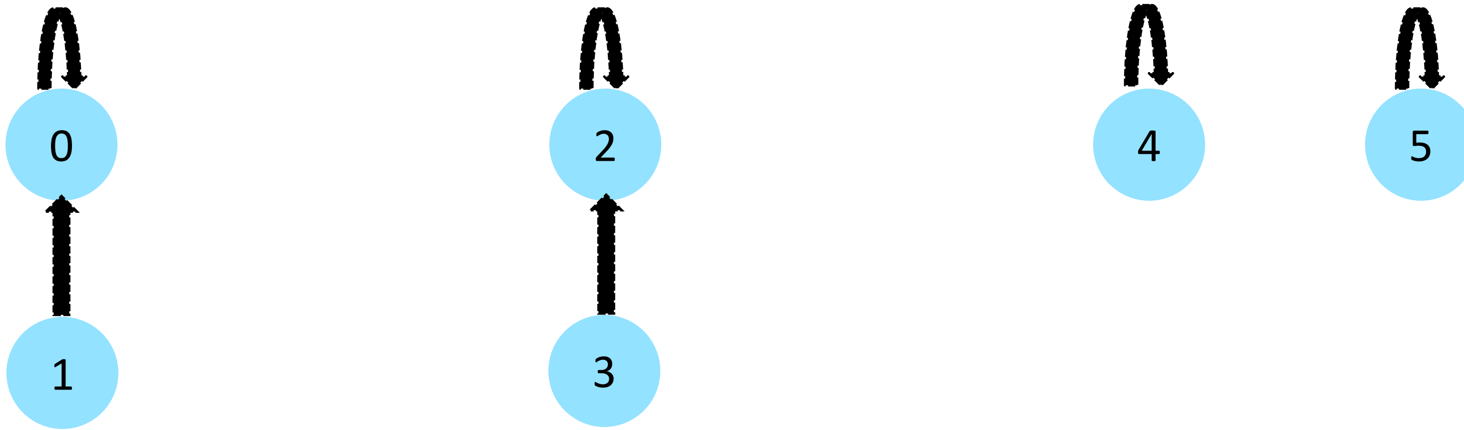
Operations:
Union(0,1)
Union(2,3)
Find(2) = 2
Find(3) = 3



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

Union-Find Data Structure

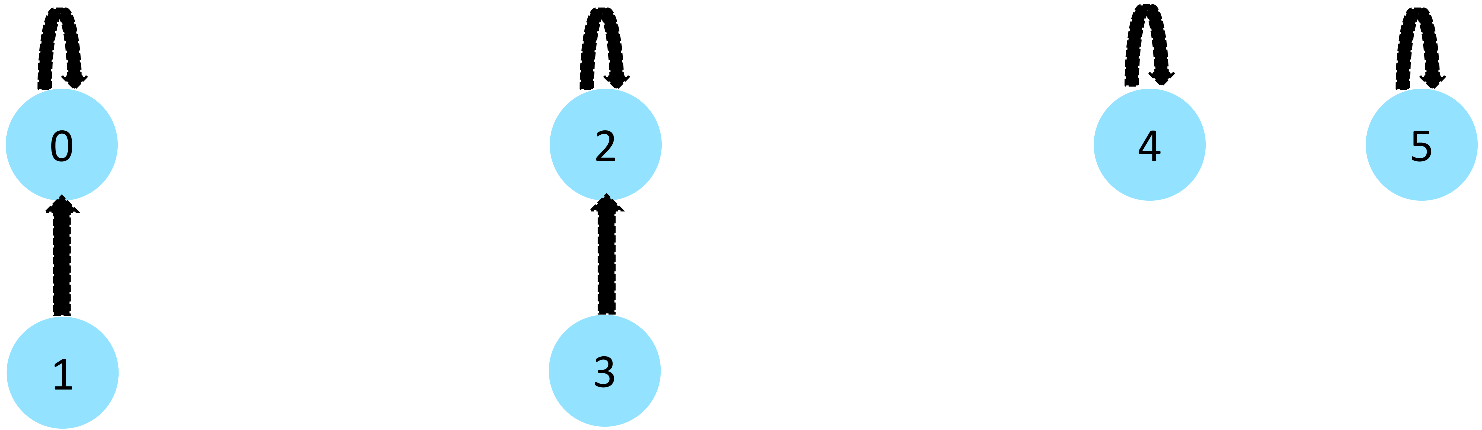
Operations:
Union(0,1)
Union(2,3)



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

Union-Find Data Structure

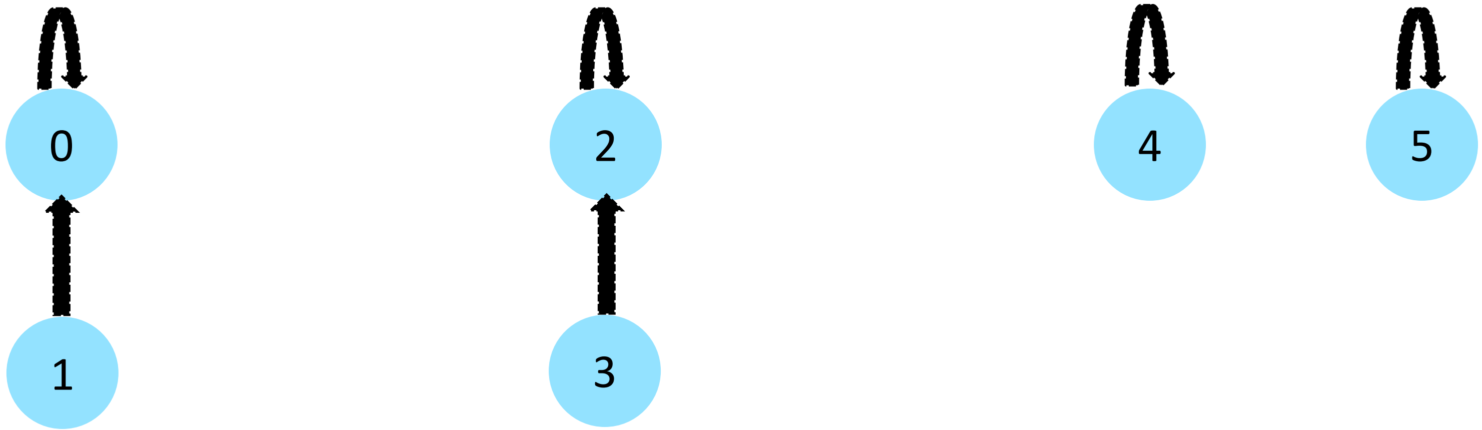
Operations:
Union(0,1)
Union(2,3)
Find(3)



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

Union-Find Data Structure

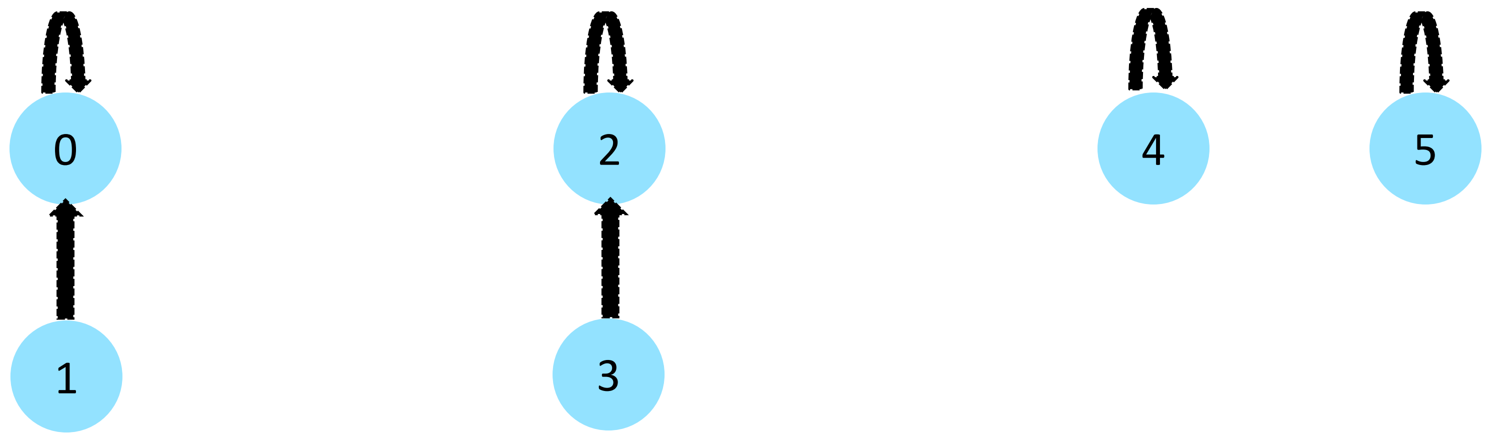
Operations:
Union(0,1)
Union(2,3)
Find(3)



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

Union-Find Data Structure

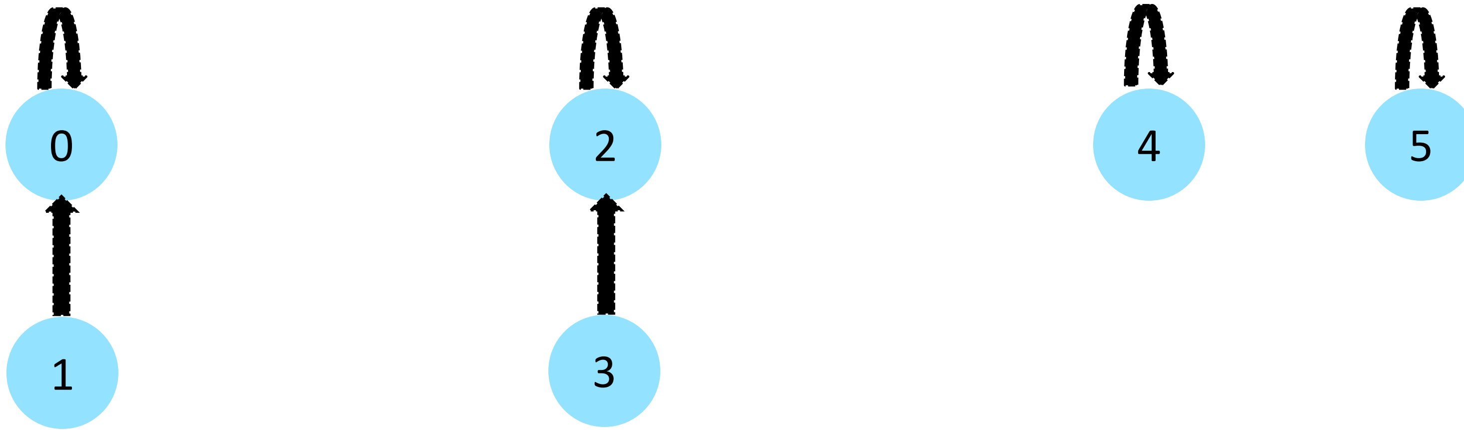
Operations:
Union(0,1)
Union(2,3)
Find(3)



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

Union-Find Data Structure

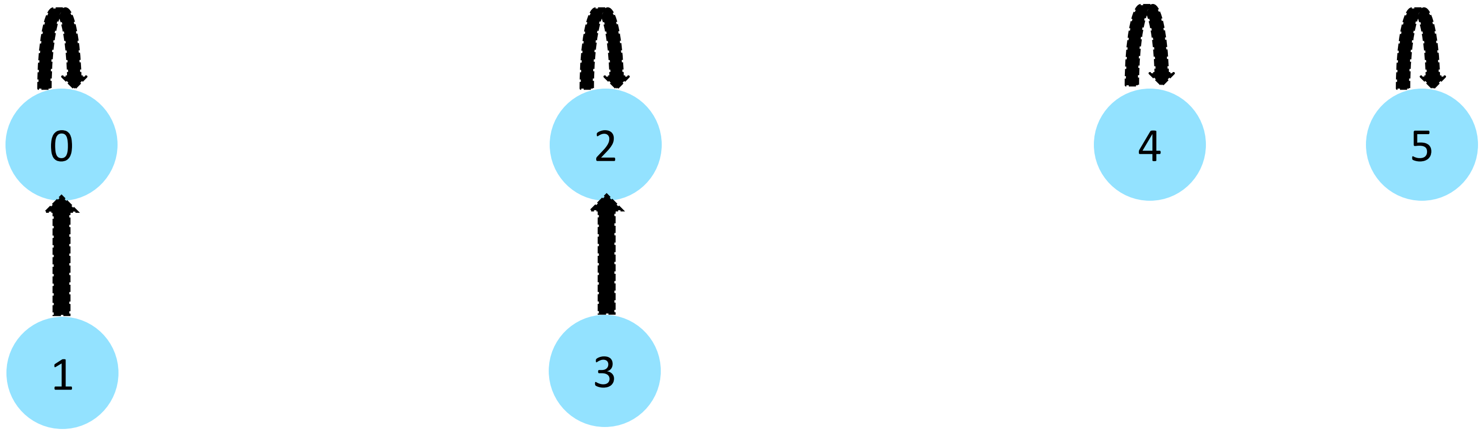
Operations:
Union(0,1)
Union(2,3)
Find(3)



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

Union-Find Data Structure

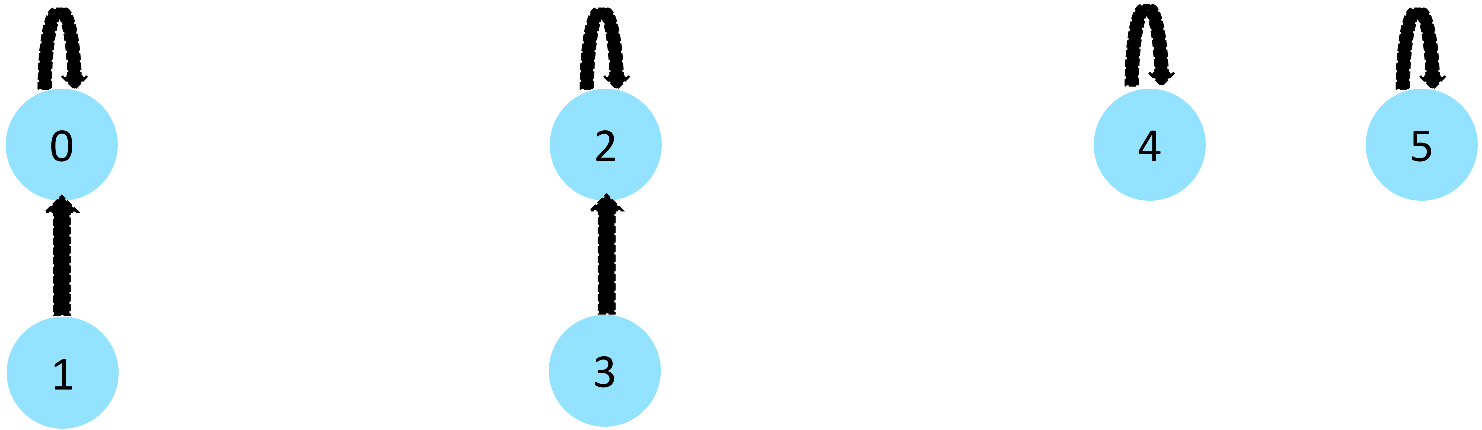
Operations:
Union(0,1)
Union(2,3)
Find(3)=2



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)

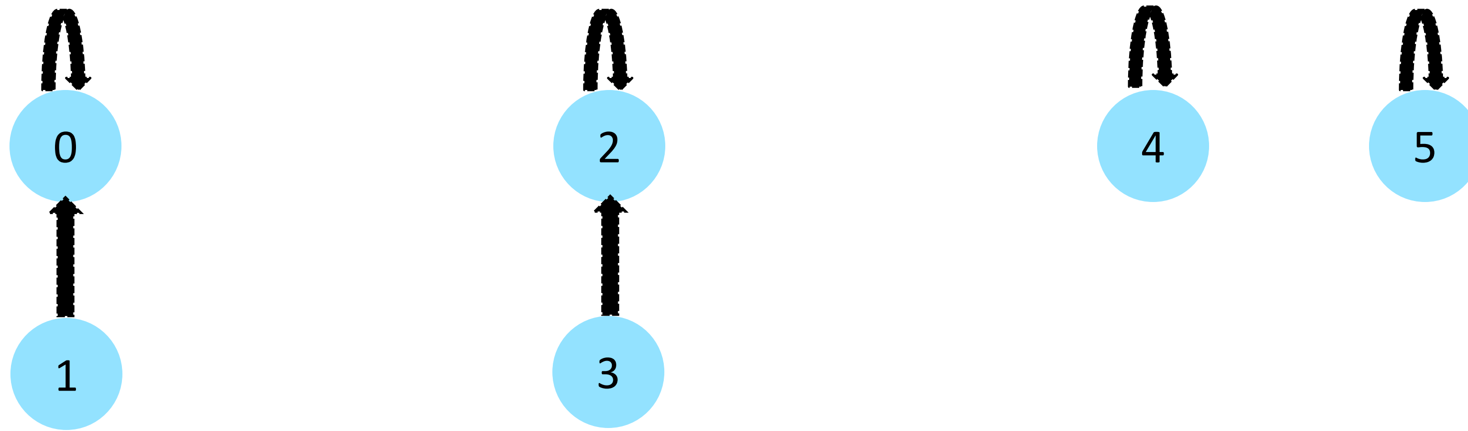
Union(2,3)

Find(3)=2

Union(0,1)

  Find(0) = 0

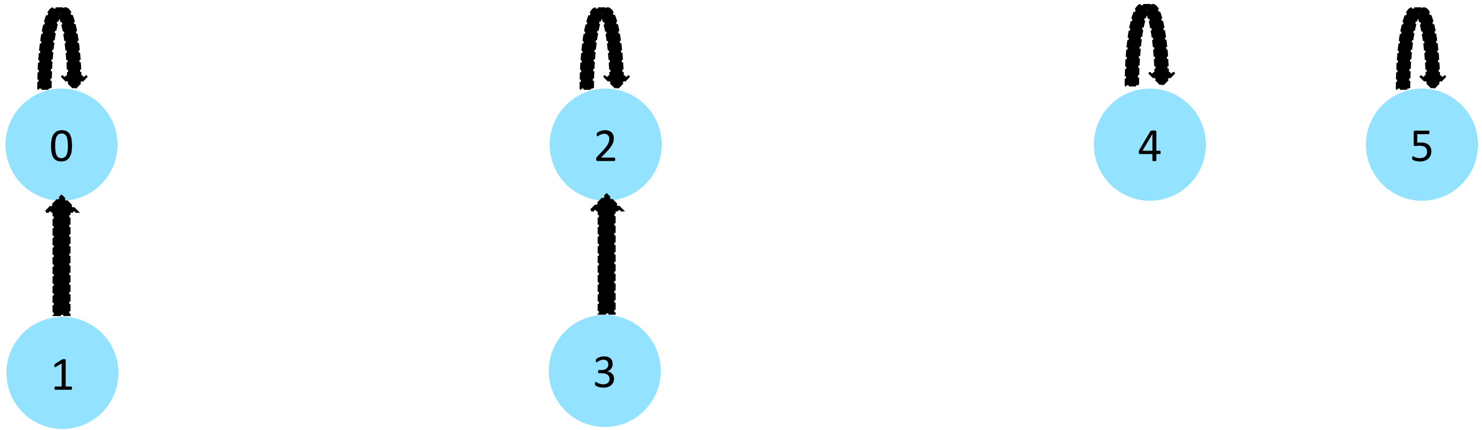
  Find(1) = 0



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

Union-Find Data Structure

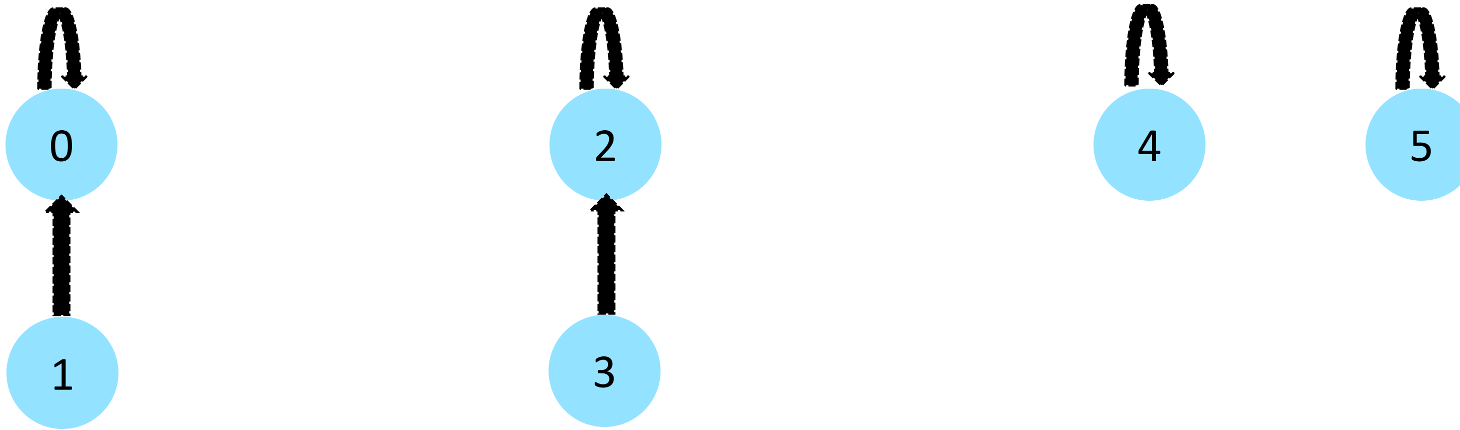
Operations:
Union(0,1)
Union(2,3)
Find(3)=2
Union(0,1)
Union(1,3)



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)

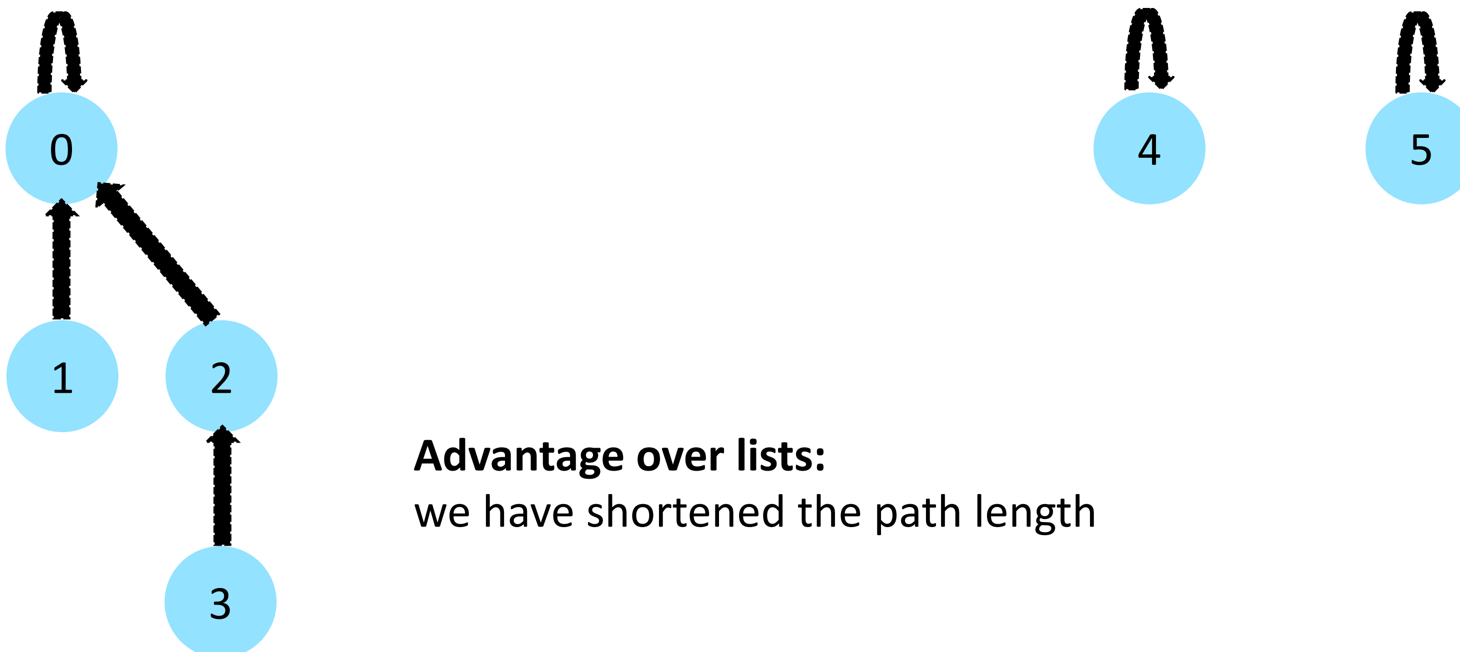
Find(3)=2

Union(0,1)

Union(1,3)

  Find(1) = 0

  Find(3) = 2

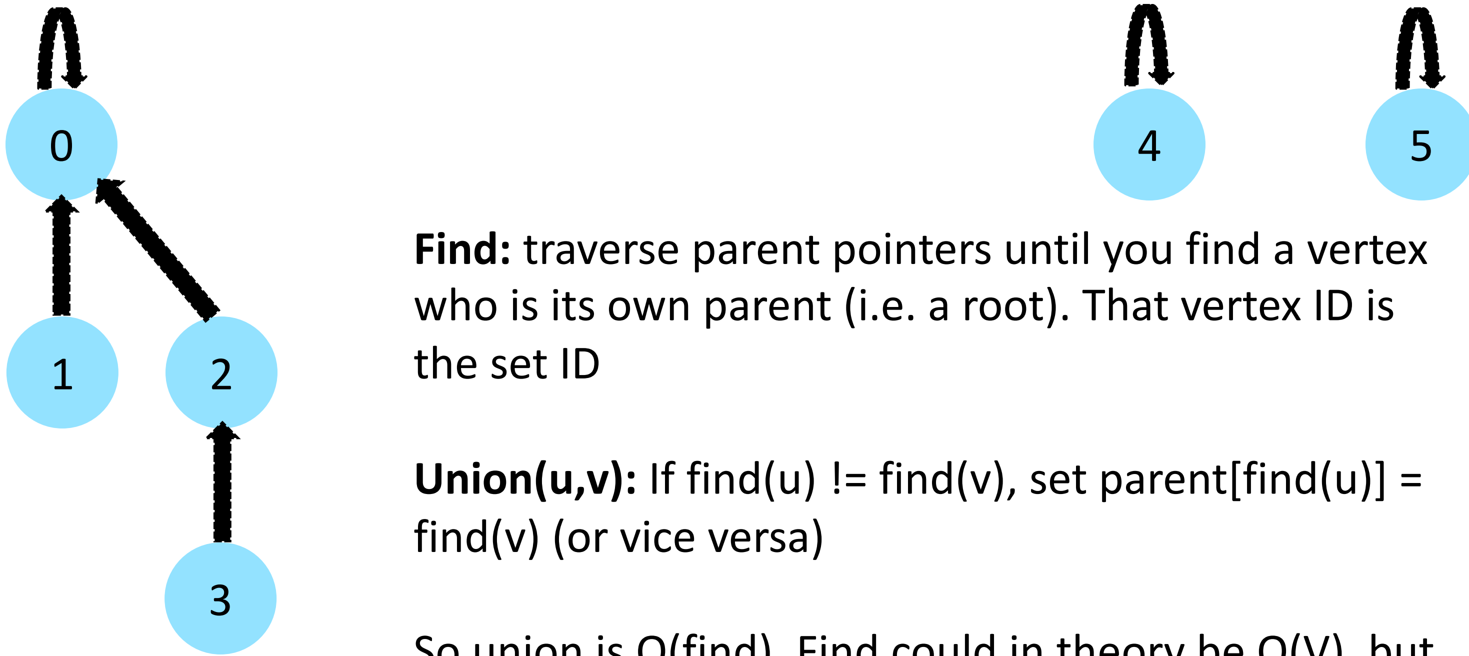


Advantage over lists:
we have shortened the path length

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

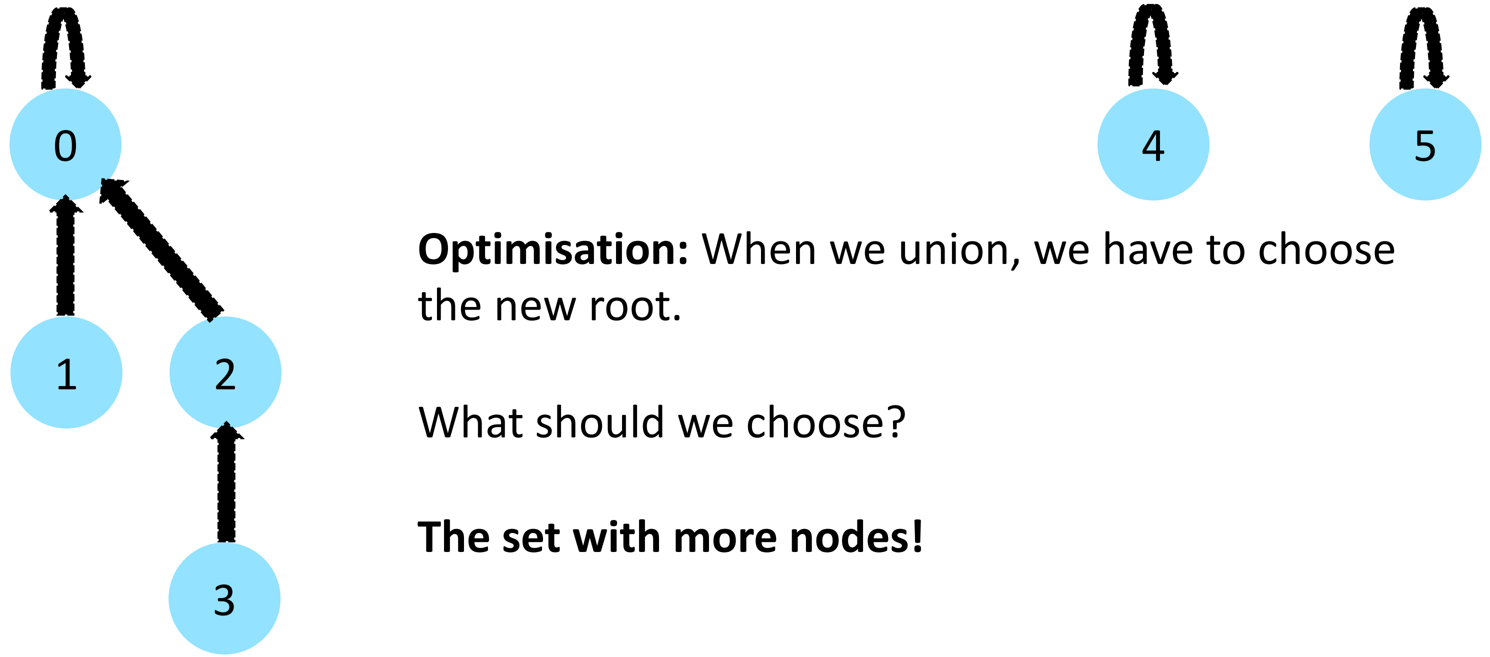
Union(u,v): If find(u) != find(v), set parent[find(u)] = find(v) (or vice versa)

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 to choose the new root.

What should we choose?

The set with more 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)

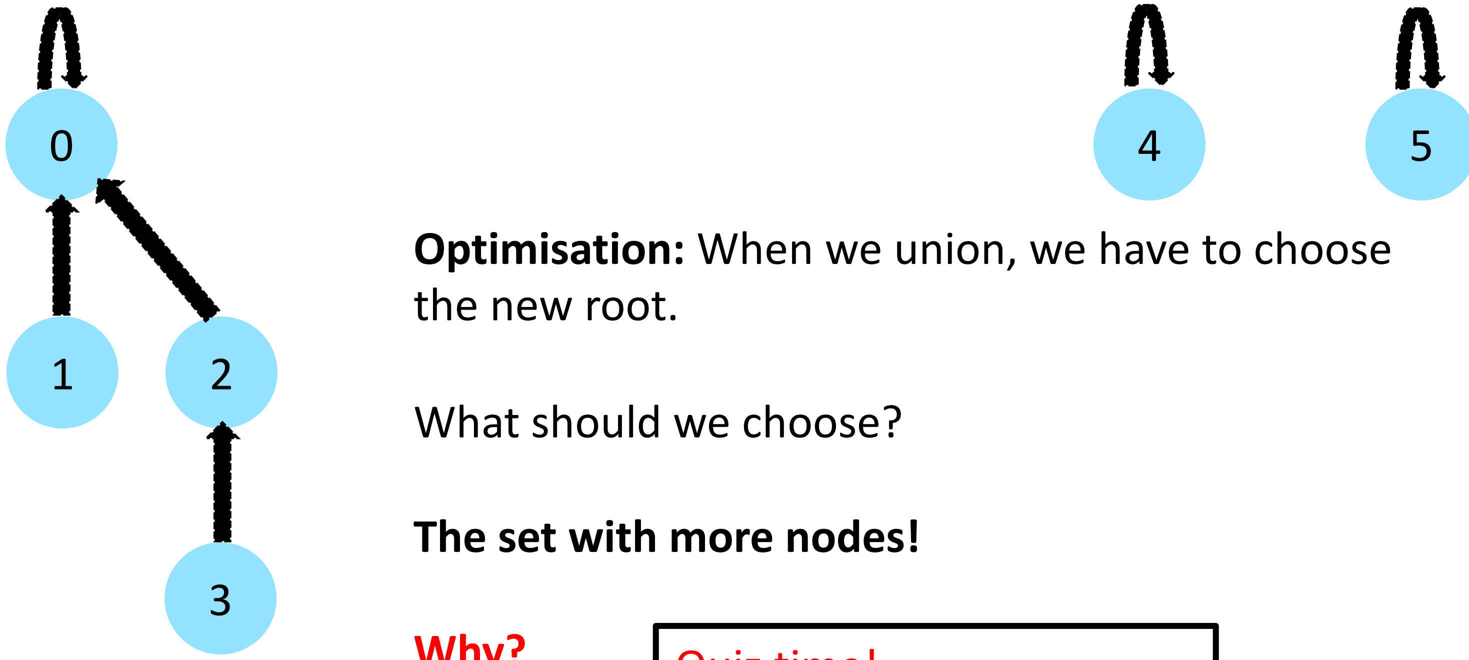
Find(3)=2

Union(0,1)

Union(1,3)

Find(1) = 0

Find(3) = 2



Optimisation: When we union, we have to choose the new root.

What should we choose?

The set with more nodes!

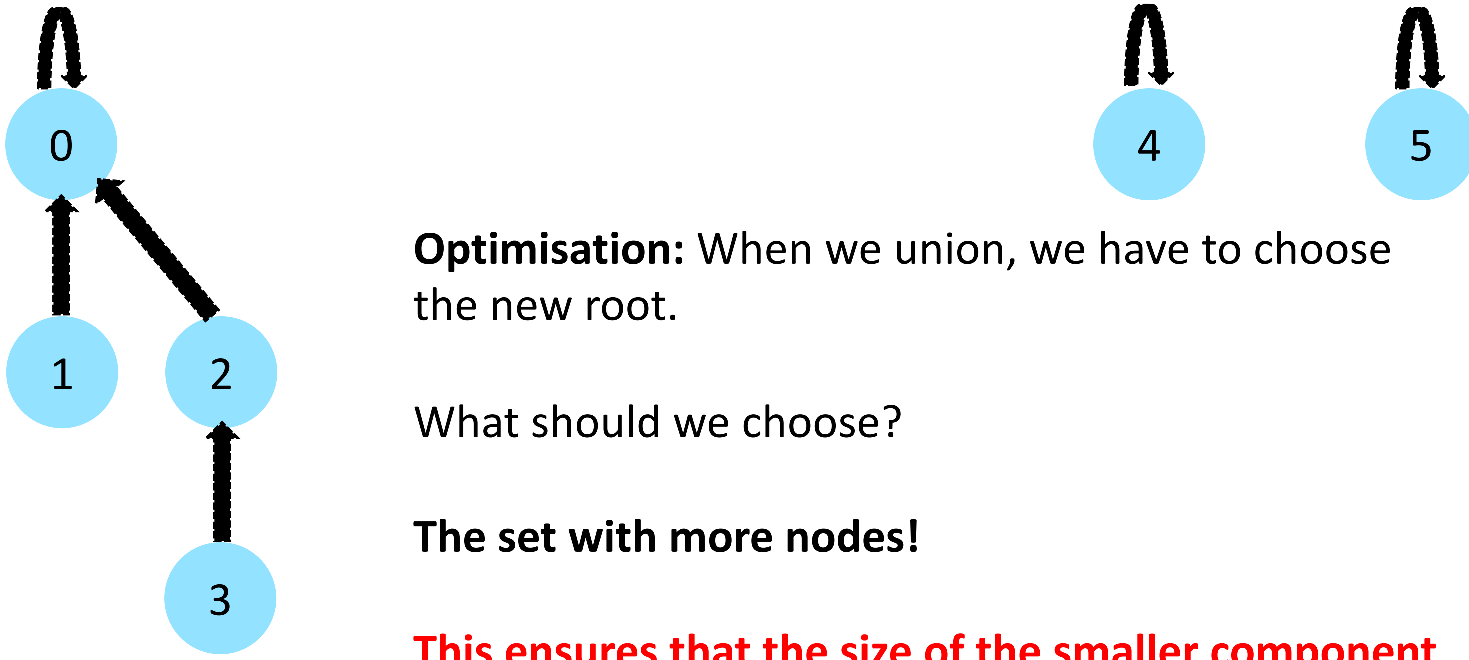
Why?

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

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 to choose the new root.

What should we choose?

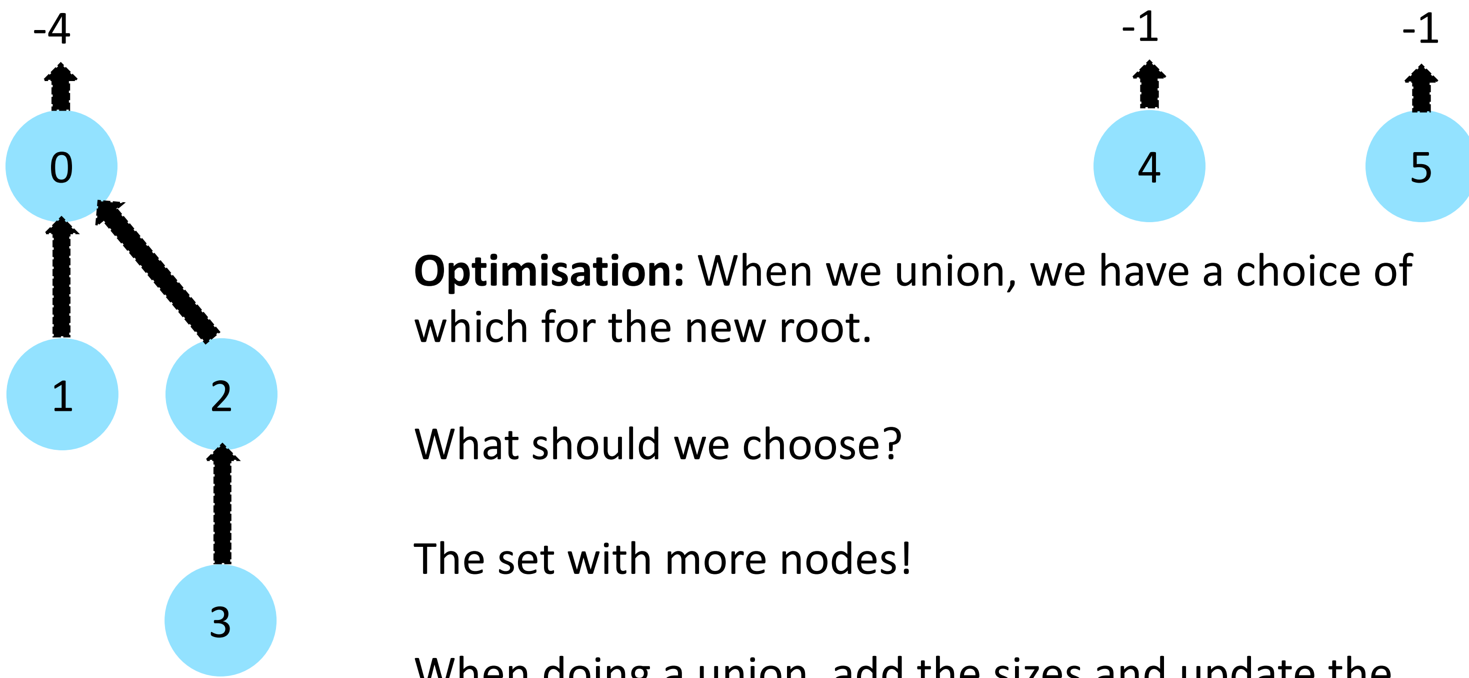
The set with more nodes!

This ensures that the size of the smaller component is at least doubled when we merge and this is the one that grows in 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 more 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 sizes (negative to distinguish from parent ID)

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
```

Time Complexity:

- Initialization of union-find: 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
  - FIND() takes O(x) where x is height of the tree
  - UNION() 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) assuming find is O(V)

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)

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
- **This is because we have ensured that the smallest component size at least doubles when its height grows**
- 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 size of the taller tree being unioned.

- We need to do V-1 unions.
- Each one has cost 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(V logV).

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
```

Time Complexity:

- Initialization of union-find: 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,
  - total for find: O(E log V)
  - 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 $O(V \log(V))$ to $O(V\,\alpha(V))$, where α denotes the inverse Ackermann function, an **extremely** slow growing function.

The Inverse Ackerman Function

- $\alpha(x)$ is the inverse Ackerman function or the "Inverse Tower of Two"
- Also often called log*(x)

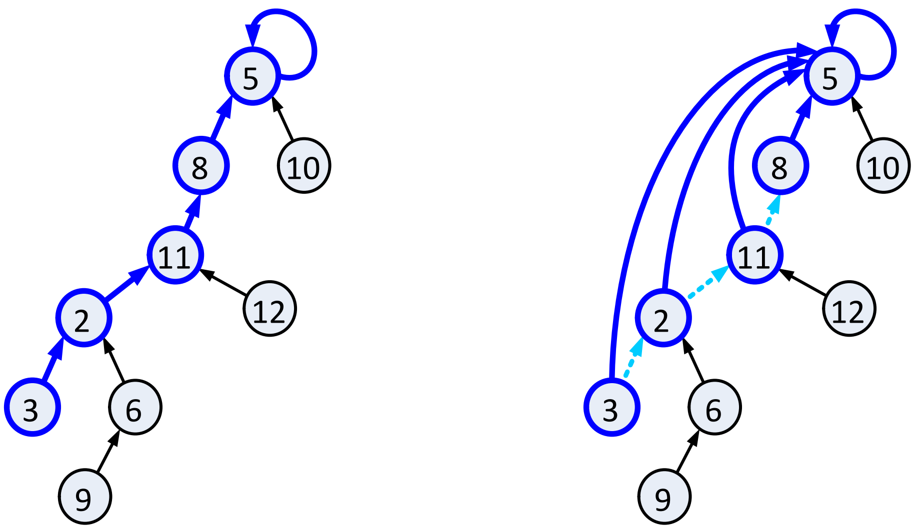
- It grows amazingly slowly

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

n	2	2^2 =4	2^{2^2} =16	$2^{2^{2^2}}$ =65536	$2^{2^{2^{2^2}}}$
log* n	1	2	3	4	5

Path Compression

- Path compression:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root



- Implies $O(n \log^* n)$ time for performing n union-find operations:
  - Proof is complex… (in Weiss 8.6.1, Theorem 8.1)

Kruskal's Algorithm: Correctness

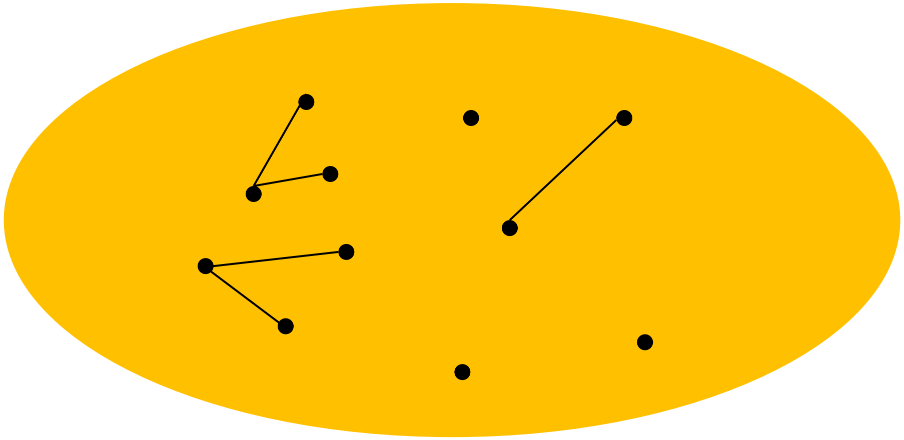
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G.

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

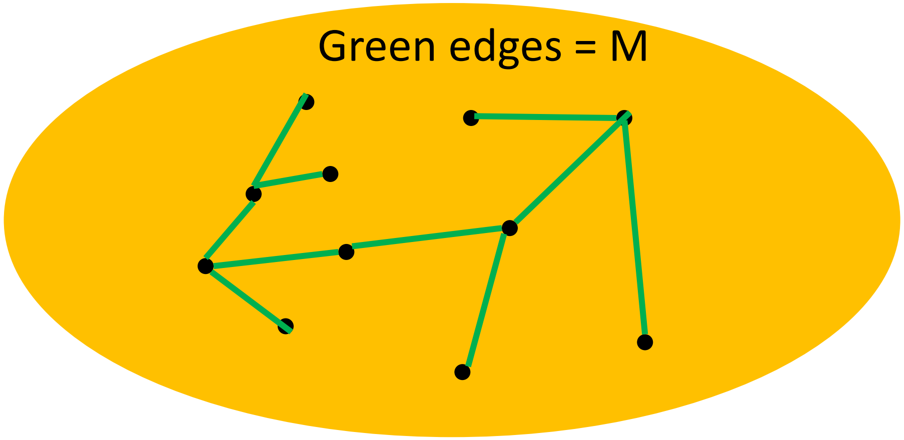
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G.

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

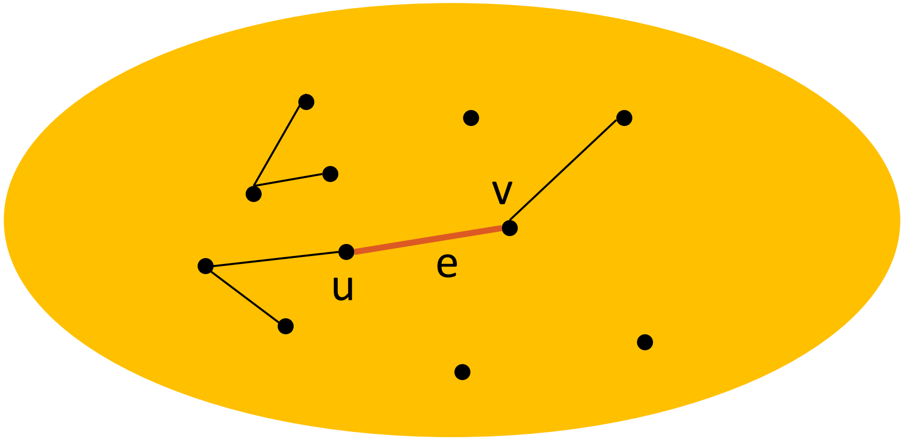
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G.

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 connected 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

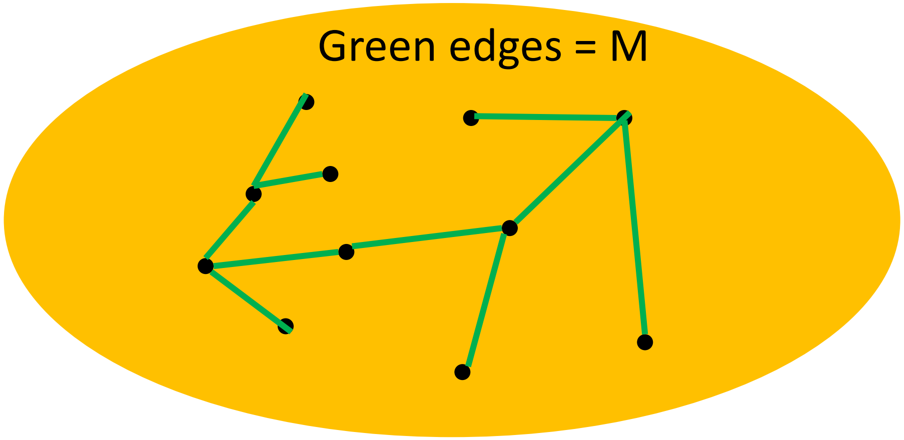
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G.

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

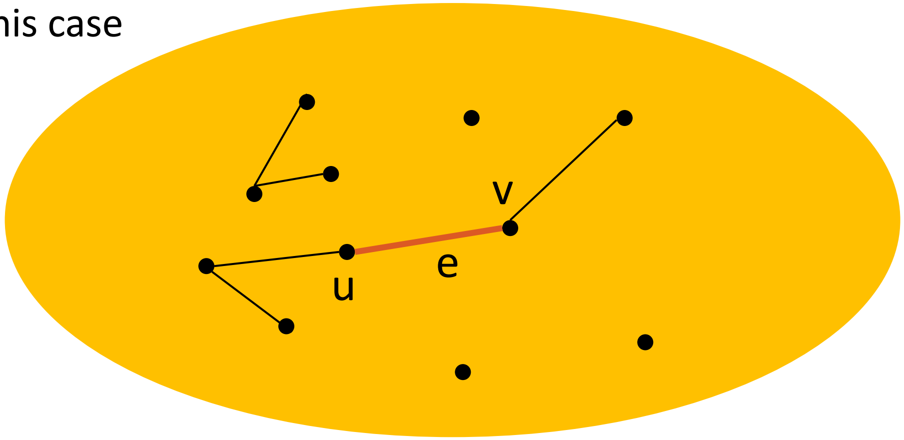
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G.

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

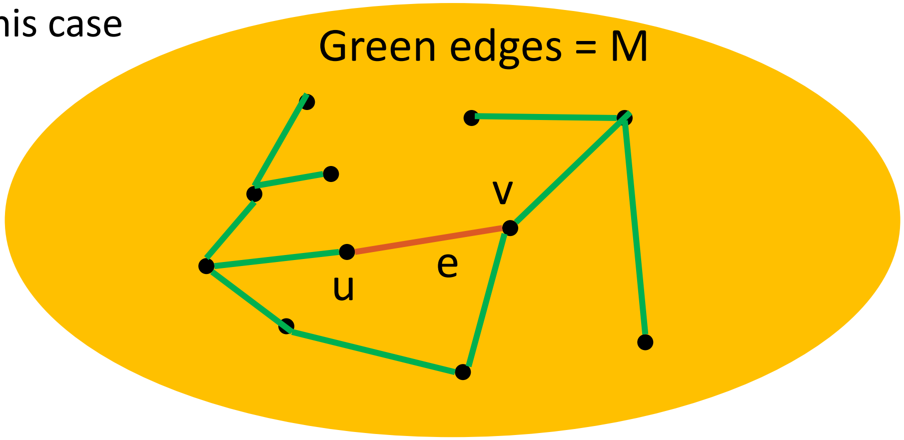
#INV: Every iteration of Kruskal's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G

Base Case:

- The invariant is true initially when T is empty

Inductive step:

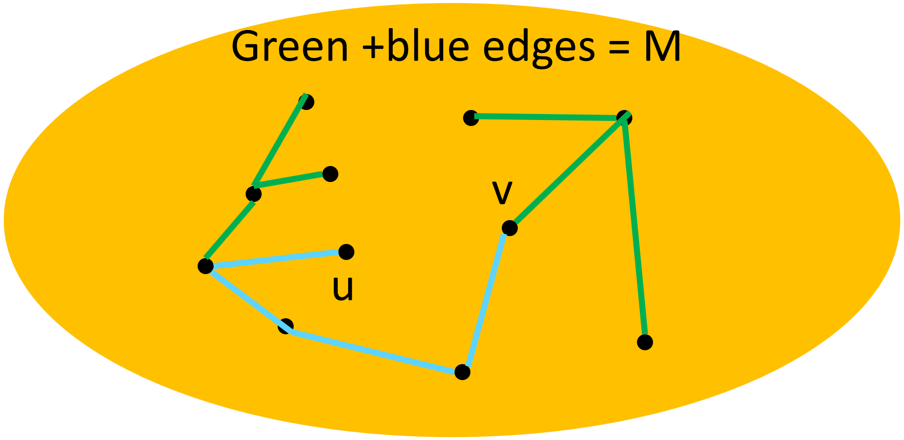
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration**
- Assume **T** is a subset of some MST **M**
- Let **e = (u,v)** be the lightest edge that connects two vertices in different components of **T** (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}**.



Kruskal's Algorithm: Correctness

Inductive step:

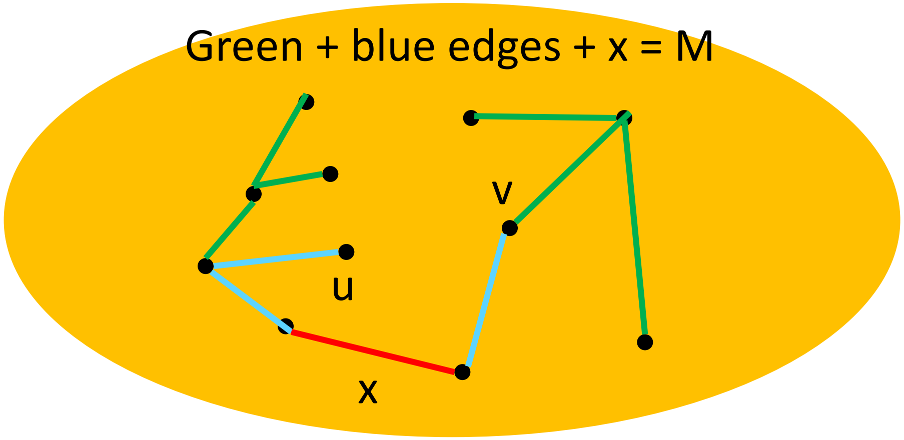
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration.**
- Since **M** is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue).
- **u** and **v** are not connected in **T** (since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

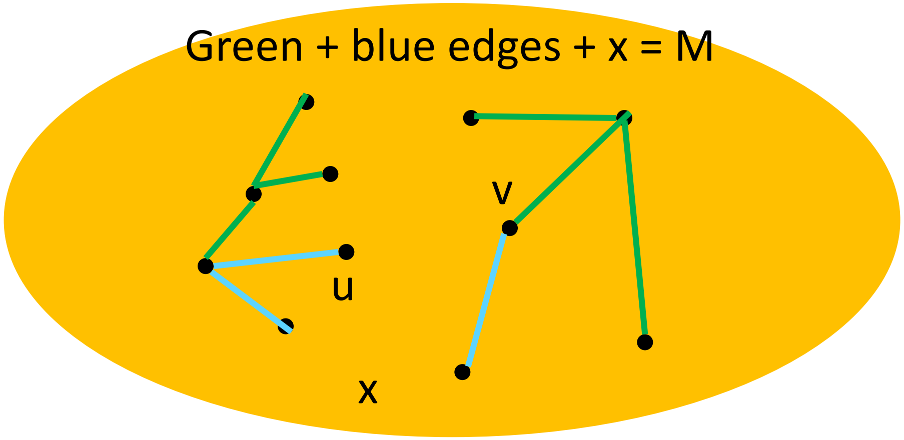
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration.**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue).
- **u** and **v** are not connected in **T** (since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

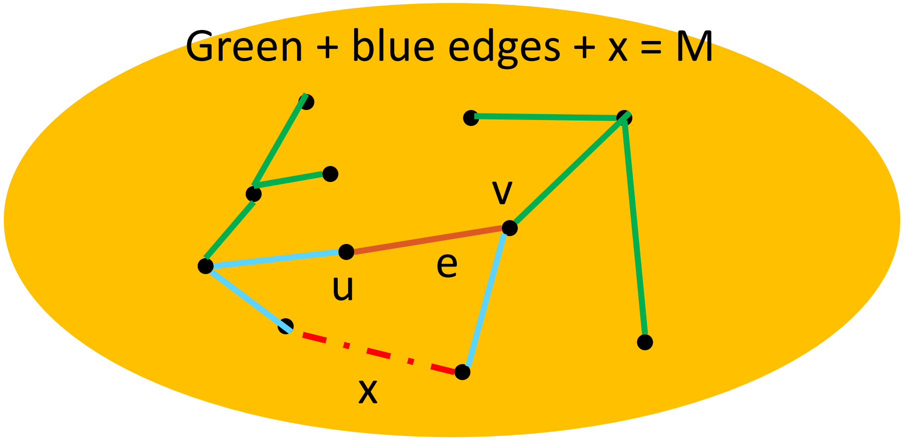
- **We want to show that, if T is a subset of some MST at the start of some iteration, it is still a subset of some MST at the start of the next iteration.**
- Since M is a tree, there is exactly one path from **u** to **v** in **M** (shown in blue).
- **u** and **v** are not connected in **T** (since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



Other MCST algorithms

- Reverse-delete Algorithm
  - The exact inverse of Kruskal's algorithm
  - O(E log V)
- Barouvka's Algorithm (1926!)
  - Grows a forest, like Kruskal's algorithm
  - In each iteration, finds the cheapest edge for each connected component that connects it to another connected component
  - O(E log V)

These are not examinable

Summary

Take home message

- Prim's algorithm and Kruskal's algorithm both are greedy algorithm that correctly determine minimum spanning trees.
- While Prim's algorithm keeps growing the same connected component at each iteration; Kruskal's algorithm merges the two connected components that are closest to each other at each iteration.
- Both algorithms run in O(E log V)

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.

Coming Up Next

- Network Flow