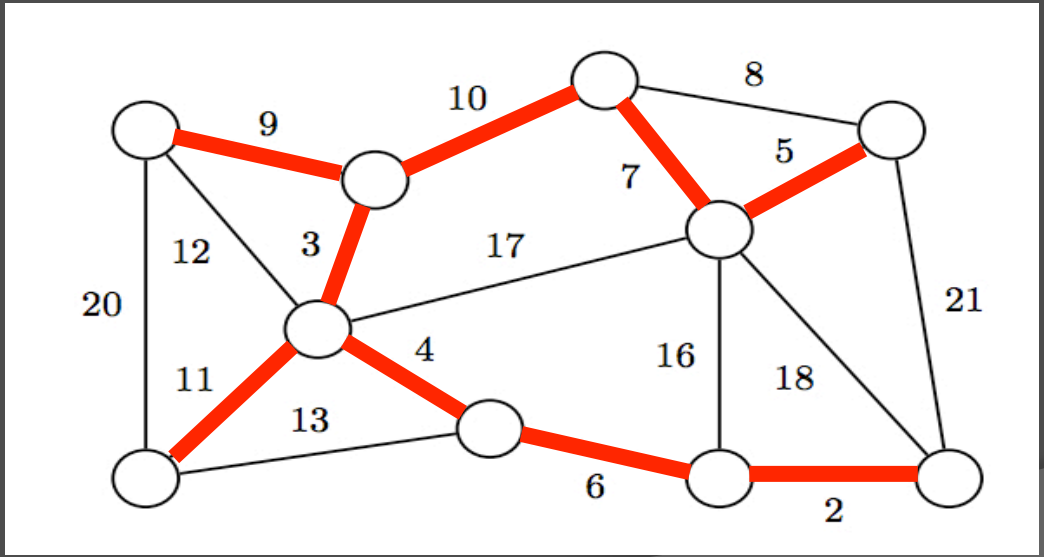
Advanced Computer Contest Preparation
Lecture 9

KRUSKAL'S ALGORITHM DISJOINT SETS

Recall: Minimum Spanning Tree

- Given a graph, the minimum spanning tree (MST) is a set of edges where all nodes are connected, and the total cost is minimum.



Recall: Minimum Spanning Tree

- We know Prim's Algorithm to compute the MST
- Code is very similar to Dijkstra's Algorithm
- Prim's is not the only MST algorithm

Kruskal's Algorithm

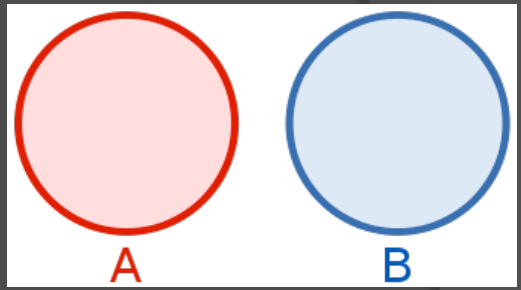
- Greedily computes the MST of a graph
- Sort all of the edges by increasing cost
- Iterate through sorted edges and add an edge to the MST if the two nodes it connects are not already connected (no path between them)
- Stop when there are $V-1$ edges

Kruskal's Algorithm

- "Iterate through sorted edges and add an edge to the MST if the two nodes it connects are not already connected"
  - How can we quickly check if two nodes are connected or not?
- Solution: Use a Disjoint Set Data Structure

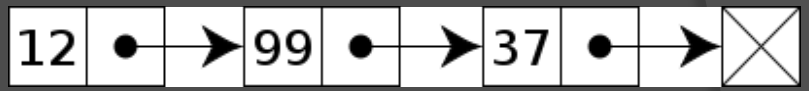
Disjoint Set Data Structure

- From mathematics: two sets are called **disjoint** if they have no element in common
- Disjoint set data structure keeps track of disjoint sets
- Supports three main operations:
  - Find: Determine which set an element is in
  - Merge/Union: Join two sets into a single one
  - Make: Form a new set with a single element
- Also called union-find data structure or merge-find set



Disjoint Set – Implementation

- The disjoint set data structure can be implemented using linked lists
- Each set is its own linked list
- Operation implementations:
  - Make: make a new linked list ($O(1)$)
  - Find: each node has a pointer to first element of its list ($O(1)$)
  - Merge: Append one linked list to the other ($O(1)$) and update first element pointer of all nodes that were appended ($O(N)$)
- Not efficient due to $O(N)$ update

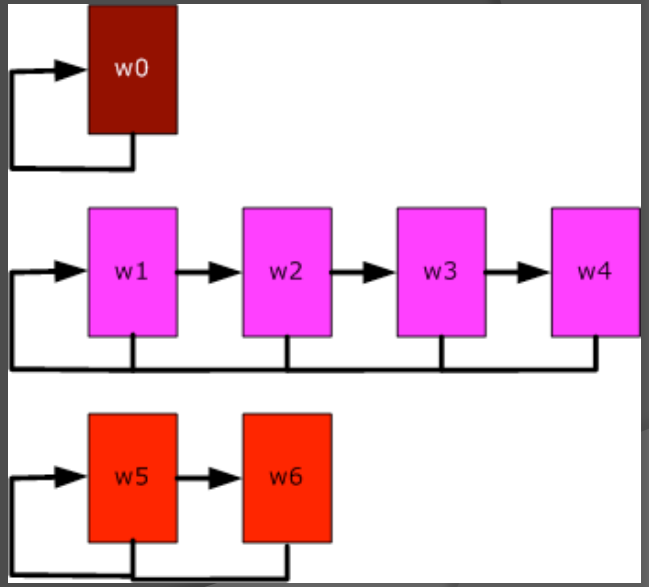


Weighted Union Heuristic

- When we are merging two sets, always merge the smaller one to the larger one
- Minimize required updates
  - The size of the smaller set cannot exceed half of the total size of the two sets
- For a disjoint set data structure using a linked list, merging all elements together using this heuristic takes $O(N \log N)$ time

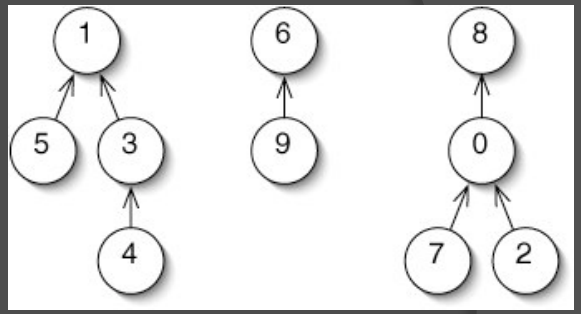
Disjoint Set Linked List

- Using a linked list implementation for a disjoint set data structure is not a bad idea
- The final time complexity across M operations (make, find, merge) is $O(M + N \log N)$
  - Weighted union heuristic must be applied in order to achieve this time complexity
- Is there a more efficient implementation?



Disjoint Set Forest

- The disjoint set data structure can be implemented using a forest (collection of trees)
- Each individual set is a tree with an arbitrary root
  - Elements in the same tree belong to the same set
- The only connection between nodes is a node's pointer to an ancestor



Disjoint Set Forest Implementation

- 2 arrays: **`lead[]`** and **`rank[]`**
- **`lead[]`** is an ancestor element of the current element
  - An element **`n`** is at the top of the set if **`lead[n]`** is equal to **`n`**
- **`rank[]`** is the number of "levels" below and including the current element
  - Not necessarily the same as height

Disjoint Set – Methods

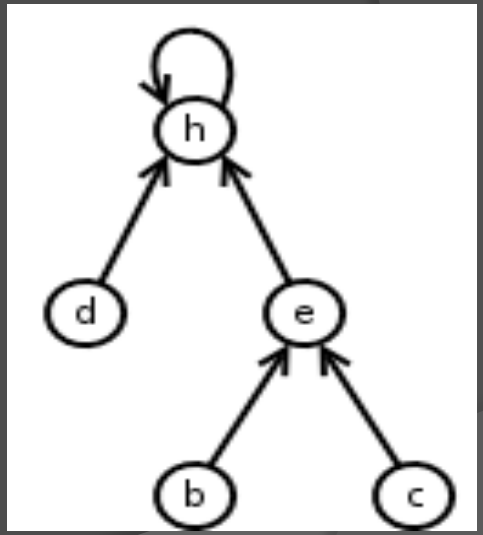
- `void make_set()`
- Creates a set containing one individual element
- Normally ran on all elements before any other operations

Disjoint Set – make_set()

```
int lead[],rank[];
void make_set(int n){
      lead[n] = n;
      rank[n] = 1;
}
```

Disjoint Set – Methods

- `int find()`
- Returns the topmost node of the set that the given element is in
- Usually implemented as a recursive function.
- In order to make this method fast, we apply **path compression**
  - Set each visited element's parent value as the top element to speed up future `find()` calls
  - Note that by using path compression, the values in `parent[]` might not represent an element's direct parent

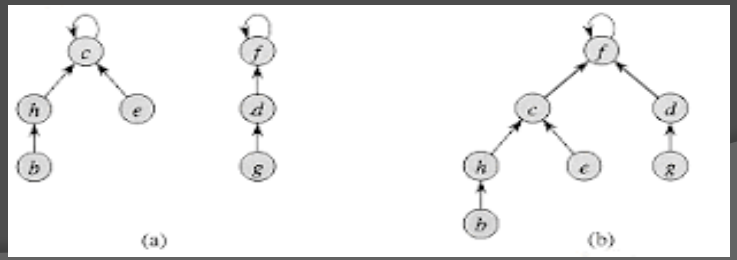


Disjoint Set – `find()`

```
int lead[],rank[];
int find(int n){
      if (lead[n] == n) return n;
      lead[n] = find(lead[n]);
      return lead[n];
}
```

Disjoint Set – Methods

- `void merge()`
- First checks if two elements are in different sets
- If they are different, change the top element of one set's parent value to the top element of the other set
- To improve time complexity, we use **union by rank**
- Always merge the set with a smaller rank to the other set since time complexity depends on number of ranks (if there is a tie, arbitrarily choose a set to merge into the other)
  - Similar to weighted union heuristic
  - This is the main purpose of `rank[]`



Disjoint Set – merge()

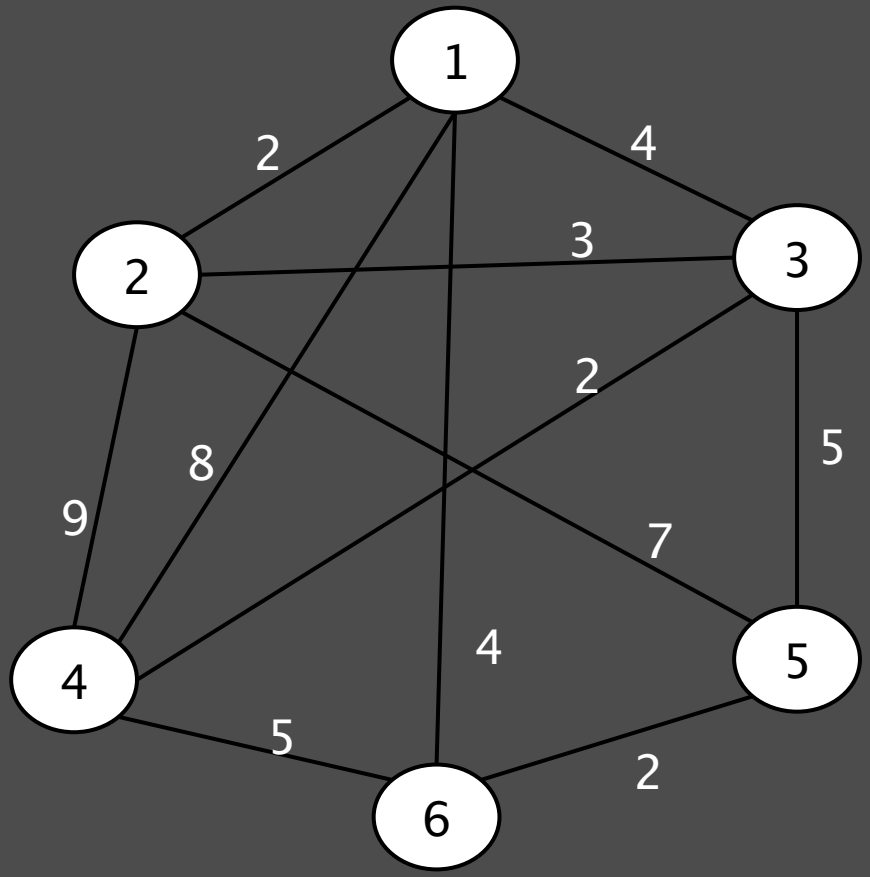
```
int lead[],rank[];
void merge(int a, int b){
      a = find(a);
      b = find(b);
      if (a != b){
            if (rank[a] > rank[b]) lead[b] = a;
            else{
                  lead[a] = b;
                  if (rank[a] == rank[b]) rank[b]++;
            }
      }
}
```

Disjoint Set Forest Analysis

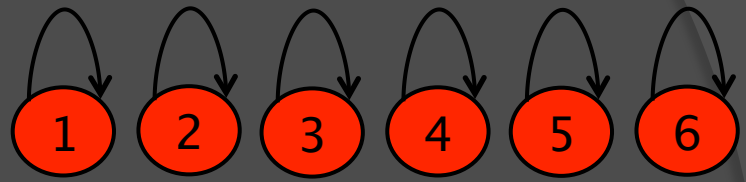
- `make()` – $O(1)$
- `find()` – amortized $O(1)$
- `merge()` – amortized $O(1)$
- Time complexity for `find()` and `merge()` using only union by rank optimization – $O(\log N)$
- Time complexity for `find()` and `merge()` using only path compression – amortized $O(\log N)$
- Time complexity for `find()` and `merge()` without any optimizations – $O(N)$
- Time complexity for M operations (using fully optimized methods) – amortized $O(M)$
- Amortized means that using the worst case per operation every time is too pessimistic; actual worst case happens very infrequently
  - More information another time!

Kruskal's Algorithm – Summary

- Make a set for each node (use **`make_set()`**)
- Sort all of the edges by increasing cost
- Iterate through sorted edges
  - Take the nodes the edge connects and check if they are in the same set (use **`find()`**)
  - If they are not the same set, combine the two sets (use **`merge()`**) and add edge to the MST
    - Instead of the two steps above, **`merge()`** can return a **`bool`** representing if merge was successful or not
    - If the merge was successful, then add the edge to the MST
  - Stop when there are $V-1$ edges



Visual Disjoint Sets:



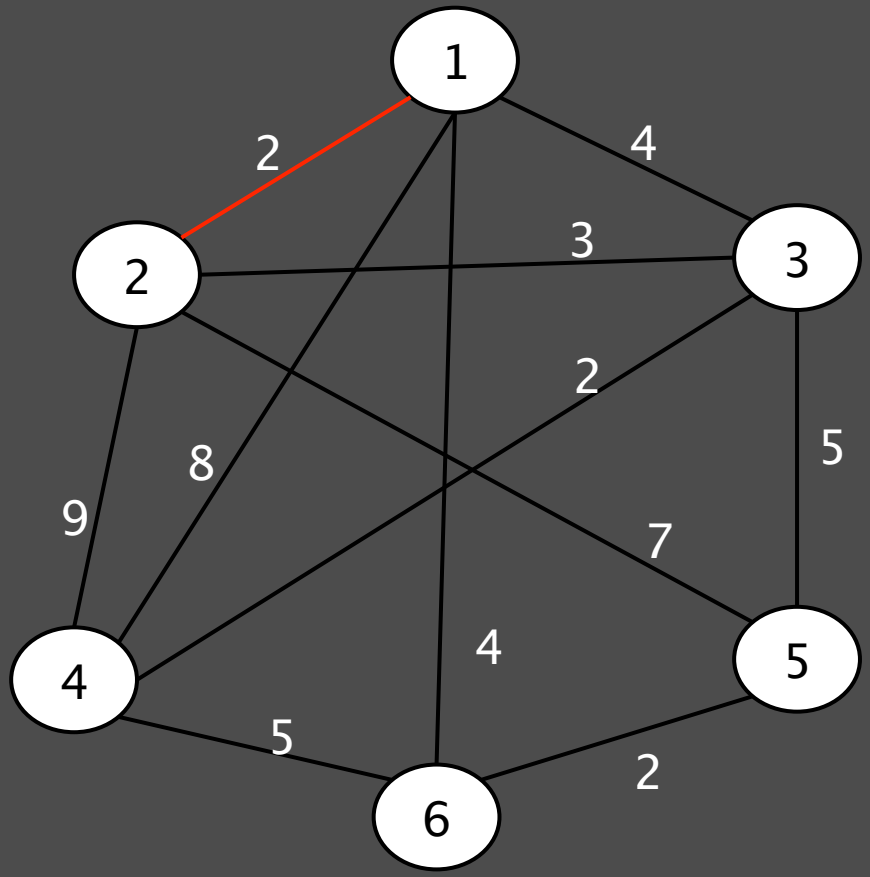
Sort edges, call **`make_set()`**

Lead

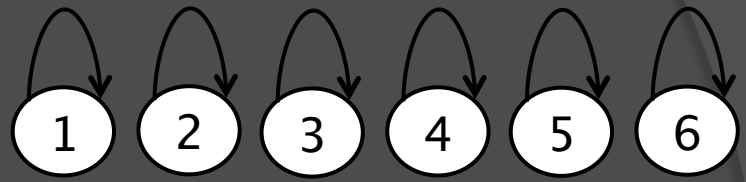
1	2	3	4	5	6
1	2	3	4	5	6

Rnk

1	2	3	4	5	6
1	1	1	1	1	1



Visual Disjoint Sets:



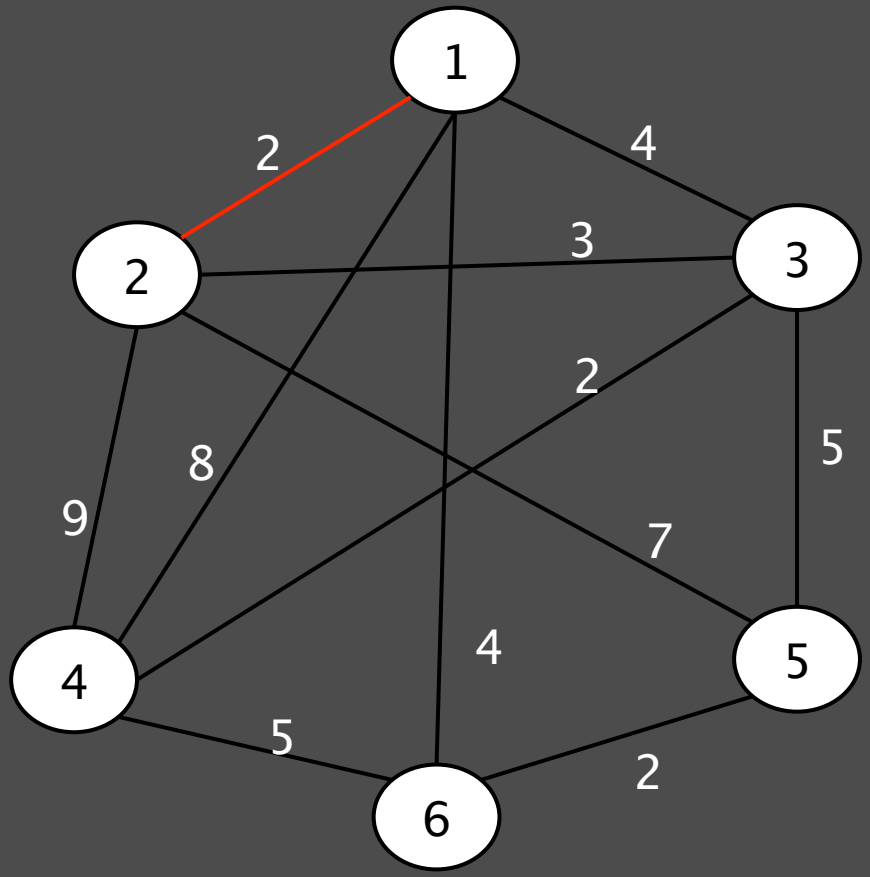
Next edge: nodes are not in the same set

Lead

1	2	3	4	5	6
1	2	3	4	5	6

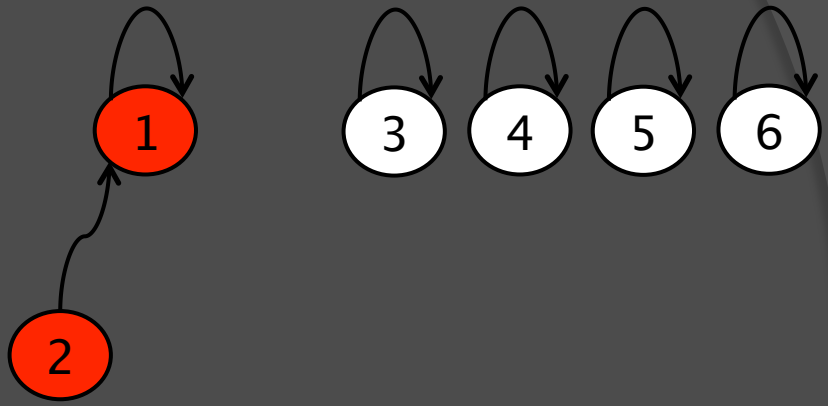
Rnk

1	2	3	4	5	6
1	1	1	1	1	1



Merge 1 and 2

Visual Disjoint Sets:

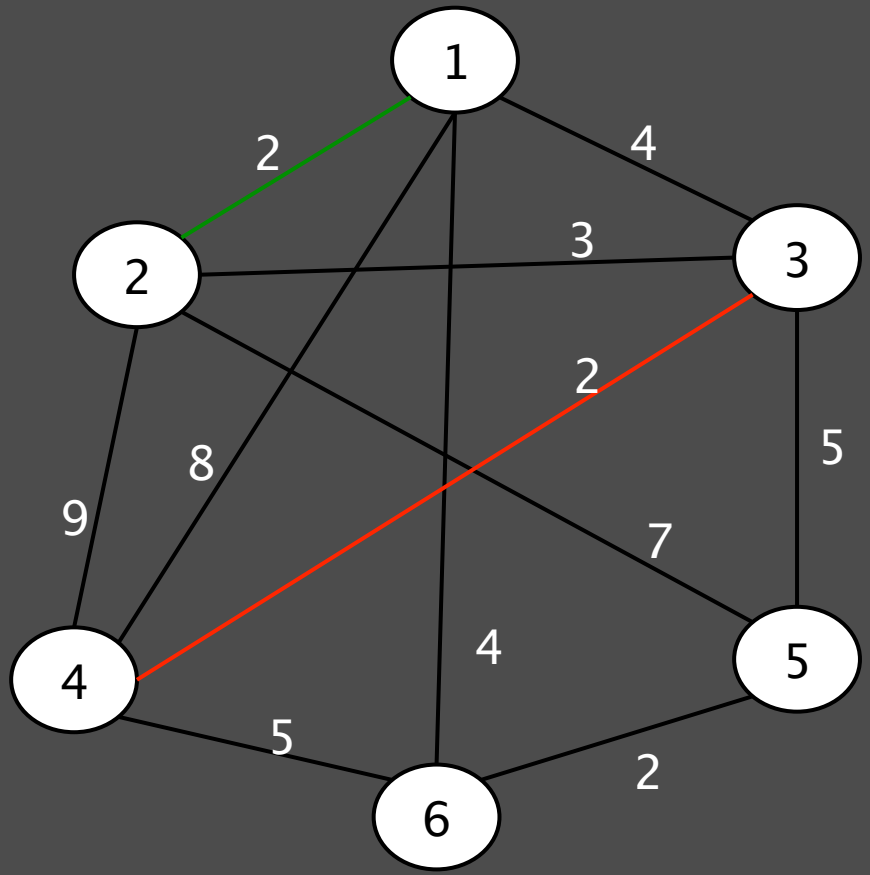


Lead

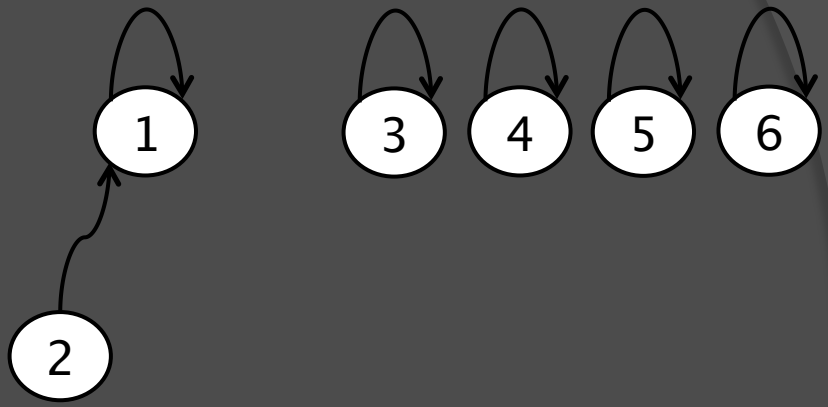
1	2	3	4	5	6
1	1	3	4	5	6

Rnk

1	2	3	4	5	6
2	1	1	1	1	1



Visual Disjoint Sets:



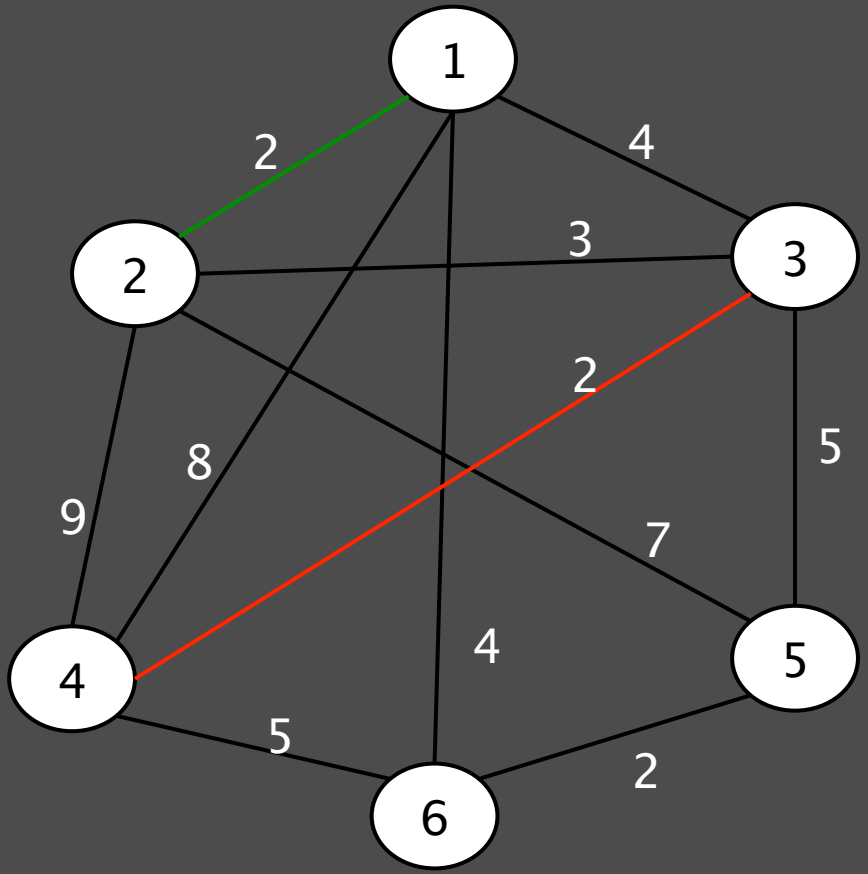
Next edge: nodes are not in the same set

Lead

1	2	3	4	5	6
1	1	3	4	5	6

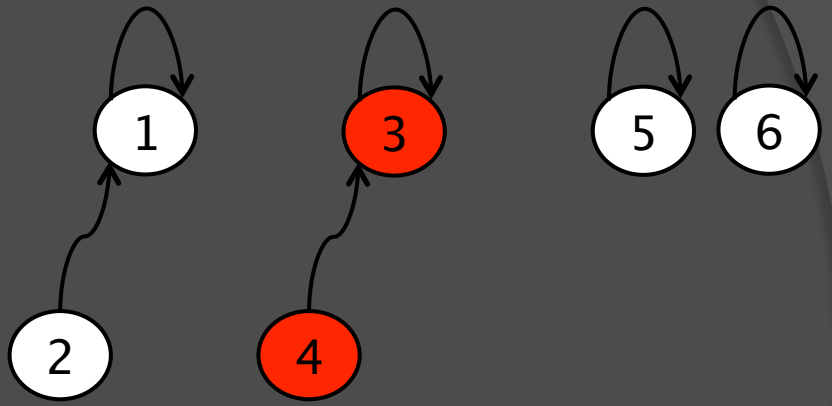
Rnk

1	2	3	4	5	6
2	1	1	1	1	1



Merge 3 and 4

Visual Disjoint Sets:

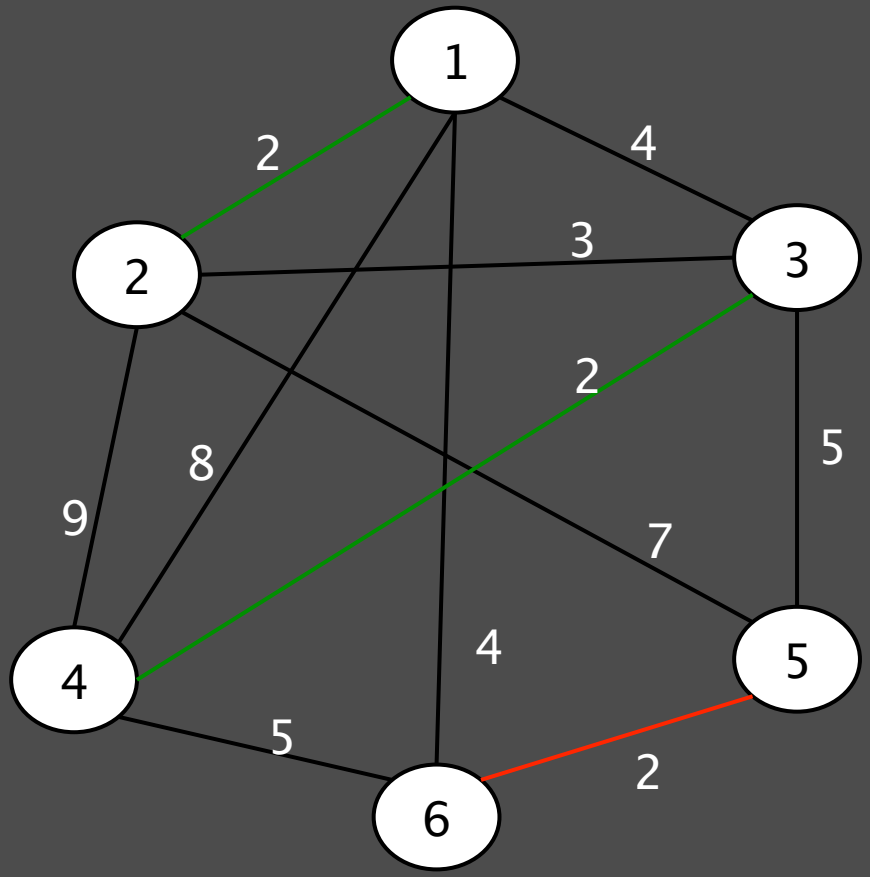


Lead

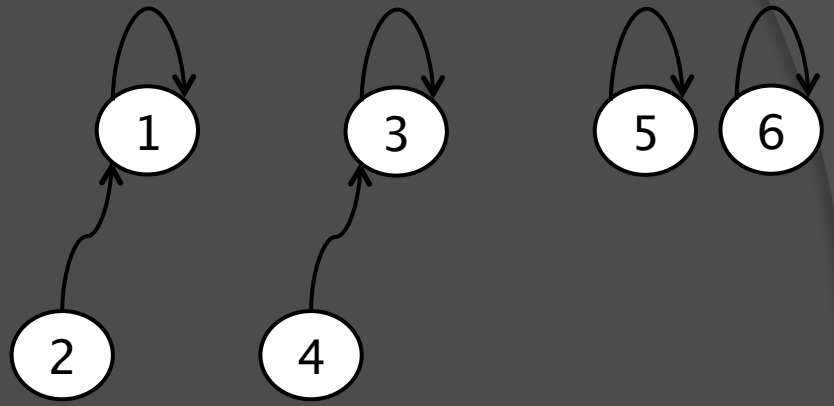
1	2	3	4	5	6
1	1	3	3	5	6

Rnk

1	2	3	4	5	6
2	1	2	1	1	1



Visual Disjoint Sets:



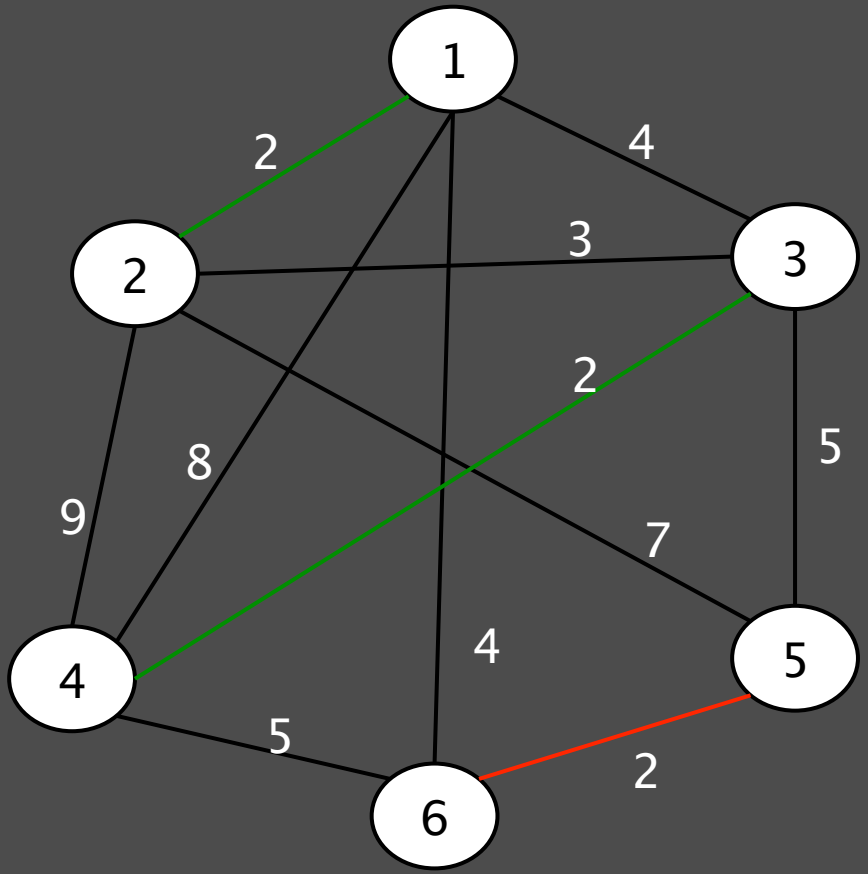
Lead

1	2	3	4	5	6
1	1	3	3	5	6

Rnk

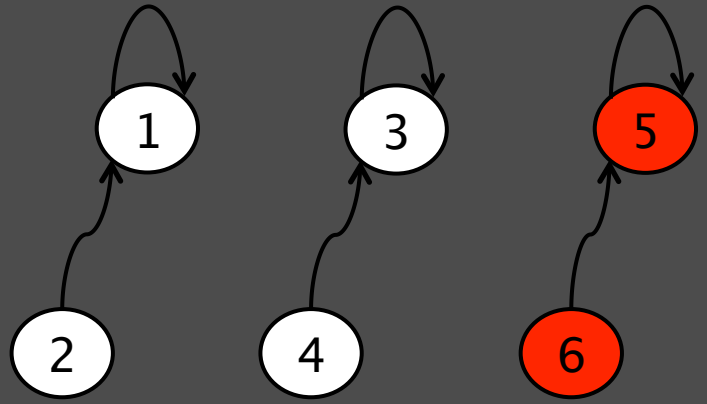
1	2	3	4	5	6
2	1	2	1	1	1

Next edge: nodes are not in the same set



Merge 5 and 6

Visual Disjoint Sets:

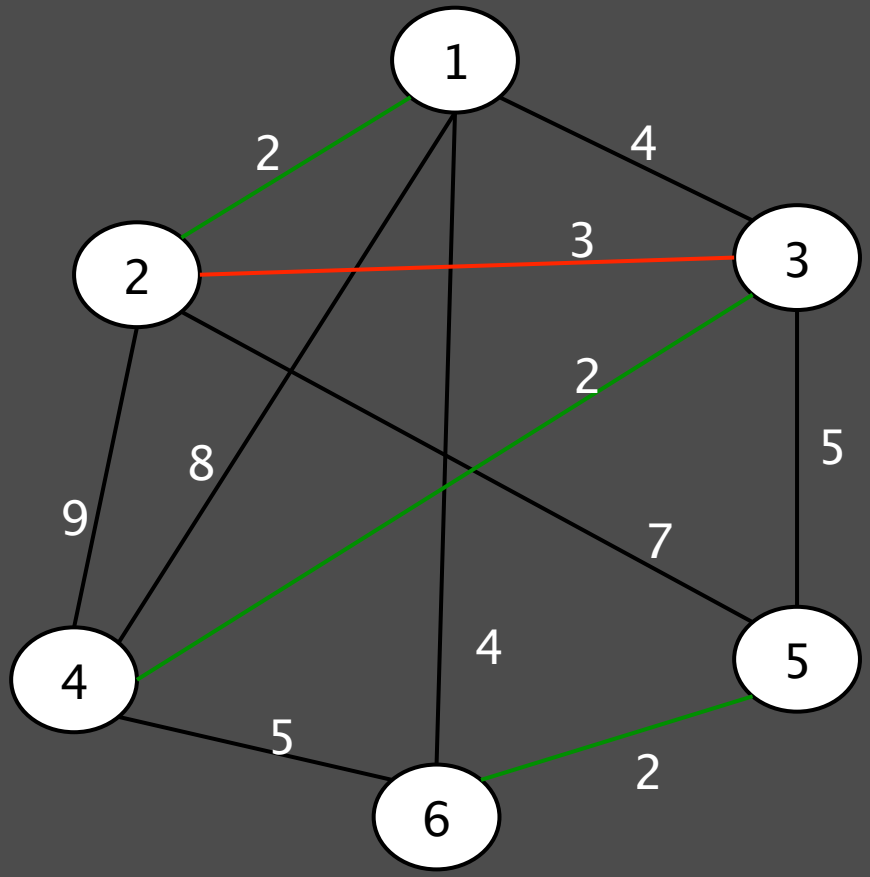


Lead

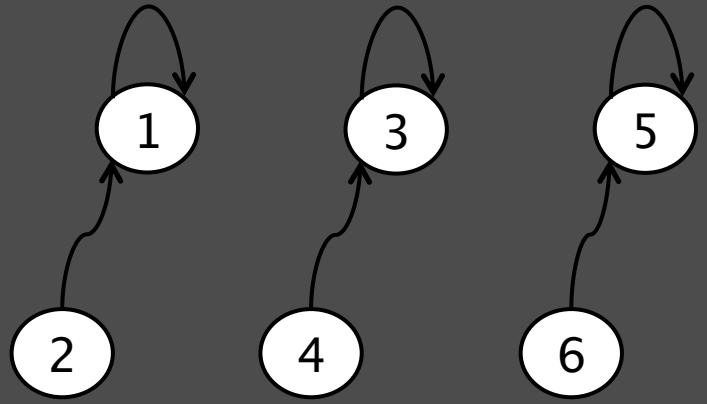
1	2	3	4	5	6
1	1	3	3	5	5

Rnk

1	2	3	4	5	6
2	1	2	1	2	1



Visual Disjoint Sets:



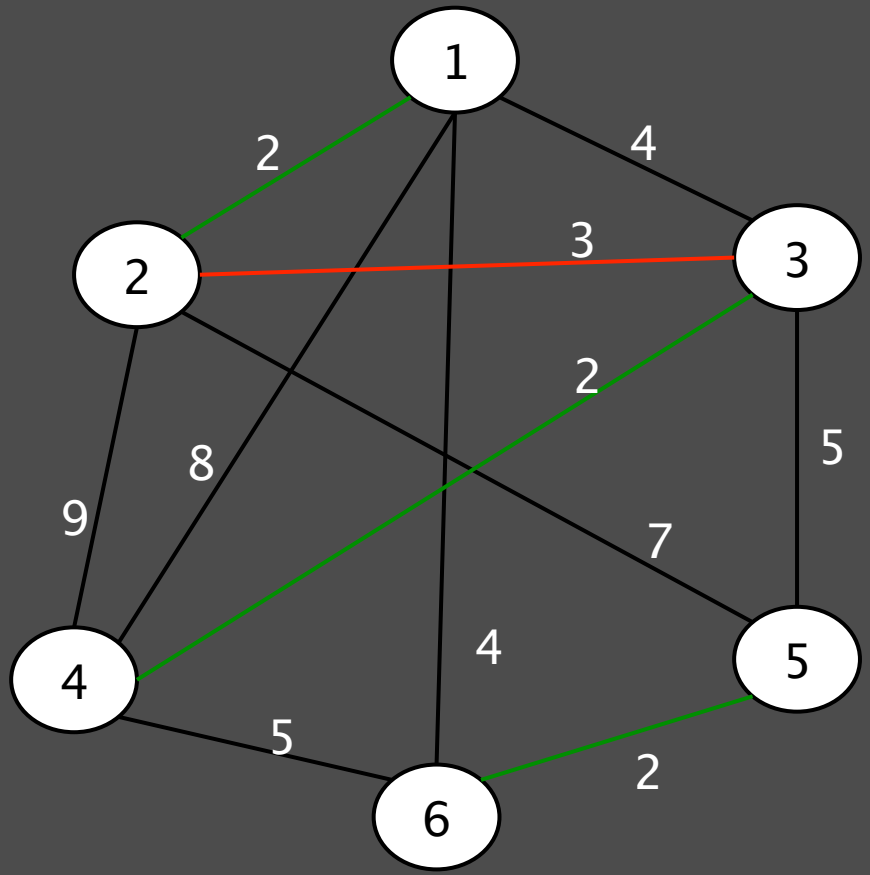
Next edge: nodes are not in the same set

Lead

1	2	3	4	5	6
1	1	3	3	5	5

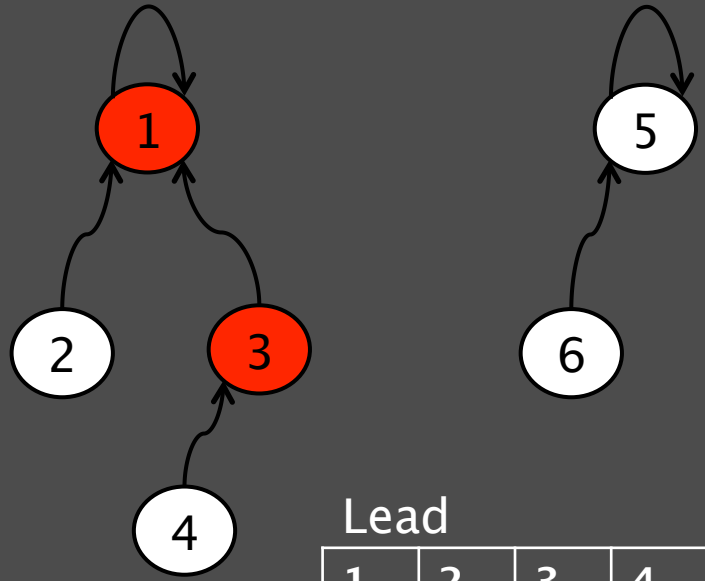
Rnk

1	2	3	4	5	6
2	1	2	1	2	1



Merge 2 and 3

Visual Disjoint Sets:

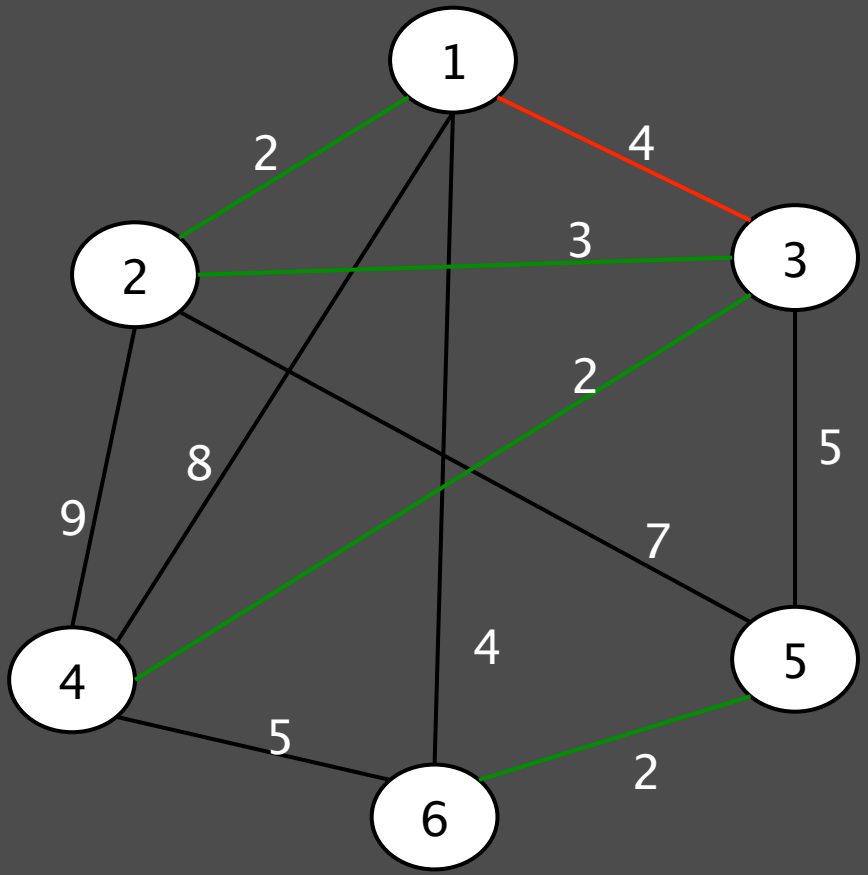


Lead

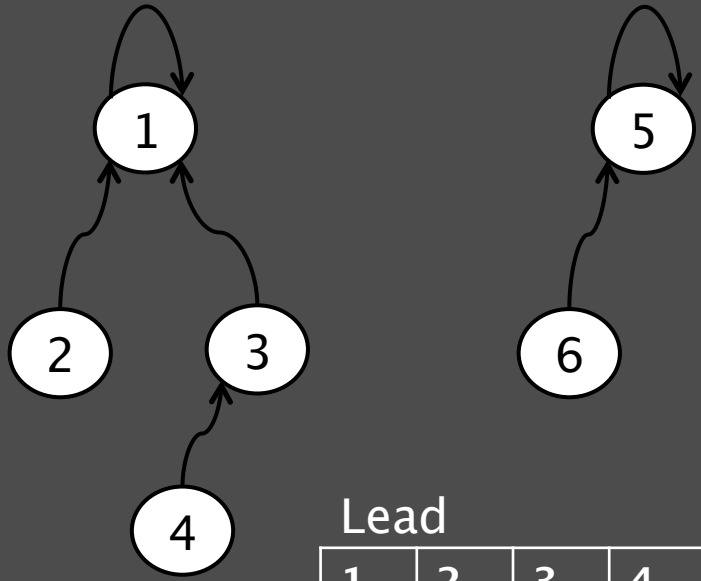
1	2	3	4	5	6
1	1	1	3	5	5

Rnk

1	2	3	4	5	6
3	1	2	1	2	1



Visual Disjoint Sets:



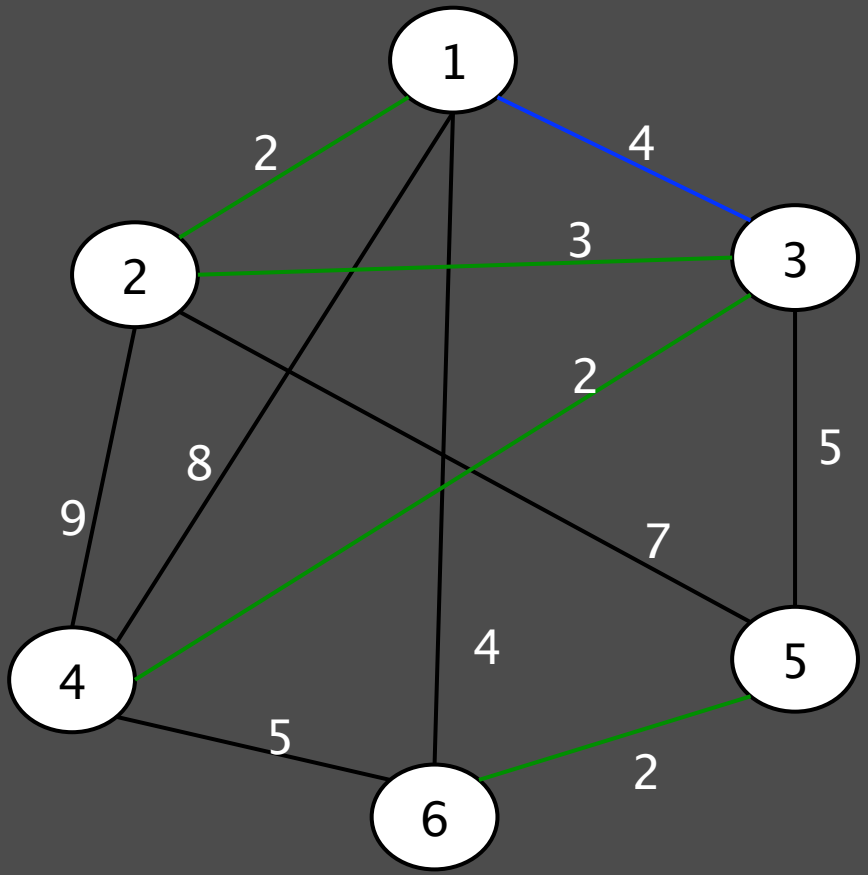
Next edge: nodes are in the same set

Lead

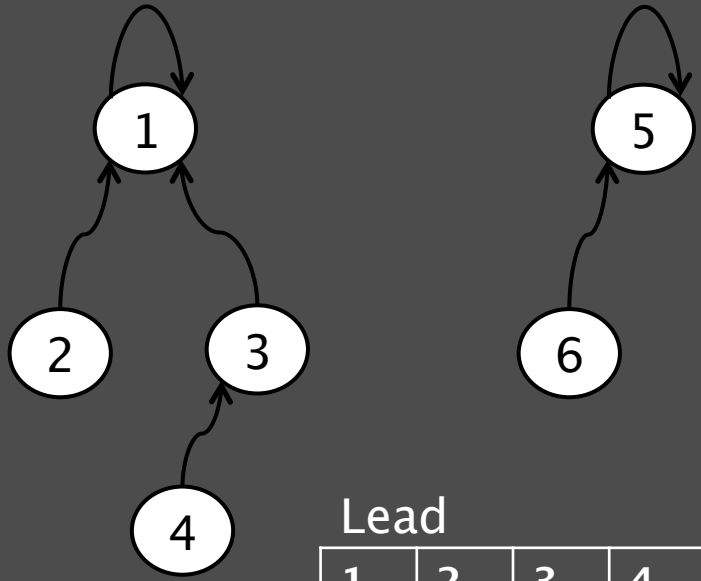
1	2	3	4	5	6
1	1	1	3	5	5

Rnk

1	2	3	4	5	6
3	1	2	1	2	1



Visual Disjoint Sets:



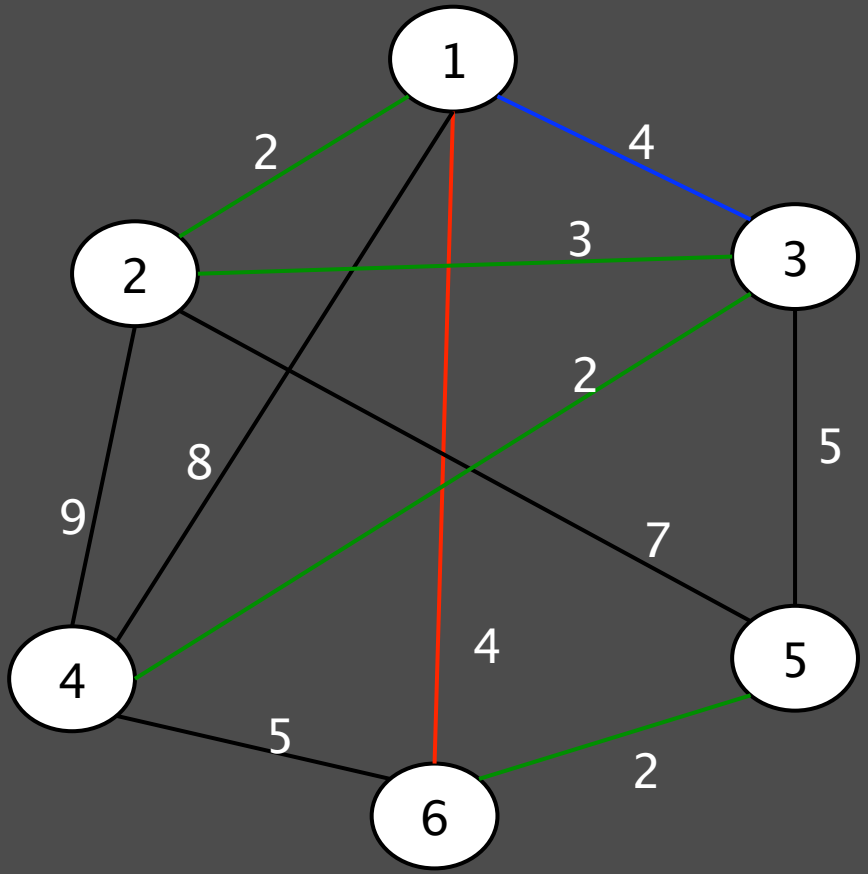
Ignore edge

Lead

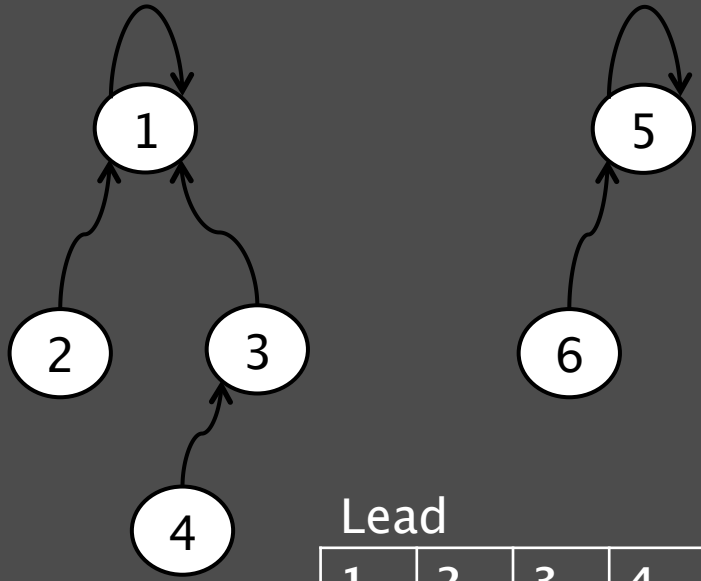
1	2	3	4	5	6
1	1	1	3	5	5

Rnk

1	2	3	4	5	6
3	1	2	1	2	1



Visual Disjoint Sets:



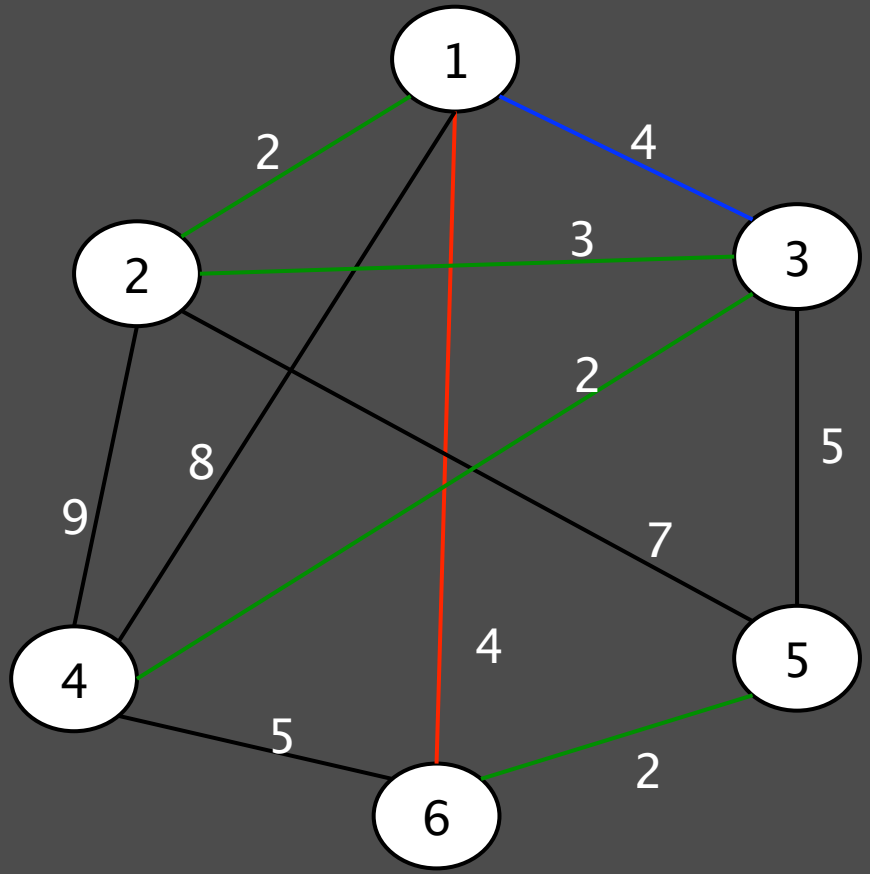
Next edge: nodes are not in the same set

Lead

1	2	3	4	5	6
1	1	1	3	5	5

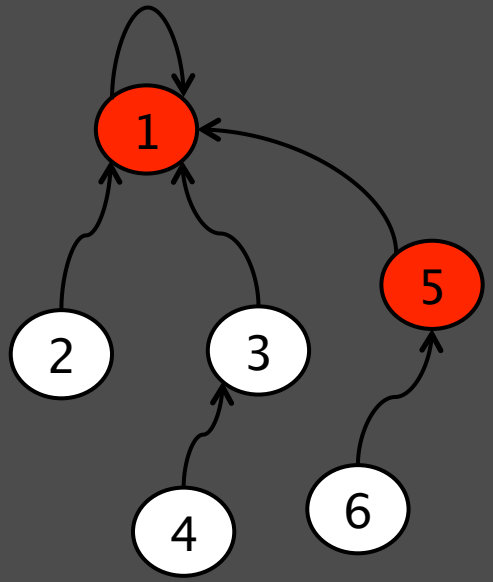
Rnk

1	2	3	4	5	6
3	1	2	1	2	1



Merge 1 and 6

Visual Disjoint Sets:

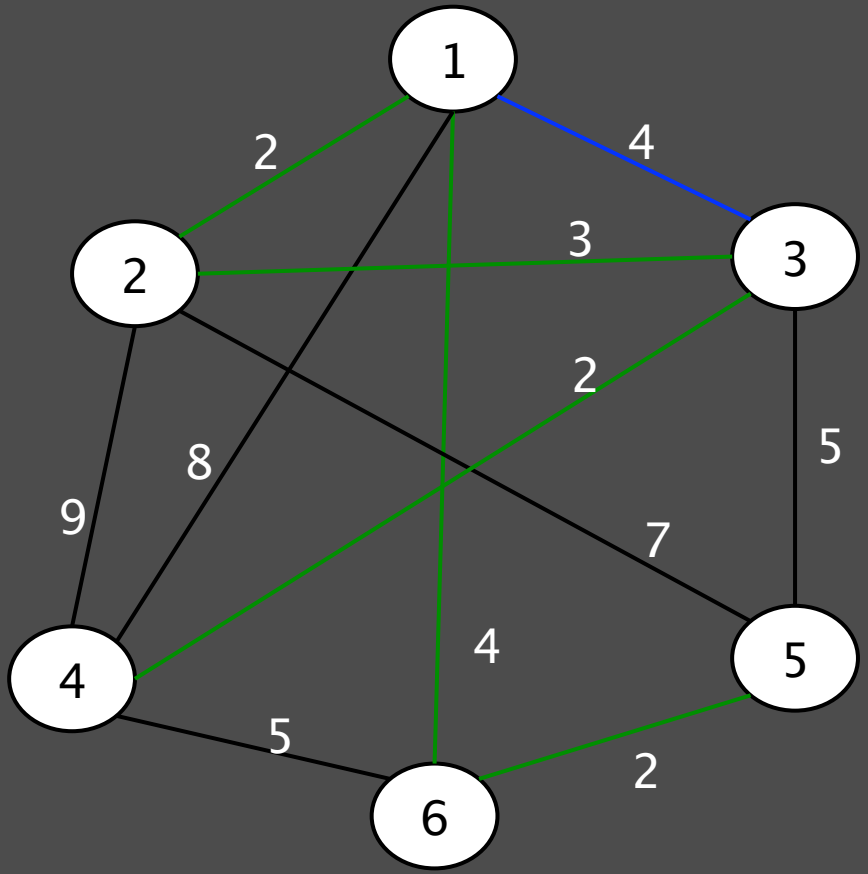


Lead

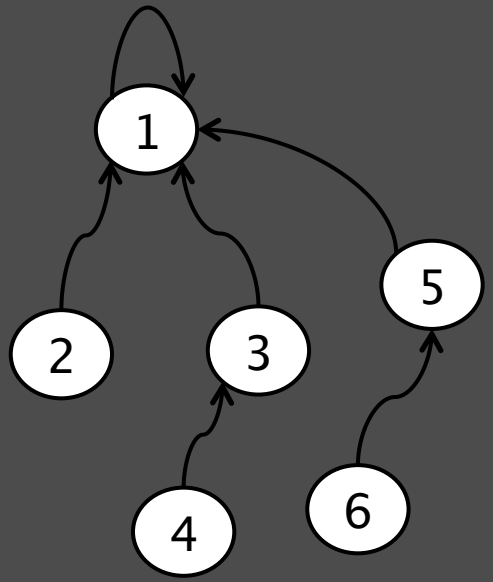
1	2	3	4	5	6
1	1	1	3	1	5

Rnk

1	2	3	4	5	6
3	1	2	1	2	1



Visual Disjoint Sets:



Lead

1	2	3	4	5	6
1	1	1	3	1	5

Rnk

1	2	3	4	5	6
3	1	2	1	2	1

There are *V-1* edges in the MST. Therefore, we are done.

THANK YOU!