Learning Algorithm: Application of Union Find in Undirected Graph Problems

Qiang (Neo) Hao

Learning Design and Technology & Computer Science

University of Georgia

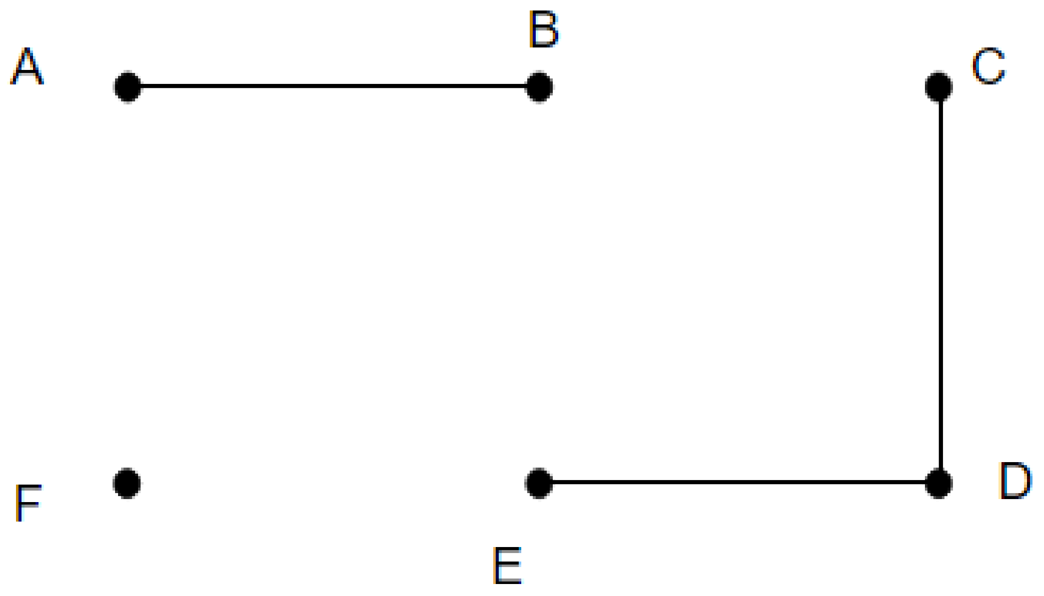
Structure

- Union Find
  - Naïve Version
    - *Practice*
  - Optimized Version
    - *Practice*
- Cycle Detection
  - *Practice*
- Kruskal's algorithm

https://github.com/Neo-Hao/union-find

Union Find

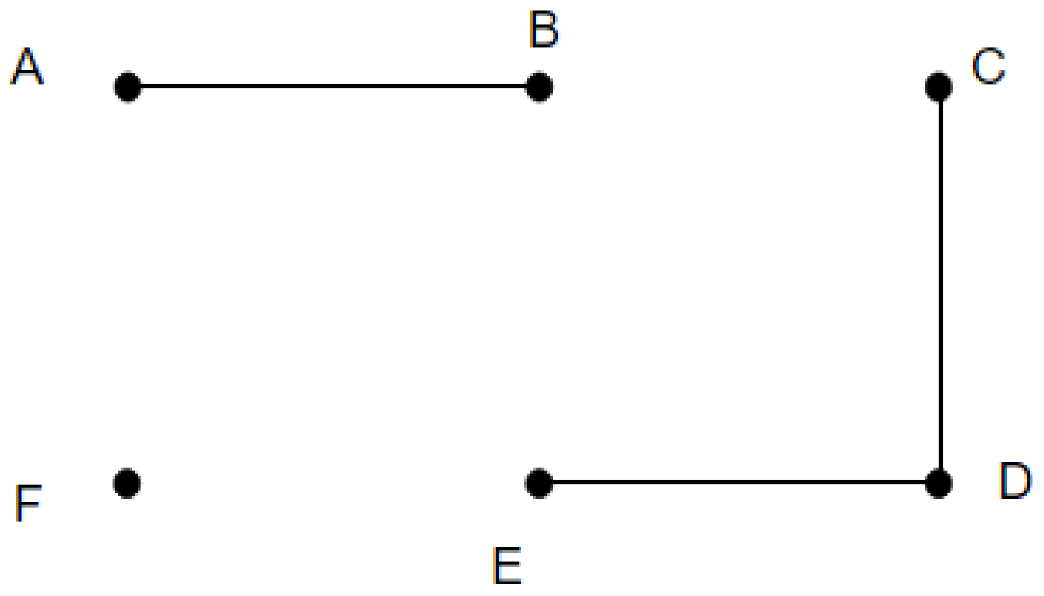
Data Structure



Given a graph, determine whether two vertices are somehow connected.

Union Find

Data Structure

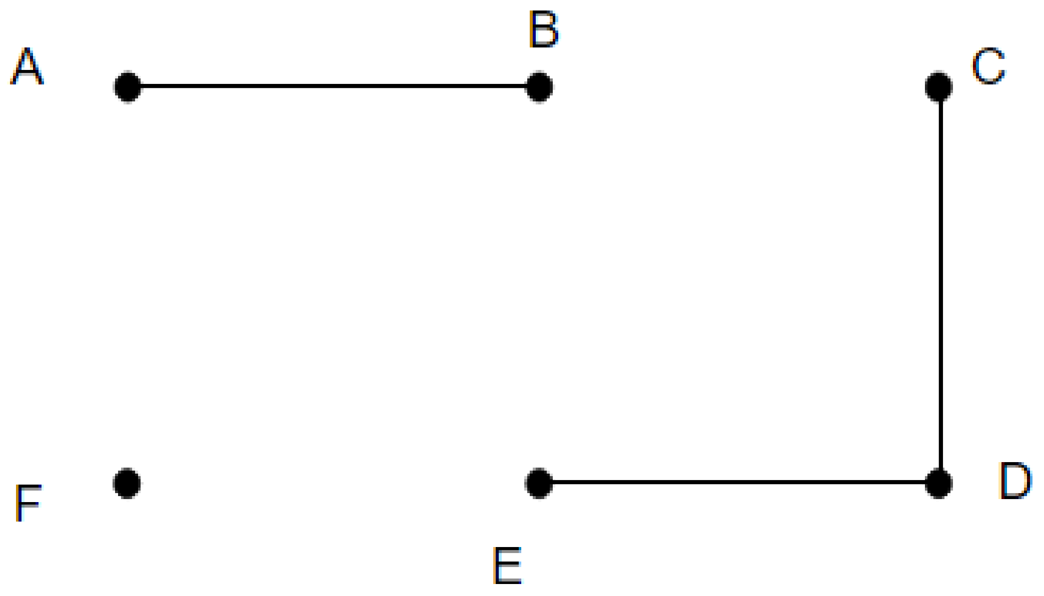


Given a graph, determine whether two vertices are somehow connected.

- makeSet
- union
- findSet

Union Find

Data Structure

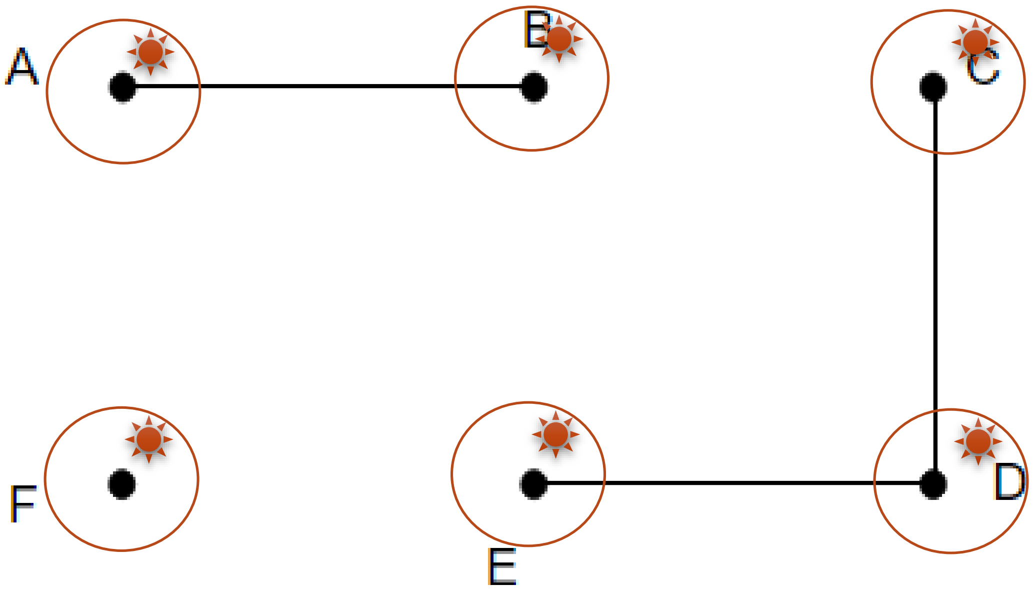


Given a graph, determine whether two vertices are somehow connected.

1. makeSet(A)
2. makeSet(B)
3. makeSet(C)
4. makeSet(D)
5. makeSet(E)
6. makeSet(F)
7. union(A, B)
8. union(C, D)
9. union(E, D)
10. findSet(A) == findSet(D)

Union Find

Data Structure



Given a graph, determine whether two vertices are somehow connected.

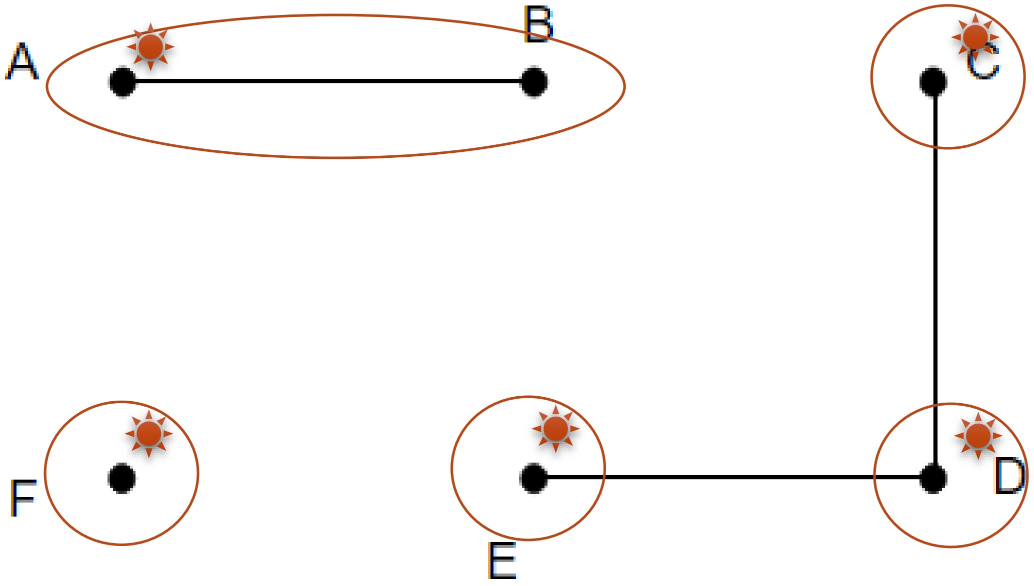
1. makeSet(A)

……

7. union(A, B)
8. union(C, D)
9. union(E, D)
10. findSet(A) == findSet(D)

Union Find

Data Structure



Given a graph, determine whether two vertices are somehow connected.

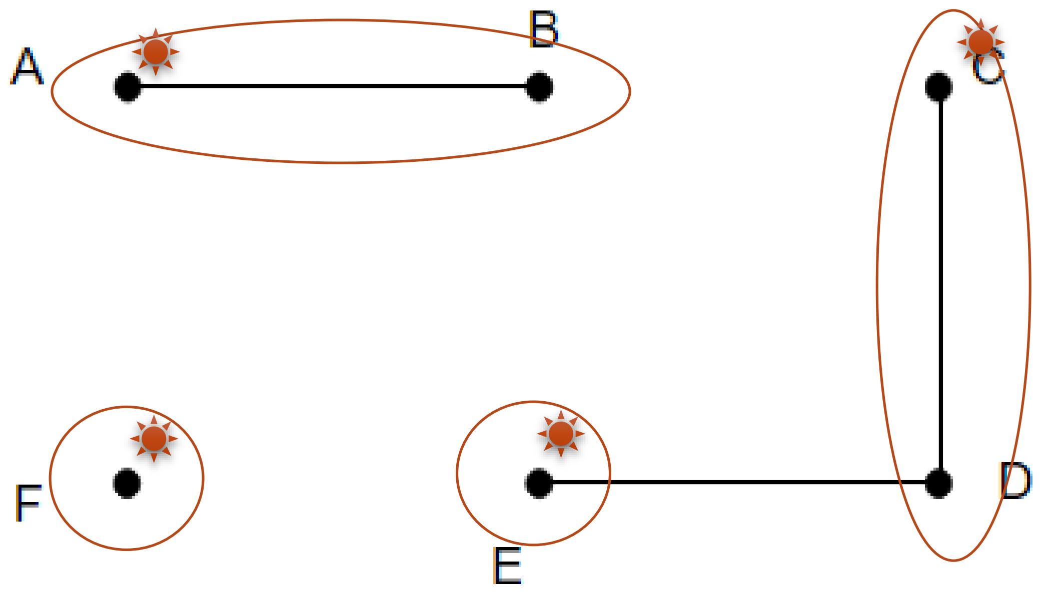
1. makeSet(A)

   ……

7. union(A, B)
8. union(C, D)
9. union(E, D)
10. findSet(A) == findSet(D)

Union Find

Data Structure



Given a graph, determine whether two vertices are somehow connected.

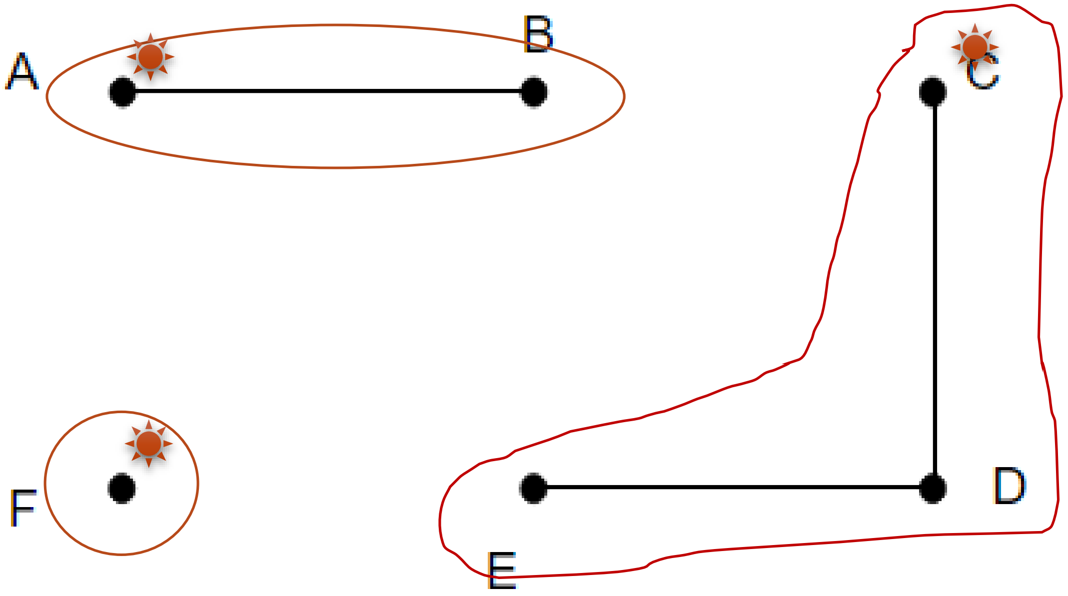
1. makeSet(A)

……

7. union(A, B)
8. union(C, D)
9. union(E, D)
10. findSet(A) == findSet(D)

Union Find

Data Structure



Given a graph, determine whether two vertices are somehow connected.

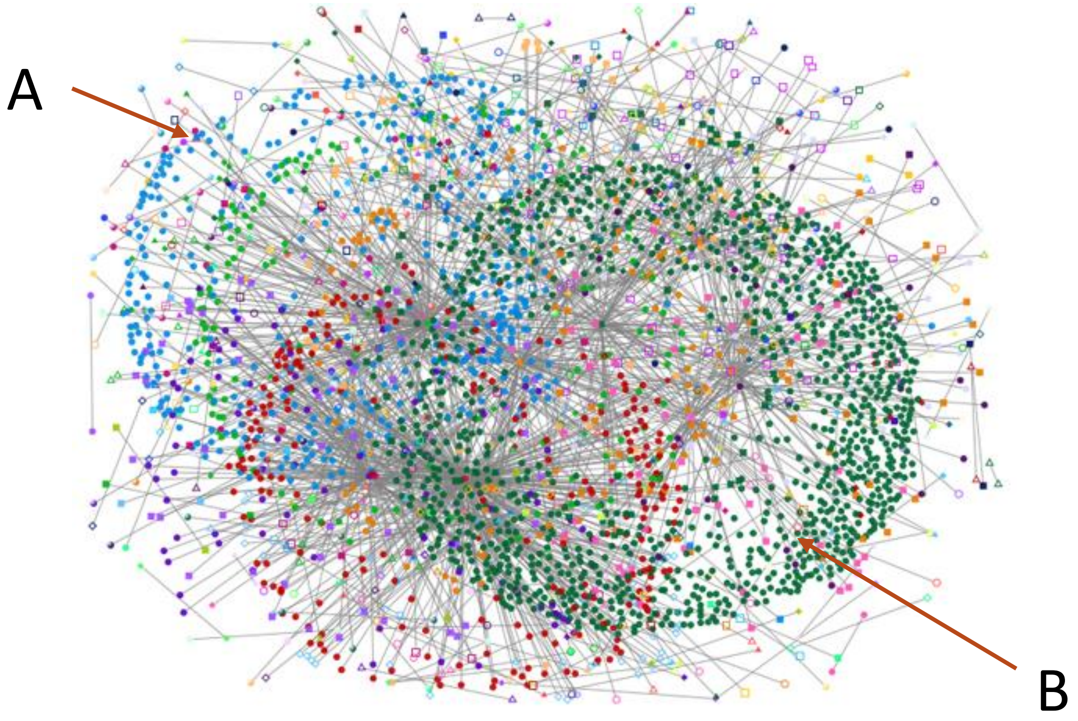
1. makeSet(A)

……

7. union(A, B)
8. union(C, D)
9. union(E, D)
10. findSet(A) == findSet(D)

Union Find

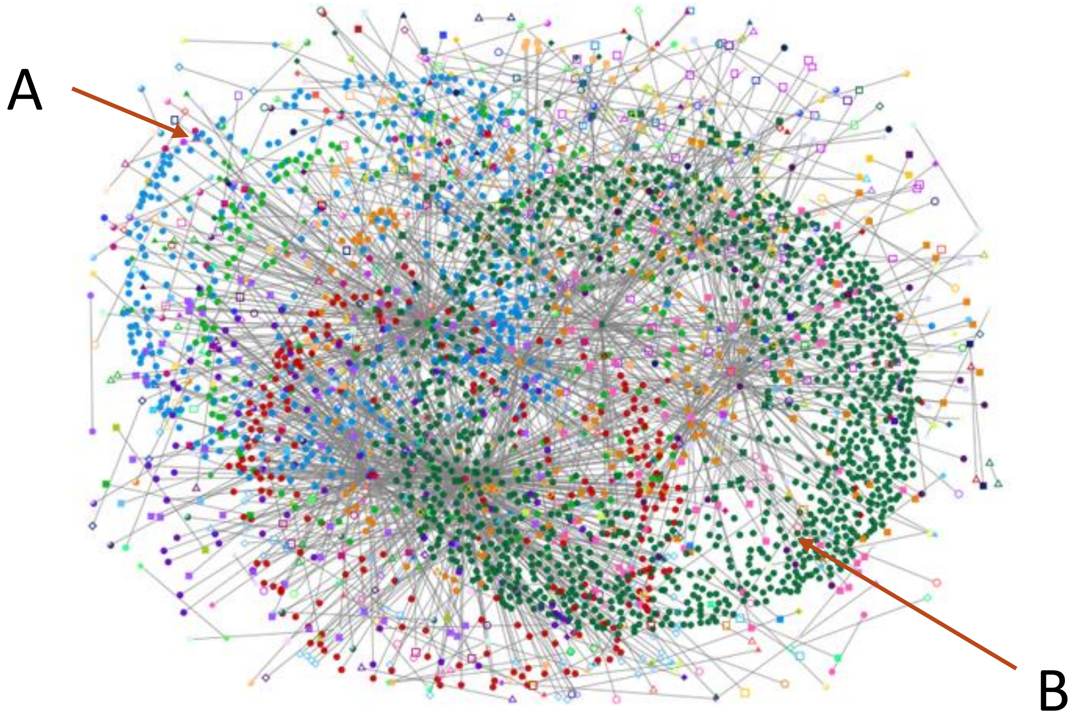
Data Structure



Given a graph, determine whether two vertices are somehow connected.

Union Find

Data Structure



Given a graph, determine whether two vertices are somehow connected.

Application:

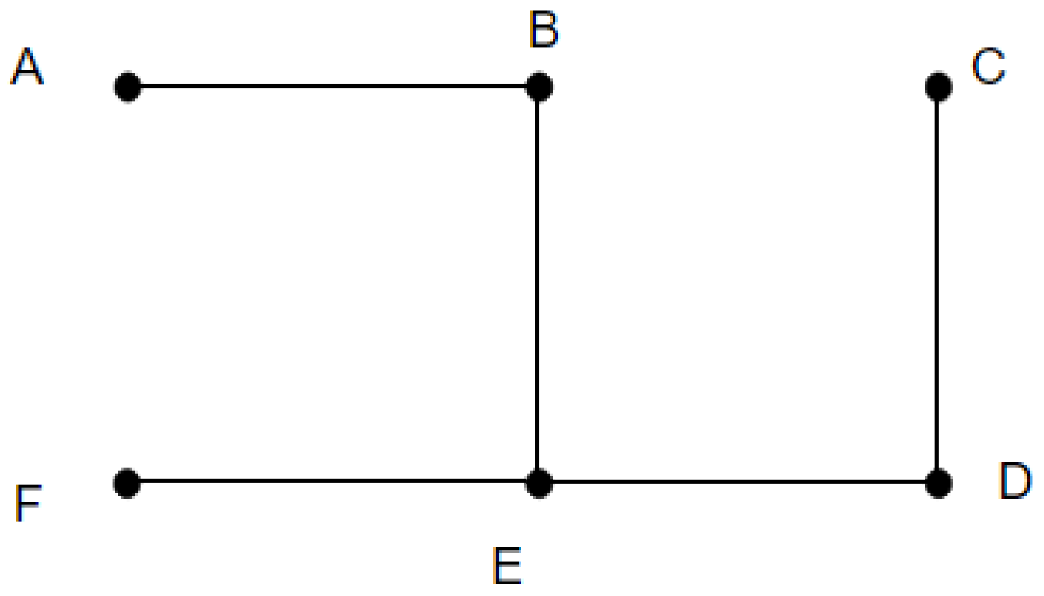
- Network security
- Social network analysis
- Image Processing

Union Find

Approaches

- Naïve Union Find
- Optimized Union Find

Union Find – Naïve Version

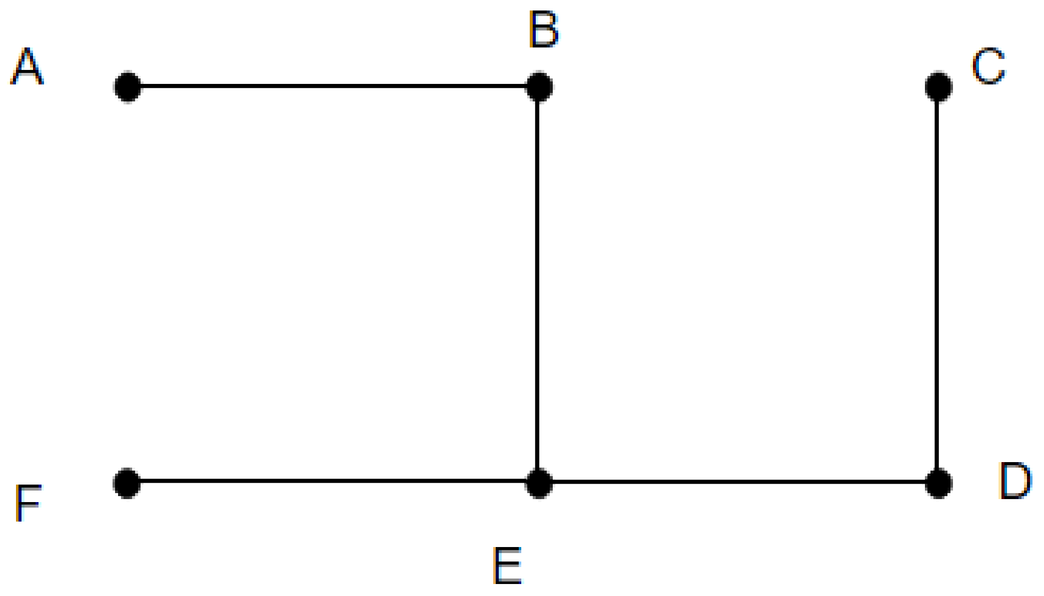


- makeSet
- union
- findSet

```
Node:
      val
      parent
```

Union Find – Naïve Version

makeSet



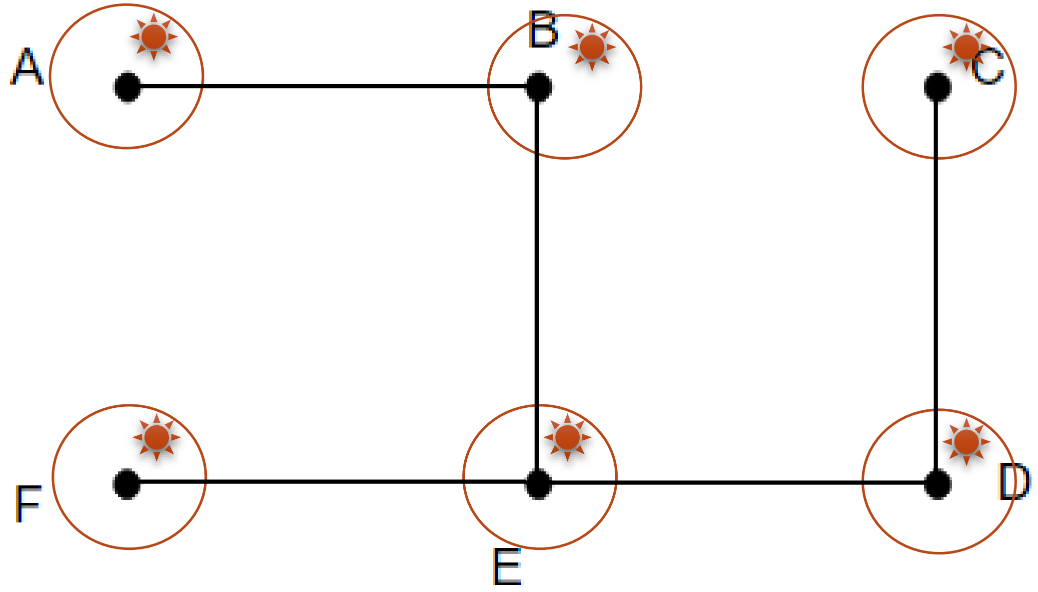

```
makeSet(v):
        n = Node(v)
        n.parent = n
```

Time Complexity

O(1)

Union Find – Naïve Version

makeSet



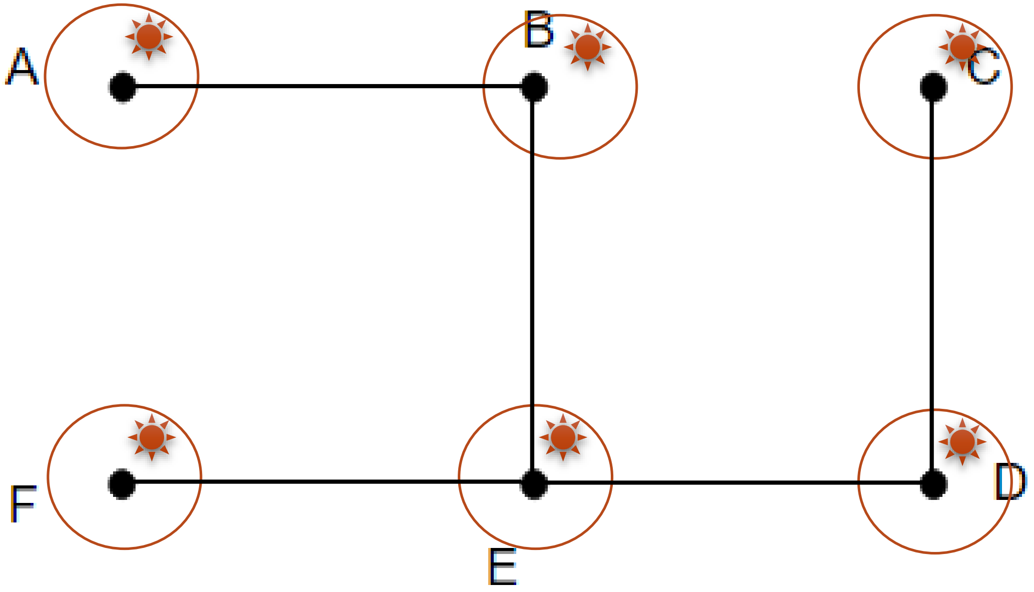

```
makeSet(v):
        n = Node(v)
        n.parent = n
```

Time Complexity

O(1)

Union Find – Naïve Version

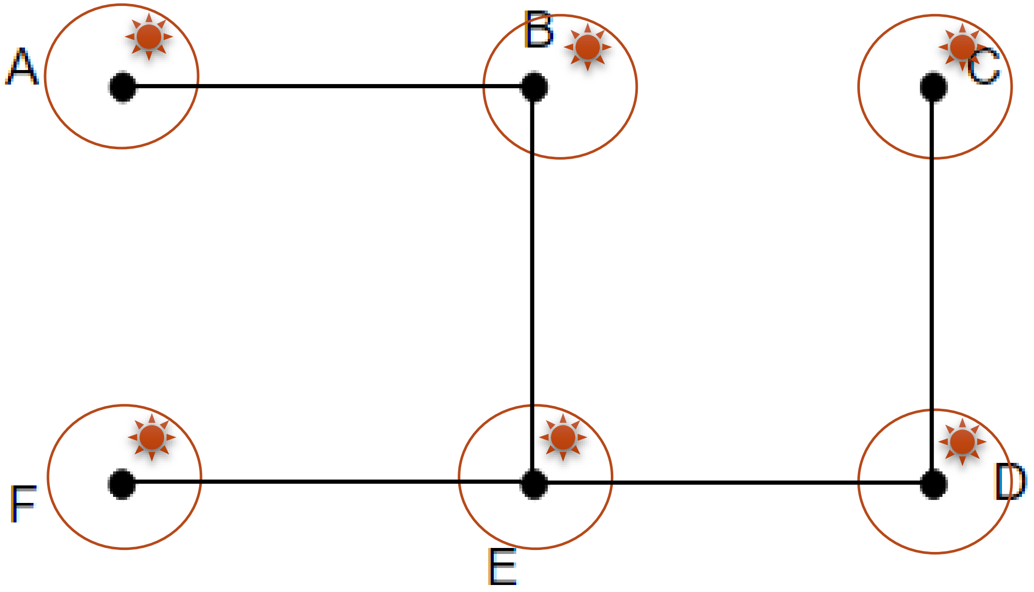
union




```
union(n1, n2):
        i = findSet(n1)
        j = findSet(n2)
        if i == j:
                return
        else:
                j.parent = i
```

Union Find – Naïve Version

union




```
union(n1, n2):
        i = findSet(n1)
        j = findSet(n2)
        if i == j:
                return
        else:
                j.parent = i
```

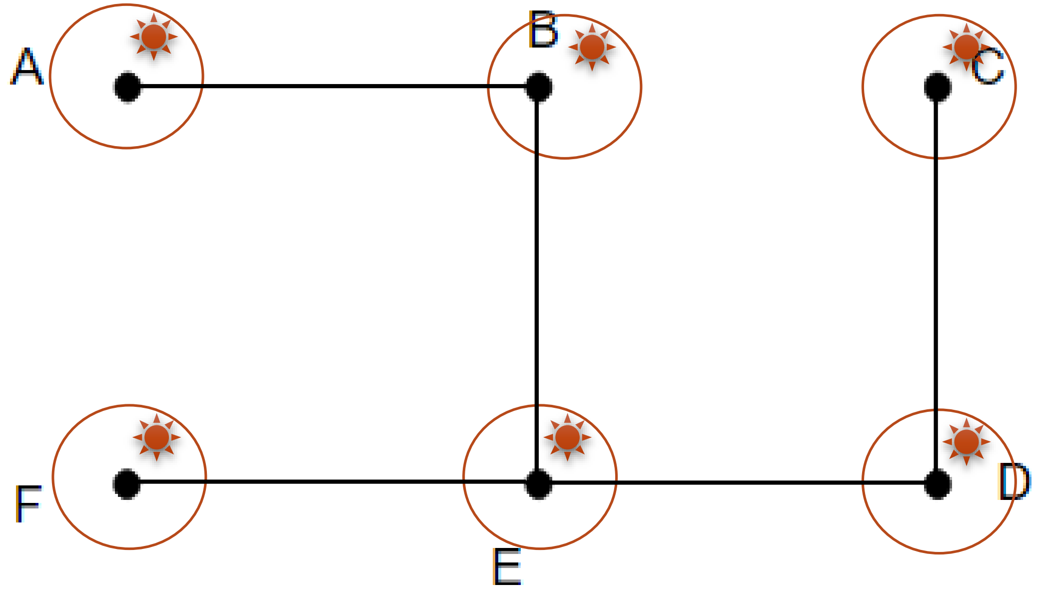
union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

C.parent ?
D.parent ?
E.parent ?

Union Find – Naïve Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

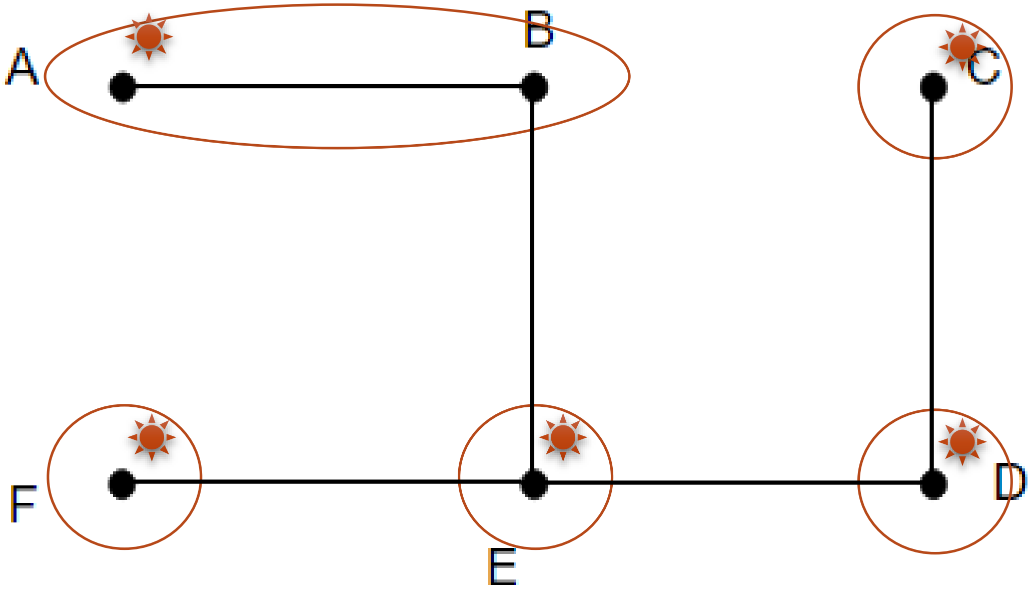
union

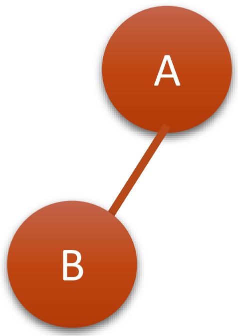


Union Find – Naïve Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

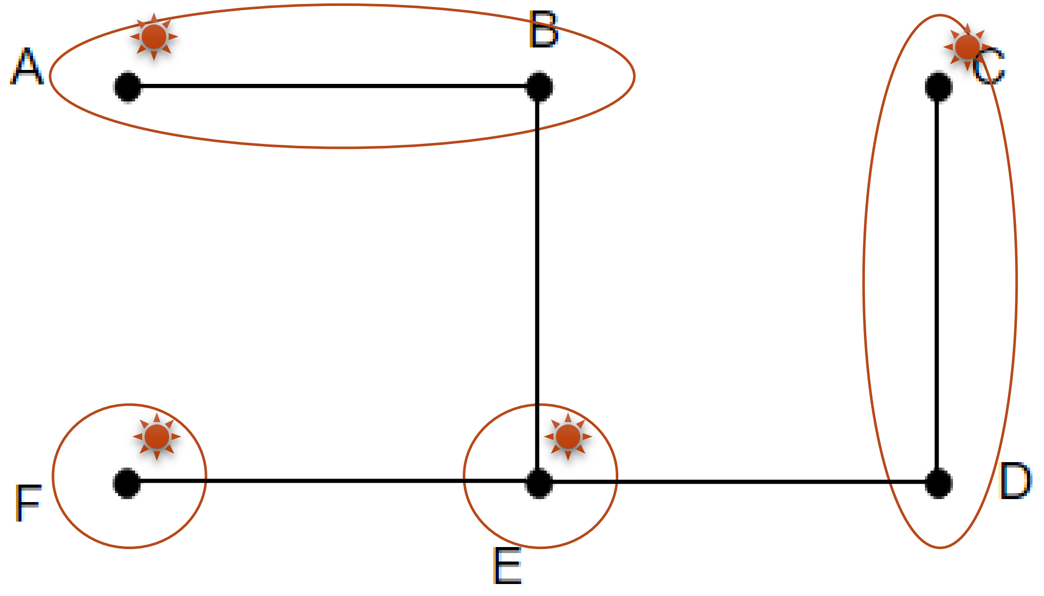


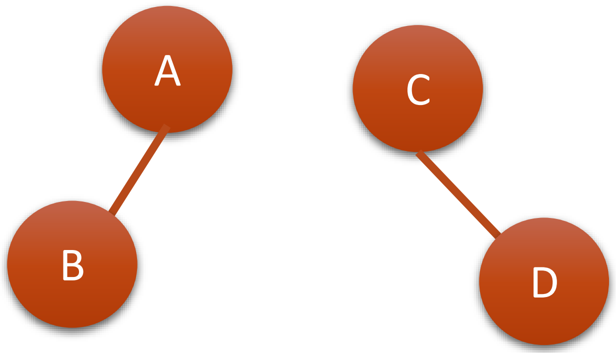


Union Find – Naïve Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

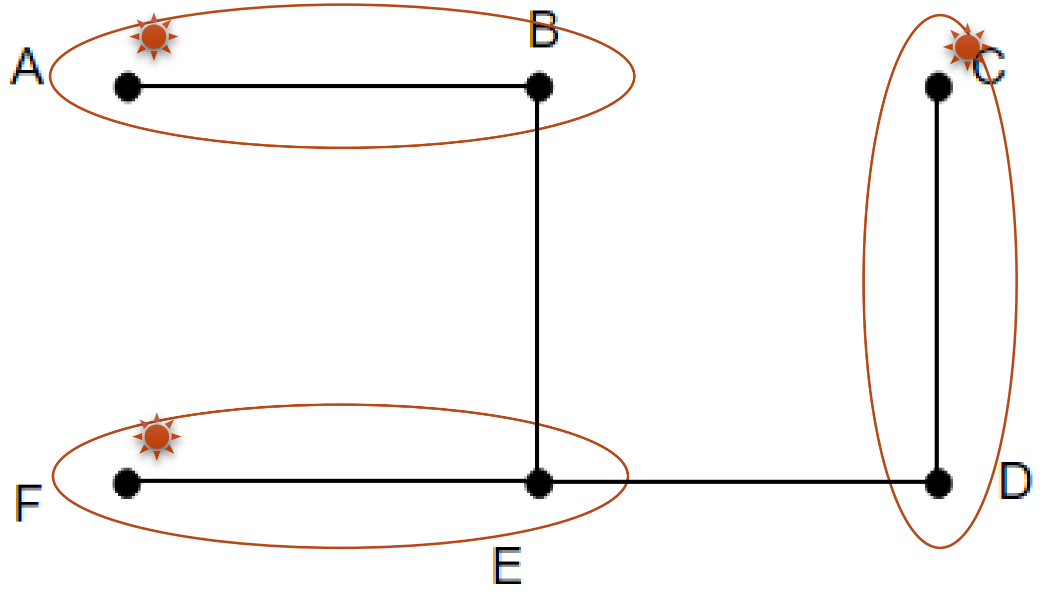


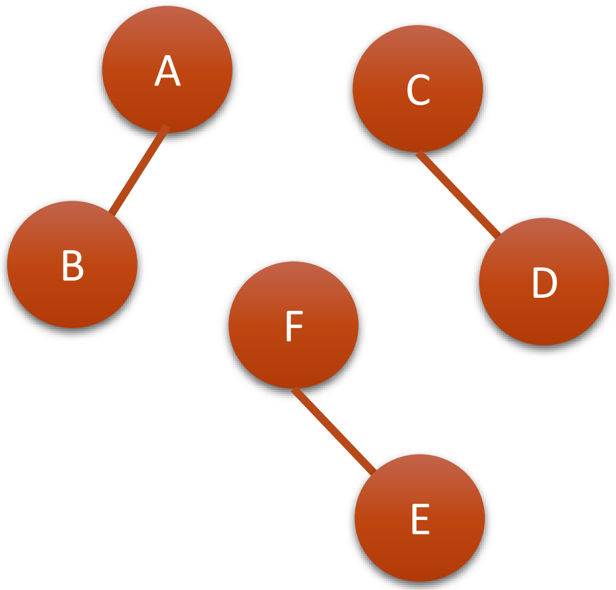


Union Find – Naïve Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

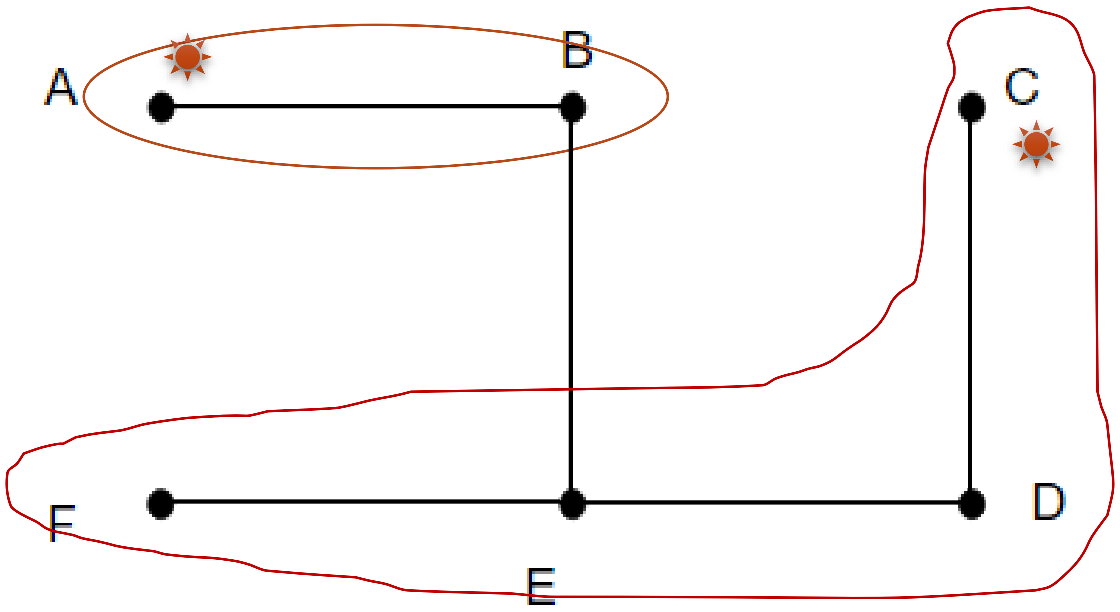


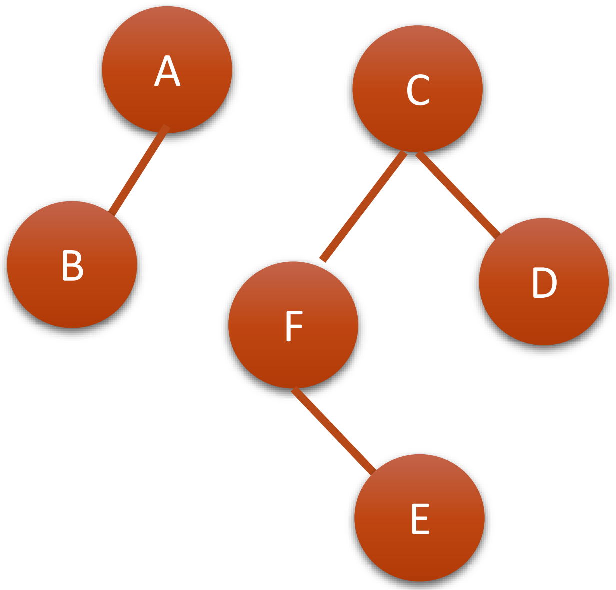


Union Find – Naïve Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

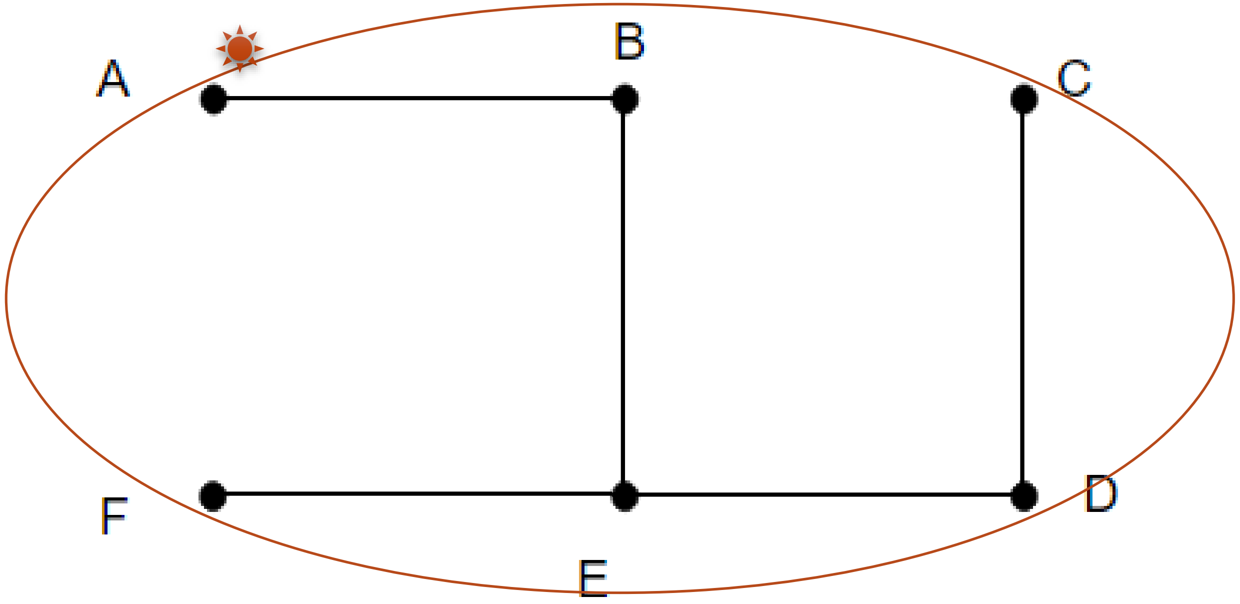


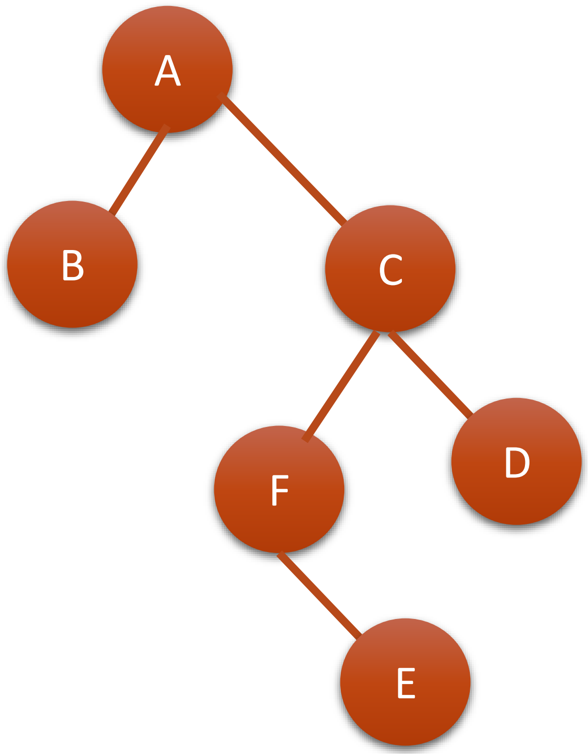


Union Find – Naïve Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

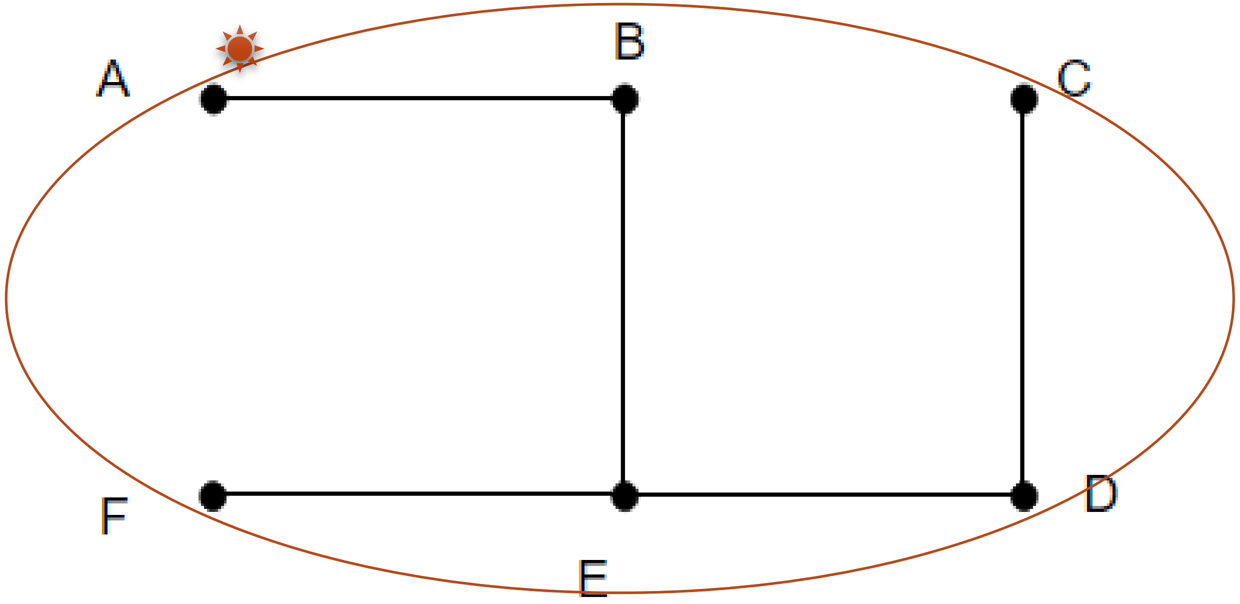


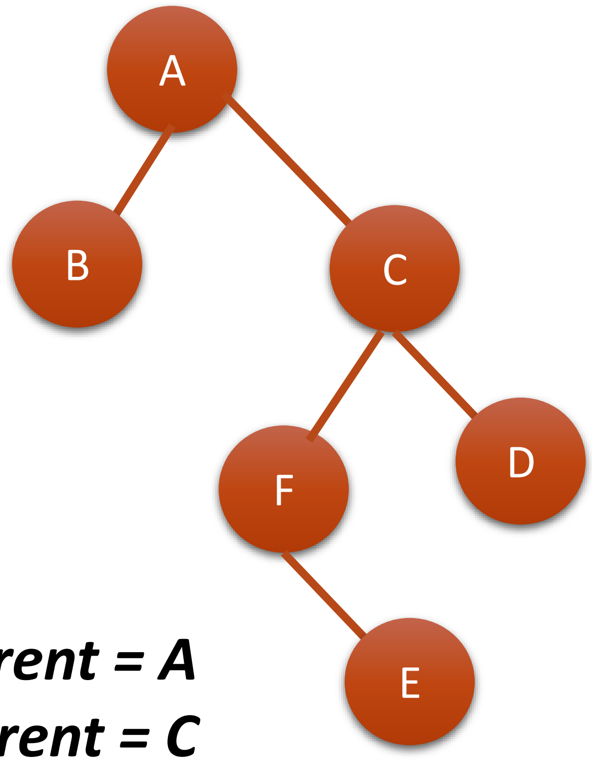


Union Find – Naïve Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

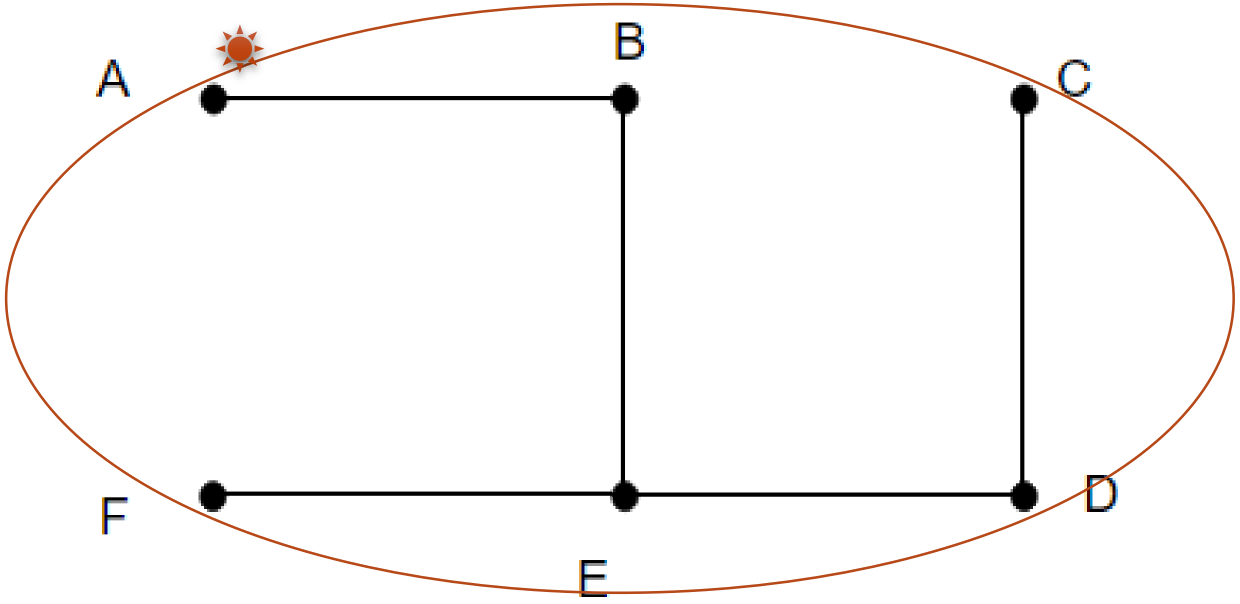




C.parent = A
D.parent = C
E.parent = F

Union Find – Naïve Version

union



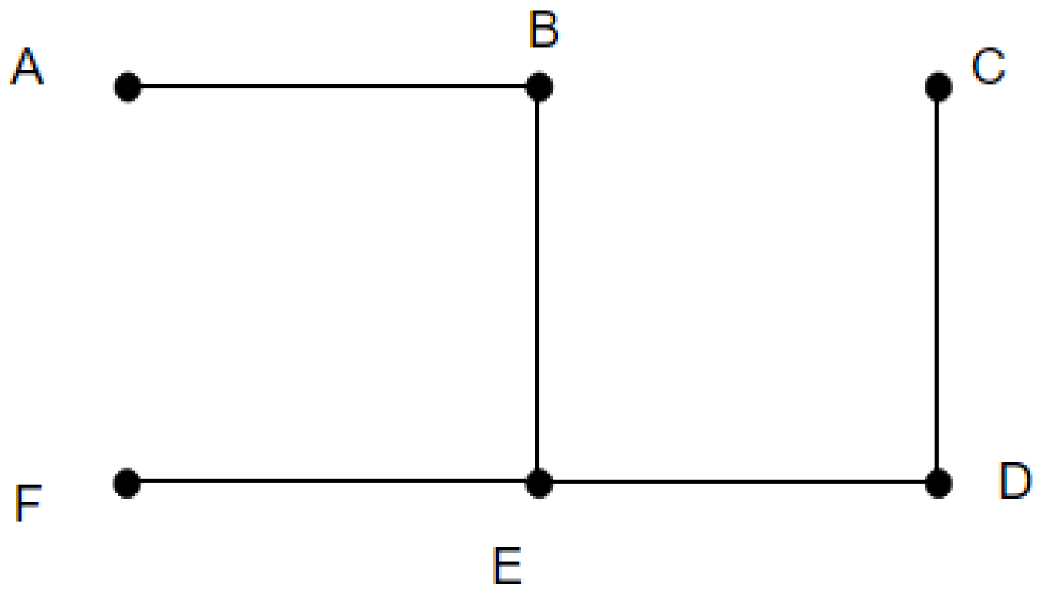

```
union(n1, n2):
        i = findSet(n1)
        j = findSet(n2)
        if i == j:
                return
        else:
                j.parent = i
```

Time Complexity

???

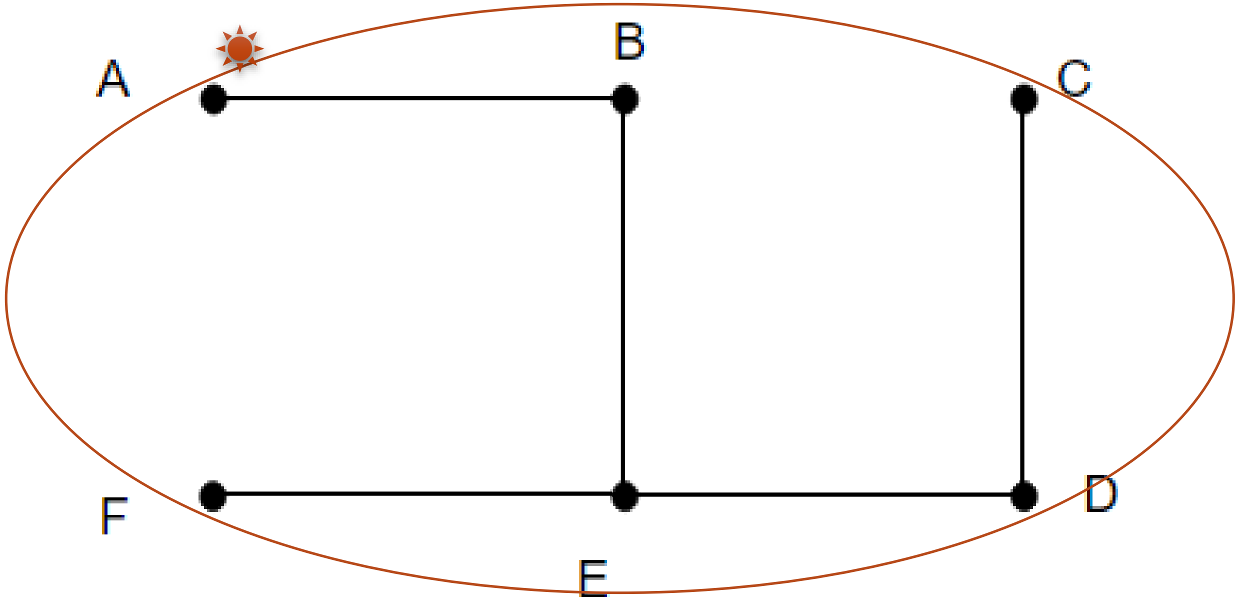
Union Find – Naïve Version

findSet



Union Find – Naïve Version

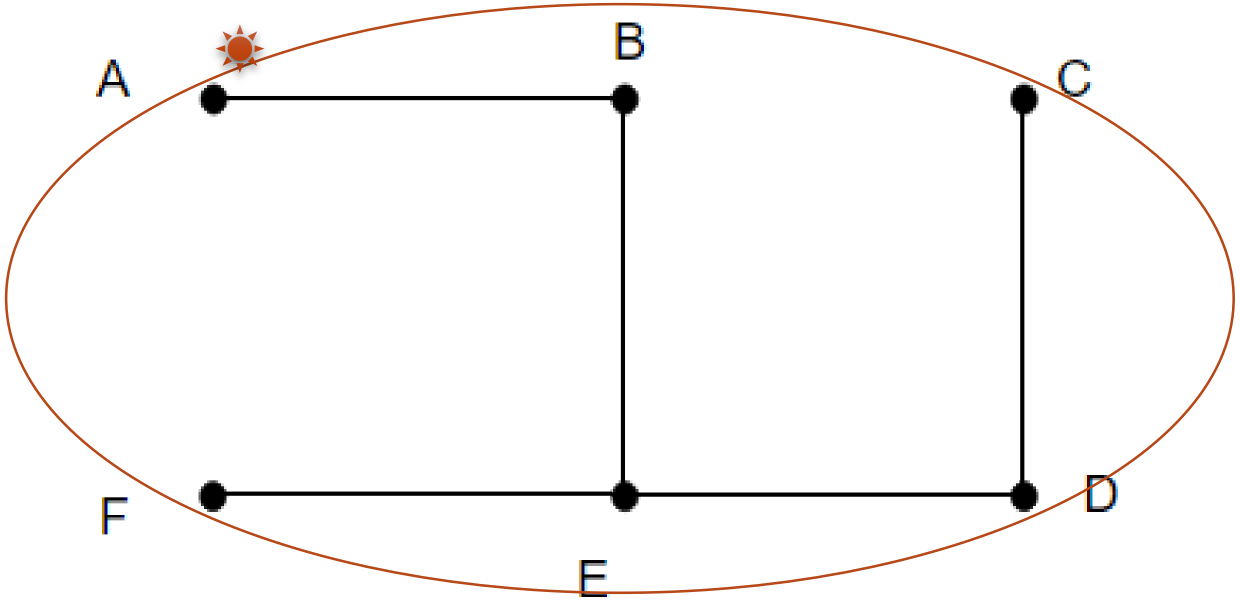
findSet



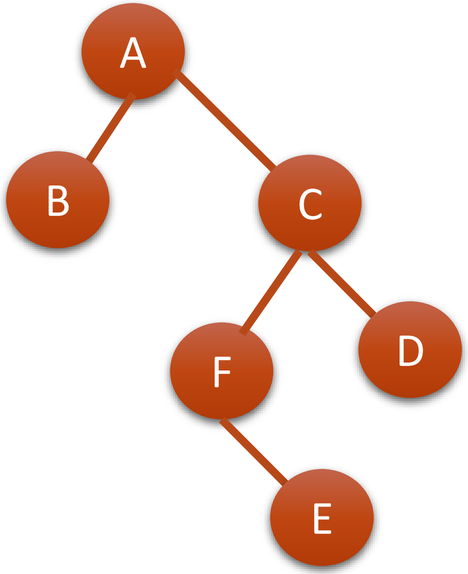

```
findSet(n):
    if n.parent == n:
        return n
    return findSet(n.parent)
```

Union Find – Naïve Version

findSet

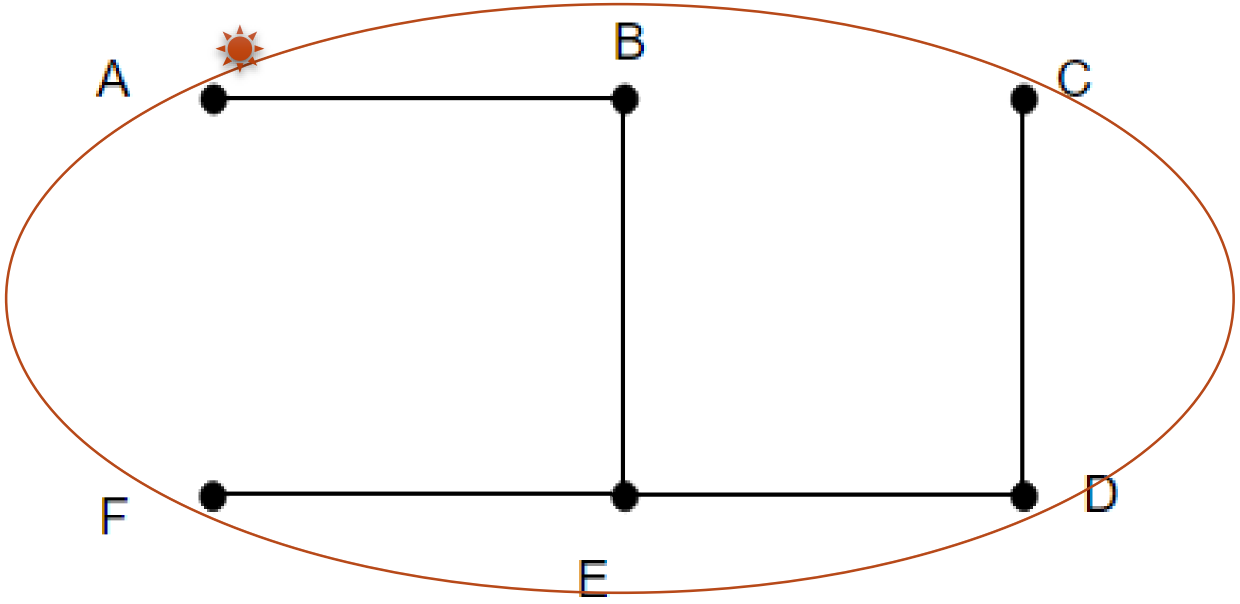



```
findSet(n):
    if n.parent == n:
        return n
    return findSet(n.parent)
```



Union Find – Naïve Version

findSet



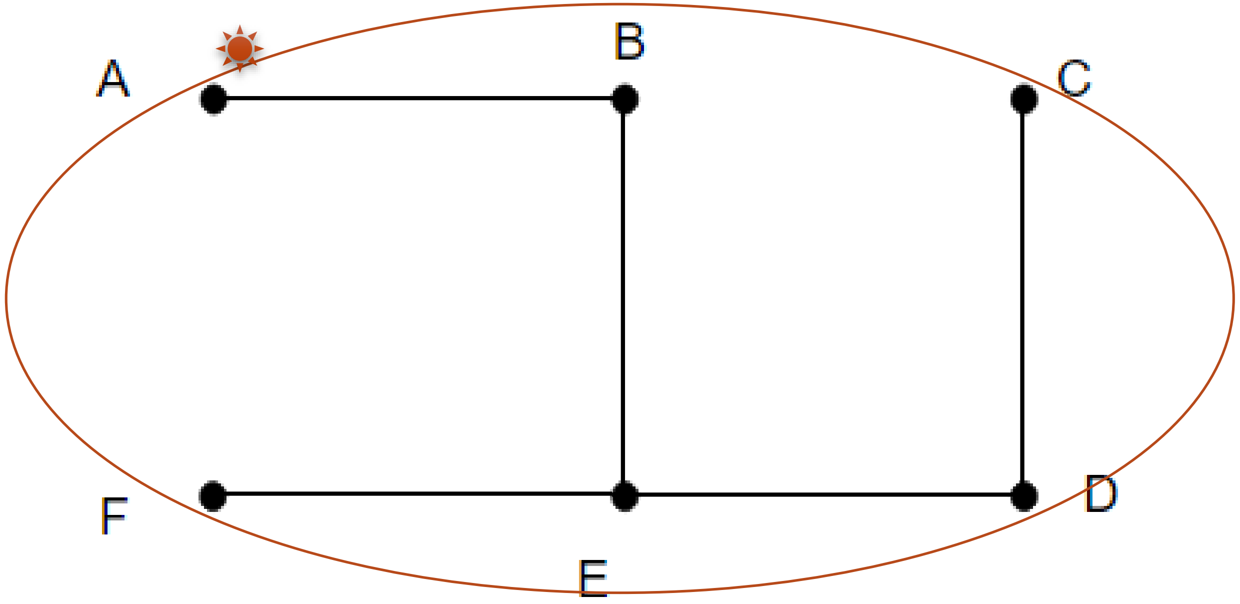

```
findSet(n):
    if n.parent == n:
        return n
    return findSet(n.parent)
```

Time Complexity

O(n)

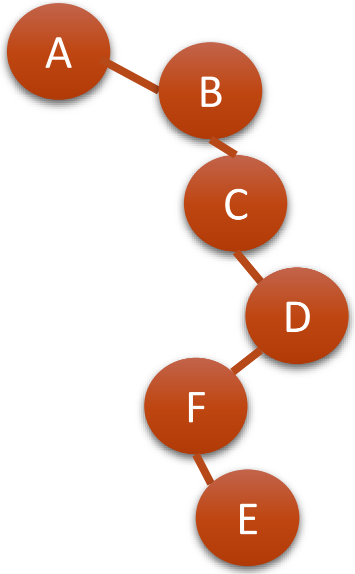
Union Find – Naïve Version

findSet



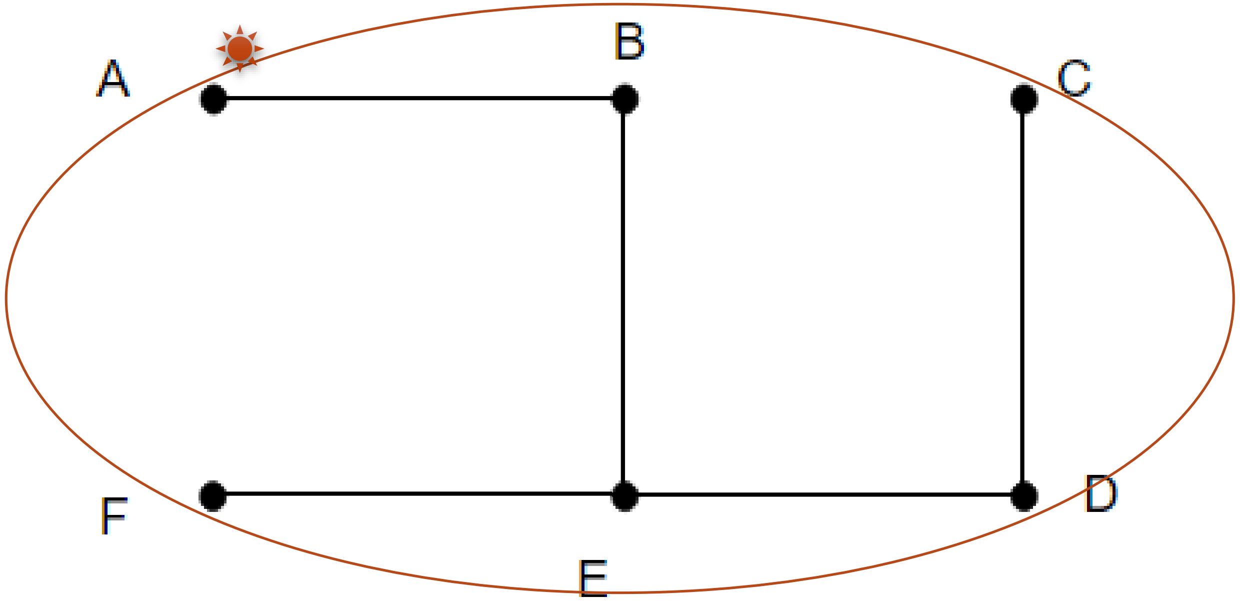

```
findSet(n):
    if n.parent == n:
        return n
    return findSet(n.parent)
```

Time Complexity
O(n)



Union Find – Naïve Version

union




```
union(n1, n2):
        i = findSet(n1)
        j = findSet(n2)
        if i == j:
                return
        else:
                j.parent = i
```

Time Complexity

O(n)

Union Find

Approaches

- Naïve Union Find
  - makeSet – O(1)
  - union – O(n)
  - findSet – O(n)

O(mn), where m stands for the number of operations

Structure

- Union Find
  - Naïve Version
    - *Practice*
  - Optimized Version
    - *Practice*
- Cycle Detection
  - *Practice*
- Kruskal's algorithm

Q1. Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:

```
0          3
|          |
1 —— 2     4
```

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.

Q1. Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 2:

```
0          4
|          |
1 — 2 — 3
```

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1

Q1. Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 2:

```
0          4
|          |
1 — 2 — 3
```

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1

```
def countComponents(self, n, edges):
        """
        :type n: int
        :type edges: List[List[int]]
        :rtype: int
        """
```

Q1. Number of Connected Components in an Undirected Graph

```
Class Union(object):
      def __inint__(self):
            # … …
            self.count = 0

      def makeSet(self, v):
            # … …
            self.count += 1

      def union(self, n1, n2):
            # … …
            self.count -= 1
```

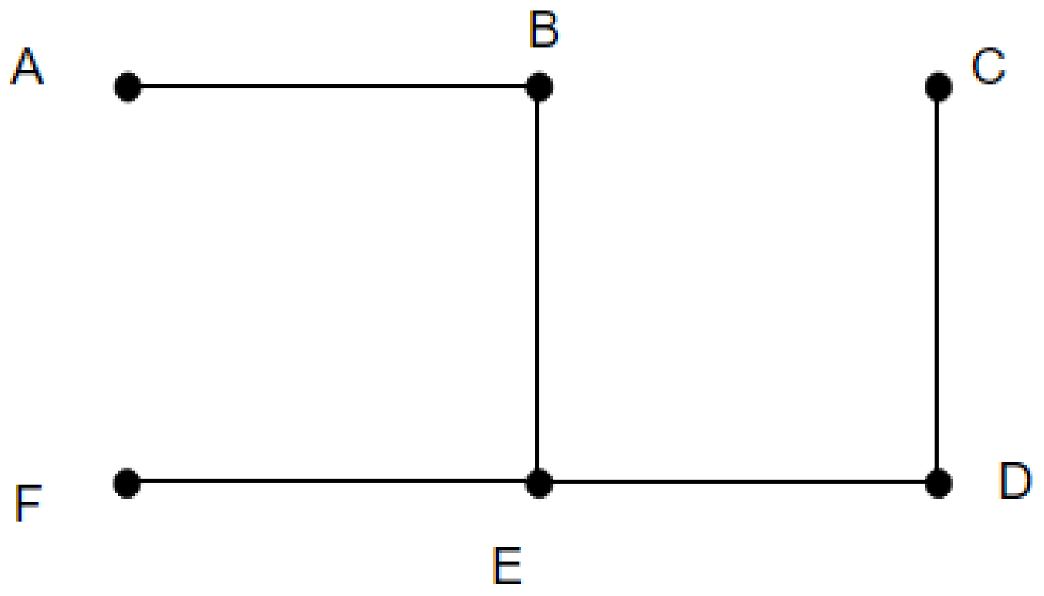
Q1. Number of Connected Components in an Undirected Graph

```
Class Union(object):
      def __inint__(self):
            # … …
            self.count = 0

      def makeSet(self, v):
            # … …
            self.count += 1

      def union(self, n1, n2):
            # … …
            self.count -= 1
```

```
union(self, n1, n2):
      i = findSet(n1)
      j = findSet(n2)
      if i == j:
            return
      else:
            j.parent = I
      self.count -= 1
```

Q1. Number of Connected Components in an Undirected Graph

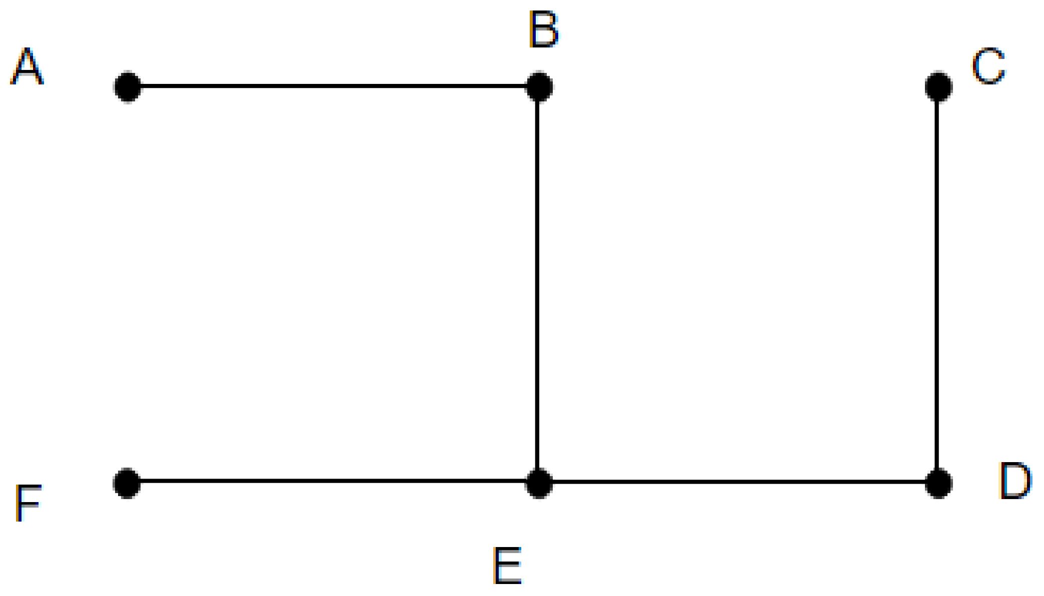
Example of count



makeSet(A)
makeSet(B)
……
makeSet(F)
union(A, B)
union(B, E)
union(A, E)

Q1. Number of Connected Components in an Undirected Graph

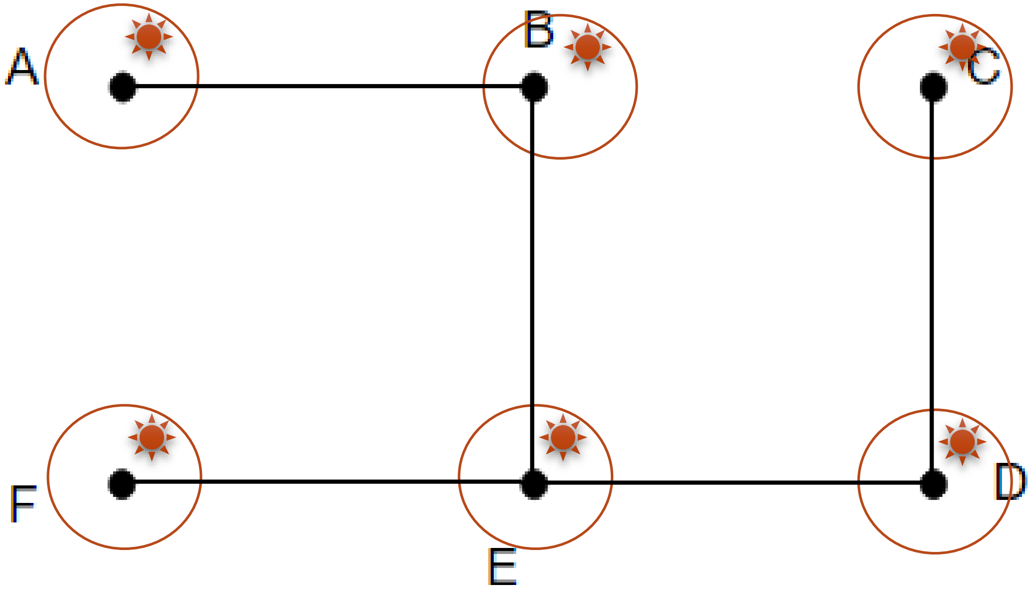
Example of count



makeSet(A)
makeSet(B)
……
makeSet(F)
union(A, B)
union(B, E)
union(A, E)

Q1. Number of Connected Components in an Undirected Graph

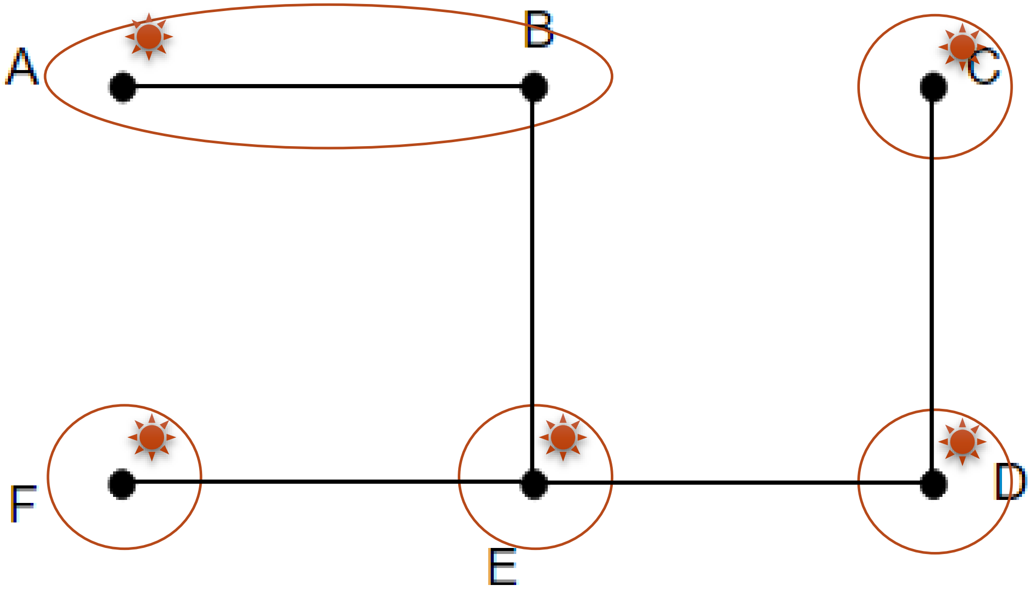
Example of count



makeSet(A)
makeSet(B)
……
makeSet(F) → Count = 6
union(A, B)
union(B, E)
union(A, E)

Q1. Number of Connected Components in an Undirected Graph

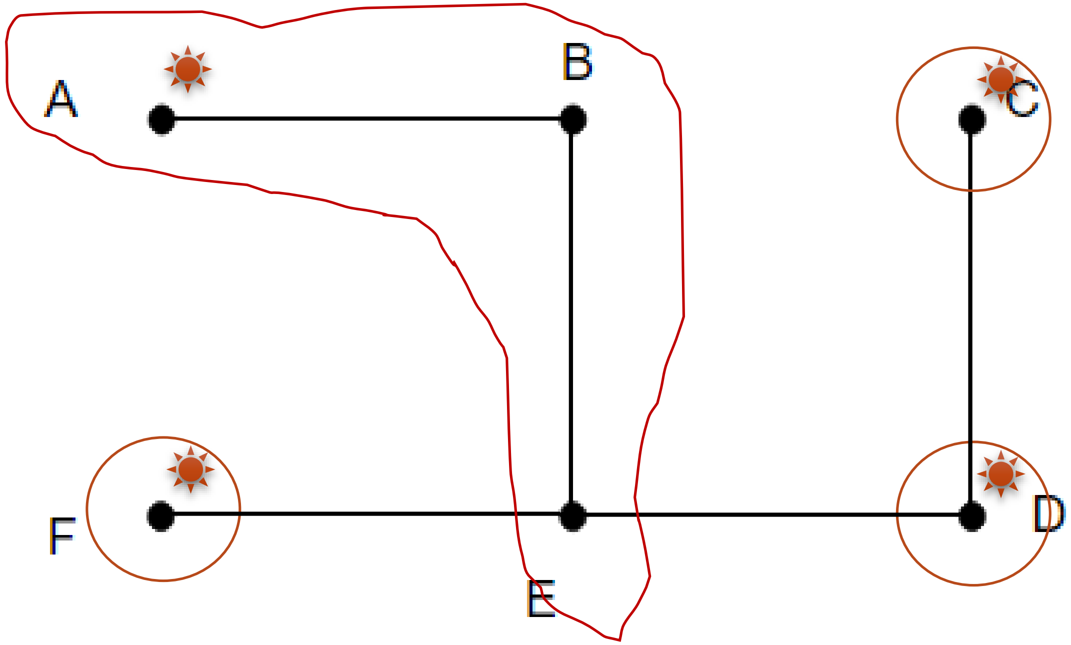
Example of count



makeSet(A)
makeSet(B)
……
makeSet(F) → Count = 6
union(A, B) → Count = 5
union(B, E)
union(A, E)

Q1. Number of Connected Components in an Undirected Graph

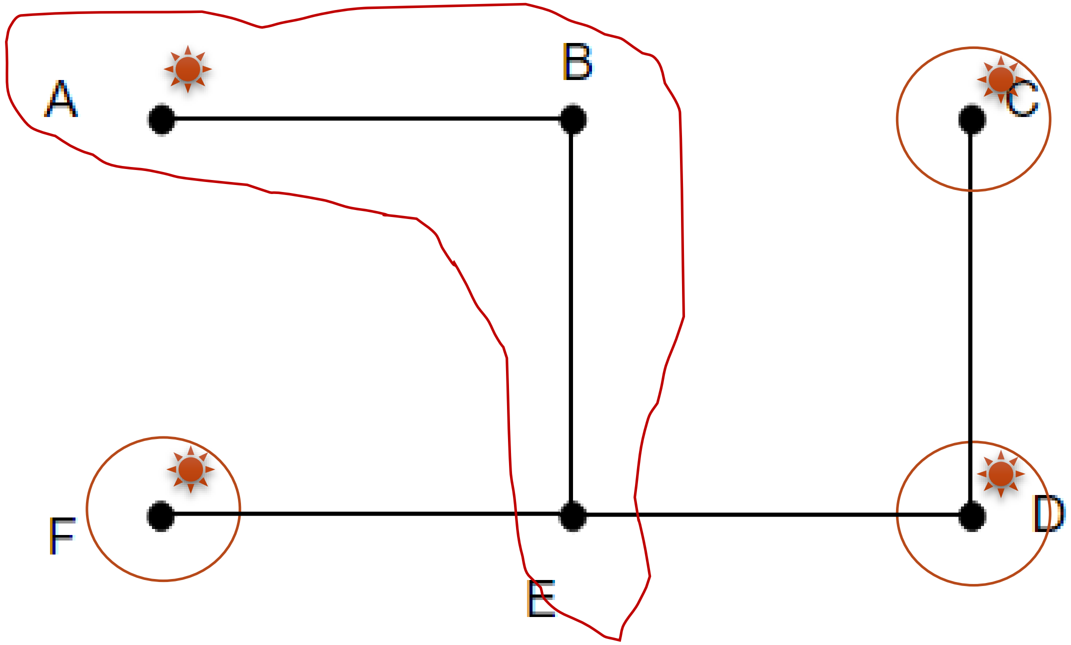
Example of count



makeSet(A)
makeSet(B)
……
makeSet(F) → Count = 6
union(A, B) → Count = 5
union(B, E) → Count = 4
union(A, E)

Q1. Number of Connected Components in an Undirected Graph

Example of count



makeSet(A)
makeSet(B)
……
makeSet(F) → Count = 6
union(A, B) → Count = 5
union(B, E) → Count = 4
union(A, E) → Count = 4

Q1. Number of Connected Components in an Undirected Graph

```
def countComponents(self, n, edges):

    union = Union()

    for i in range(n):
        union.makeSet(i)
    for i, j in edges:
        union.union(i, j)

    return union.count
```

Example 1:

```
 0        3
 |        |
 1 —2     4
```

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.

Structure

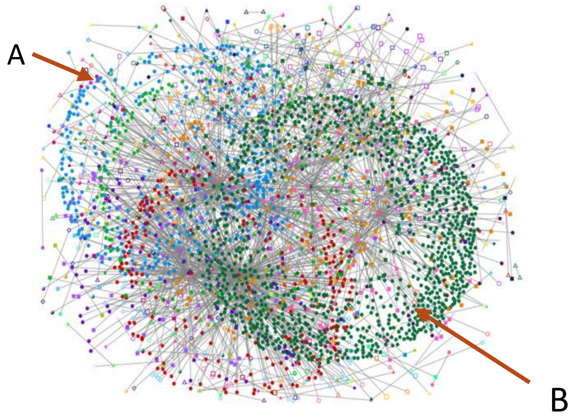
- Union Find
  - Naïve Version
    - *Practice*
  - Optimized Version
    - *Practice*
- Cycle Detection
  - *Practice*
- Kruskal's algorithm

Union Find

Approaches

- Naïve Union Find
  - makeSet – O(1)
  - union – O(n)
  - findSet – O(n)

O(mn), where m stands for the number of operations



Graph: 5000 vertices
Union find: 5 operations

Max steps: 25000

Union Find – Optimized Version

Goals:

- makeSet – O(1)
- union – O(n) → O(X)
- findSet – O(n) → O(X)

where x is a constant

Union Find – Optimized Version

Optimizations:

- Union by rank
- Path compression

Union Find – Optimized Version

Optimizations:

- Union by rank

    Rule for merging set A and B:

    if A is larger or equal to B:

        merge B into A

    else:

        merge A into B

Union Find – Optimized Version

Optimizations:

- Union by rank

    Rule for merging set A and B:

    if A is larger or equal to B:

    merge B into A

    else:

    merge A into B

Time Complexity:

- findSet(v) – O(lg n)
- union(v) – O(lg n)

Union Find – Optimized Version

Optimizations:

- Union by rank

  Rule for merging set A and B:

  if A is larger or equal to B:

  merge B into A

  else:

  merge A into B

A, B, C, D, E, F, G, H

Union Find – Optimized Version

Optimizations:

- Union by rank

  Rule for merging set A and B:

  if A is larger or equal to B:

  merge B into A

  else:

  merge A into B

A, B, C, D, E, F, G, H

Union Find – Optimized Version

Optimizations:

- Union by rank

  Rule for merging set A and B:

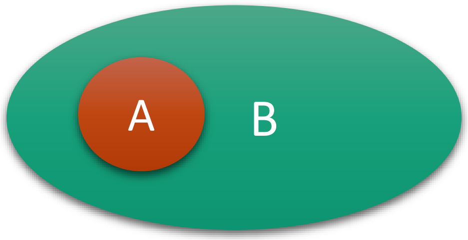
  if A is larger or equal to B:

  merge B into A

  else:

  merge A into B

A, B, C, D, E, F, G, H



Union Find – Optimized Version

Optimizations:

- Union by rank

Rule for merging set A and B:

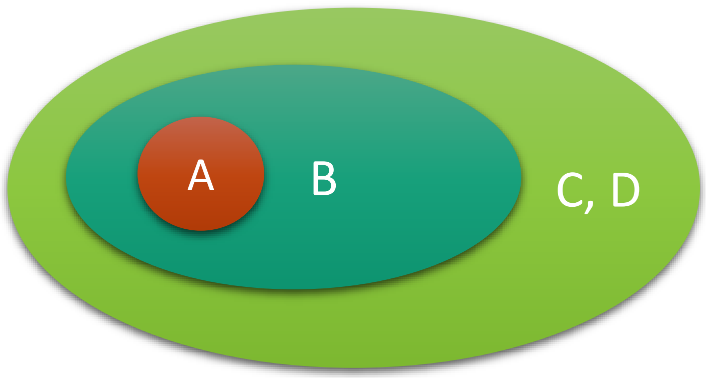
if A is larger or equal to B:

merge B into A

else:

merge A into B

A, B, C, D, E, F, G, H



Union Find – Optimized Version

Optimizations:

- Union by rank

    Rule for merging set A and B:

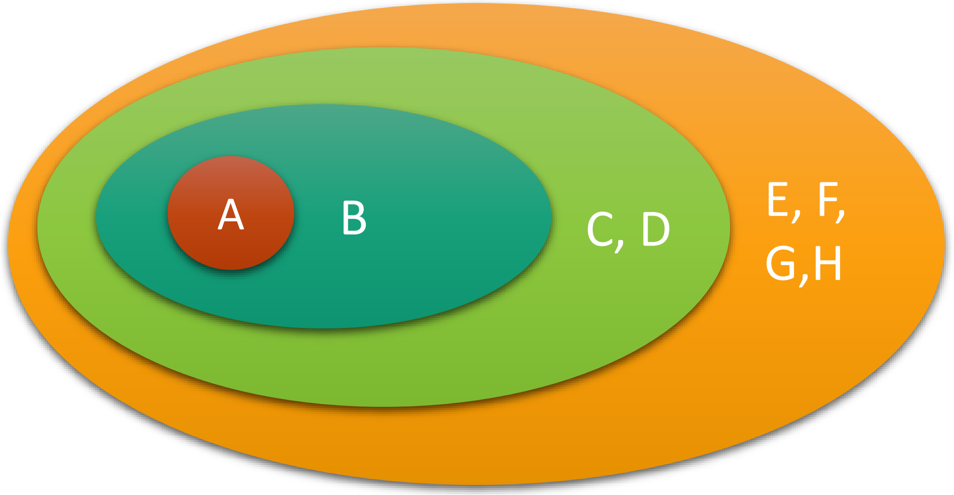
    if A is larger or equal to B:

    merge B into A

    else:

    merge A into B

A, B, C, D, E, F, G, H



Union Find – Optimized Version

Optimizations:

- Union by rank

Rule for merging set A and B:

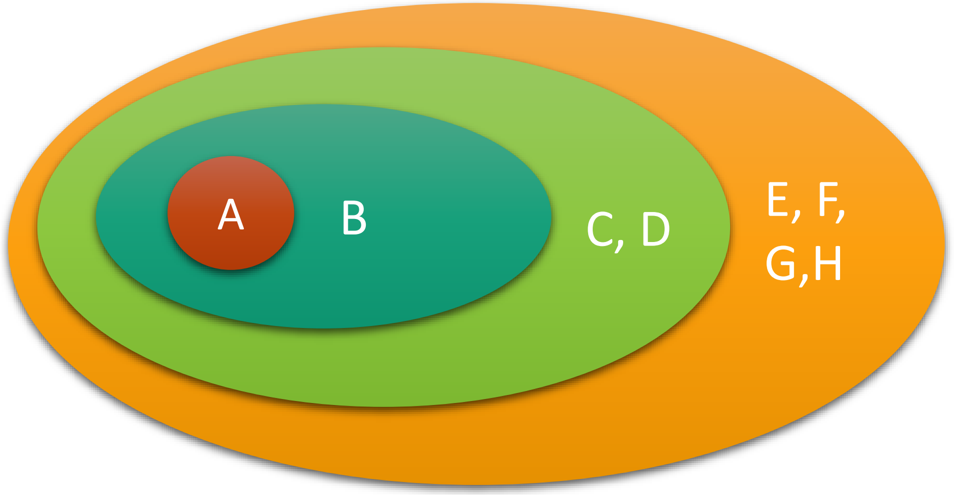
if A is larger or equal to B:

merge B into A

else:

merge A into B

A, B, C, D, E, F, G, H



How many times can a number starting from 1 double itself before reaching n?   --- O(lg n)

Union Find – Optimized Version

Optimizations:

- Union by rank

    Rule for merging set A and B:

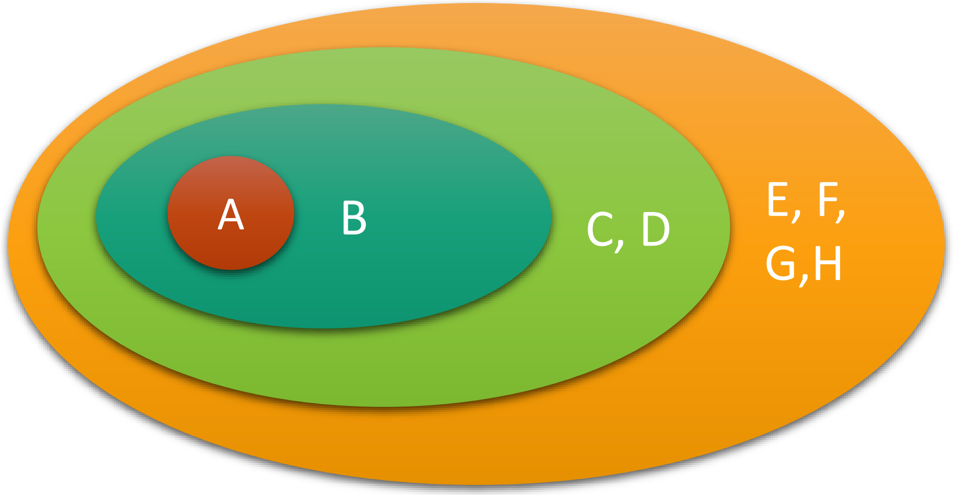
    if A is larger or equal to B:

    merge B into A

    else:

    merge A into B

A, B, C, D, E, F, G, H



How many times can a number starting from 1 double itself before reaching n? --- O(lg n)

findSet(v) – O(lg n)

union(v) – O(lg n)

Union Find – Optimized Version

Optimizations:

- Union by rank

Rule for merging set A and B:

if A is larger or equal to B:

merge B into A

else:

merge A into B

Union Find – Optimized Version

Optimizations:

- Union by rank

  Rule for merging set A and B:

  if A is larger or equal to B:

  merge B into A

  else:

  merge A into B

```
Node:
    val
    parent
    rank

makeSet(v):
    n = Node(v)
    n.parent = n
    n.rank = 0
```

Union Find – Optimized Version

Optimizations:

- Union by rank

    Rule for merging set A and B:

    if A is larger or equal to B:

    merge B into A

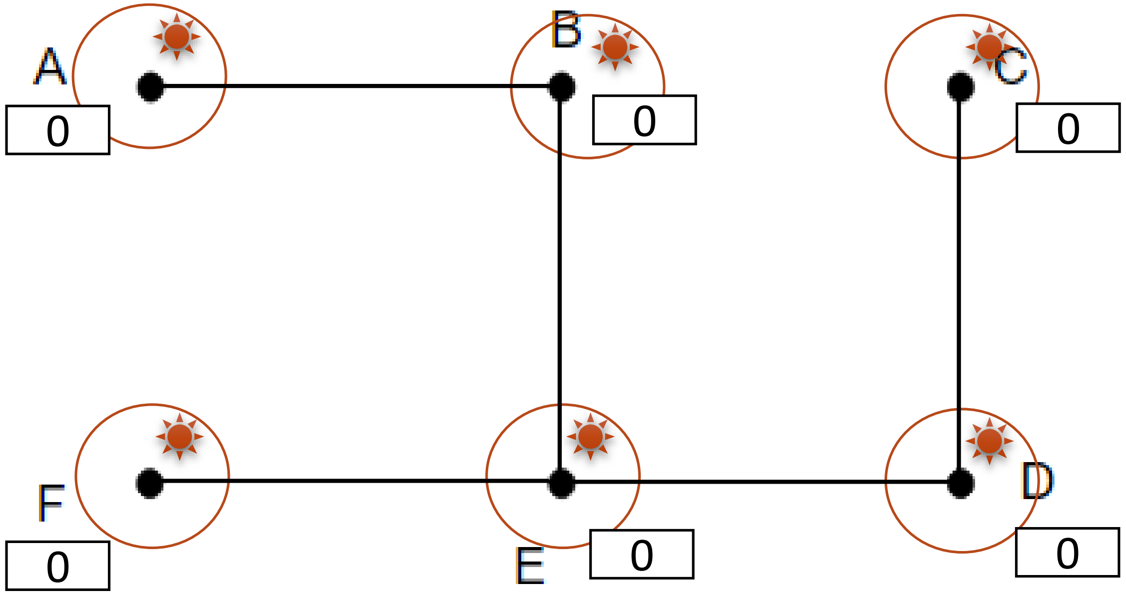
    else:

    merge A into B

```
union(n1, n2):
        i = findSet(n1)
        j = findSet(n2)
        if i == j: return

        if i.rank > j.rank:
             j.parent = i
         elif i.rank < j.rank:
             i.parent = j
         else:
             j.parent = i
             i.rank += 1
```

Union Find – Optimized Version

union

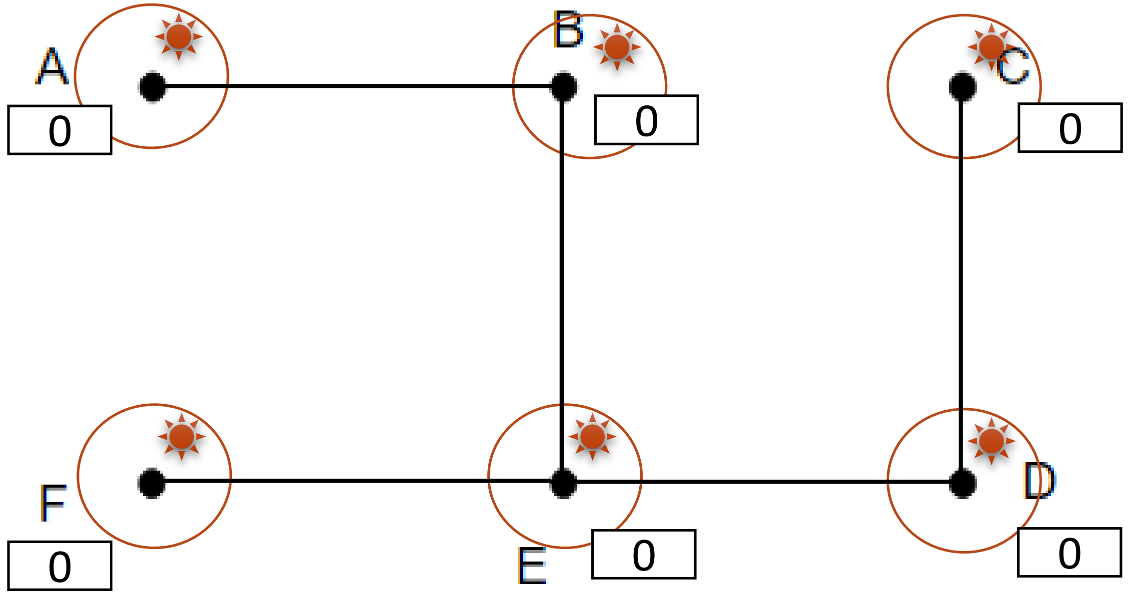


union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

Union Find – Optimized Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

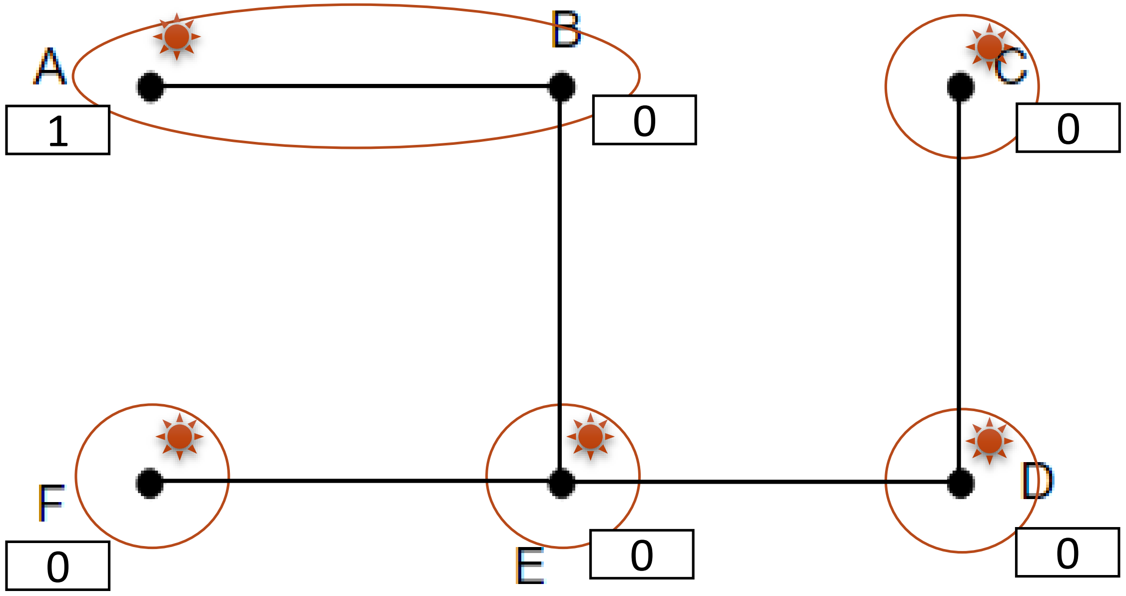
union

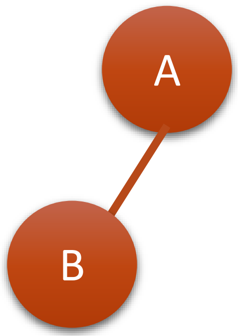


Union Find – Optimized Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

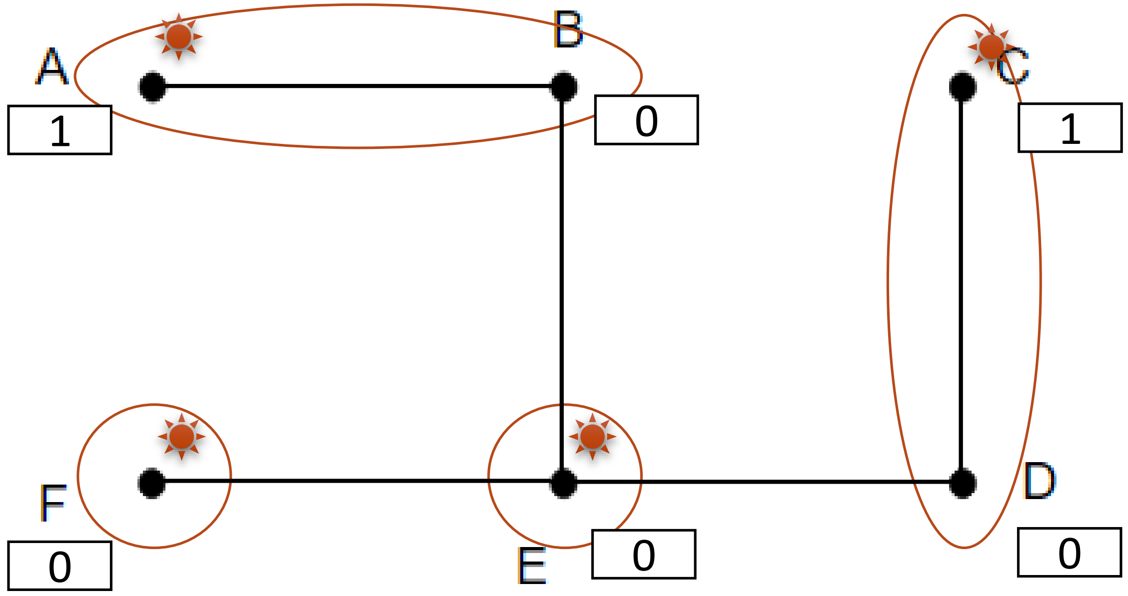


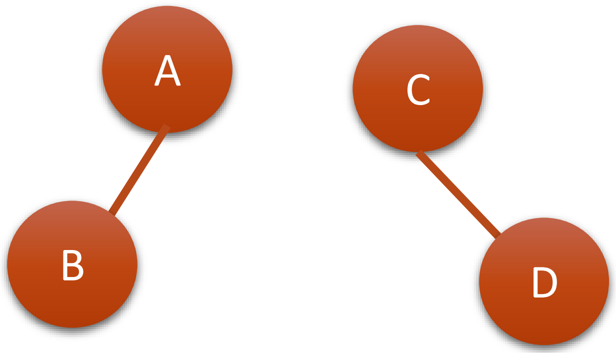


Union Find – Optimized Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

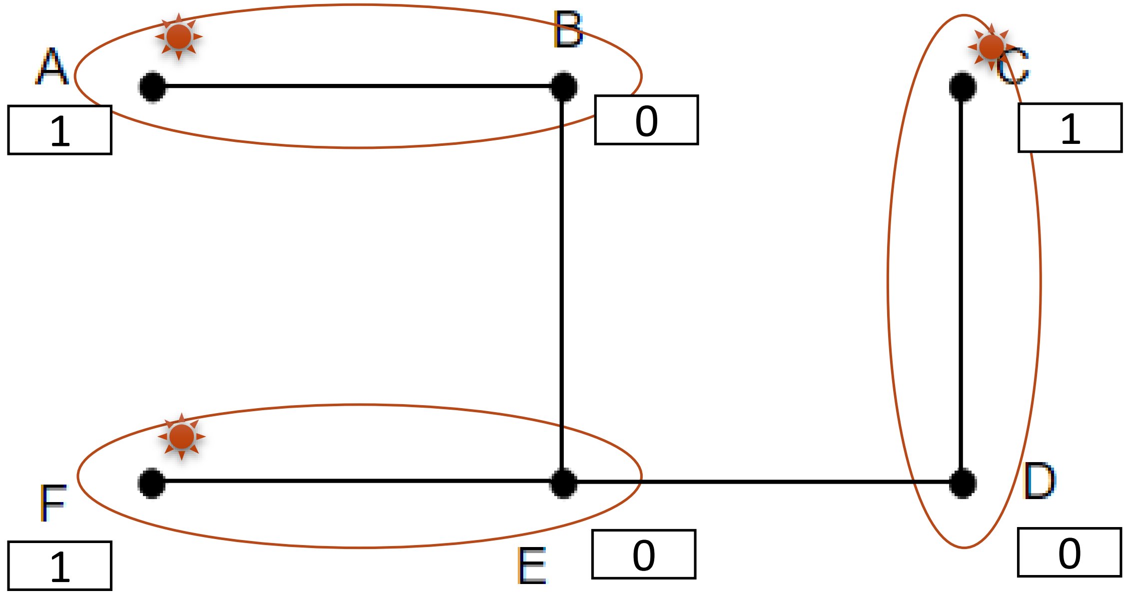


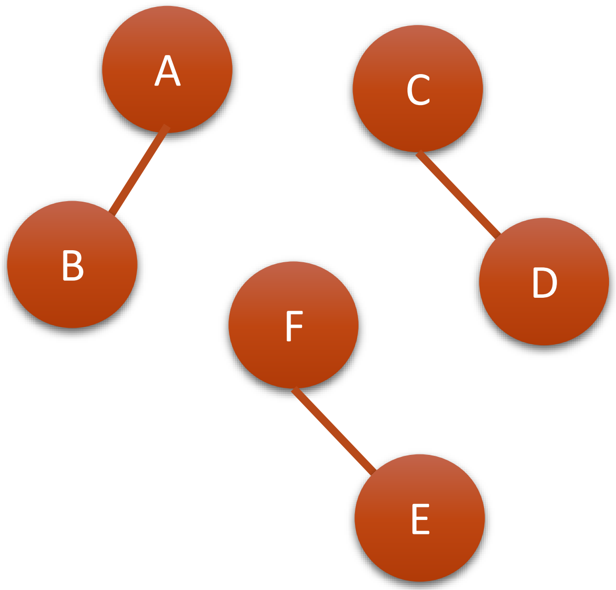


Union Find – Optimized Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

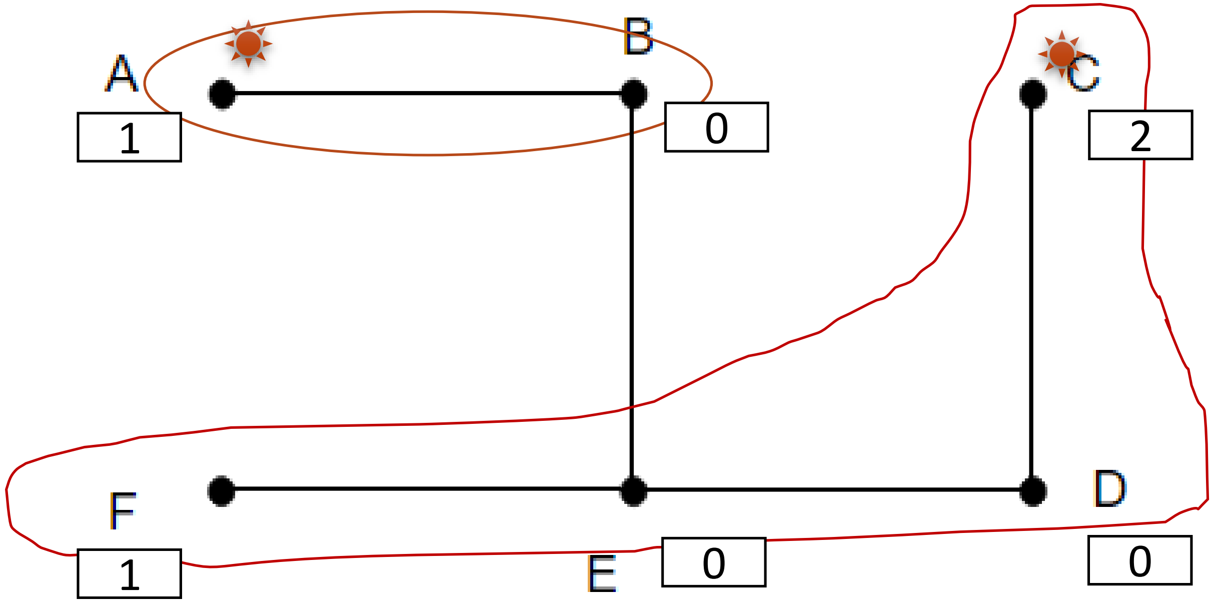


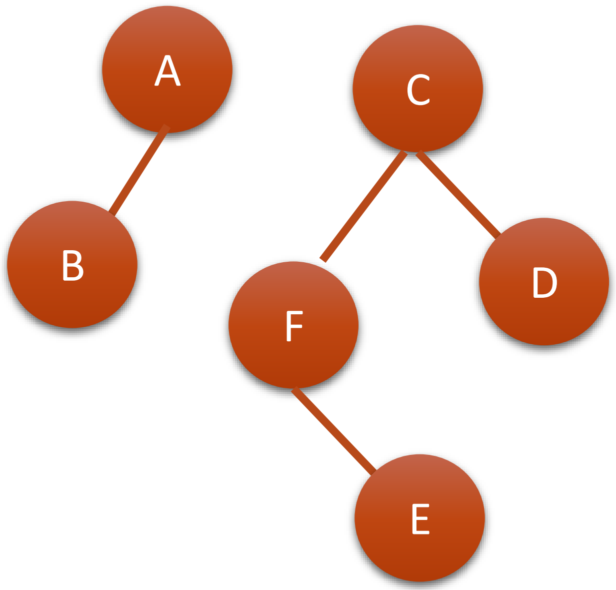


Union Find – Optimized Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

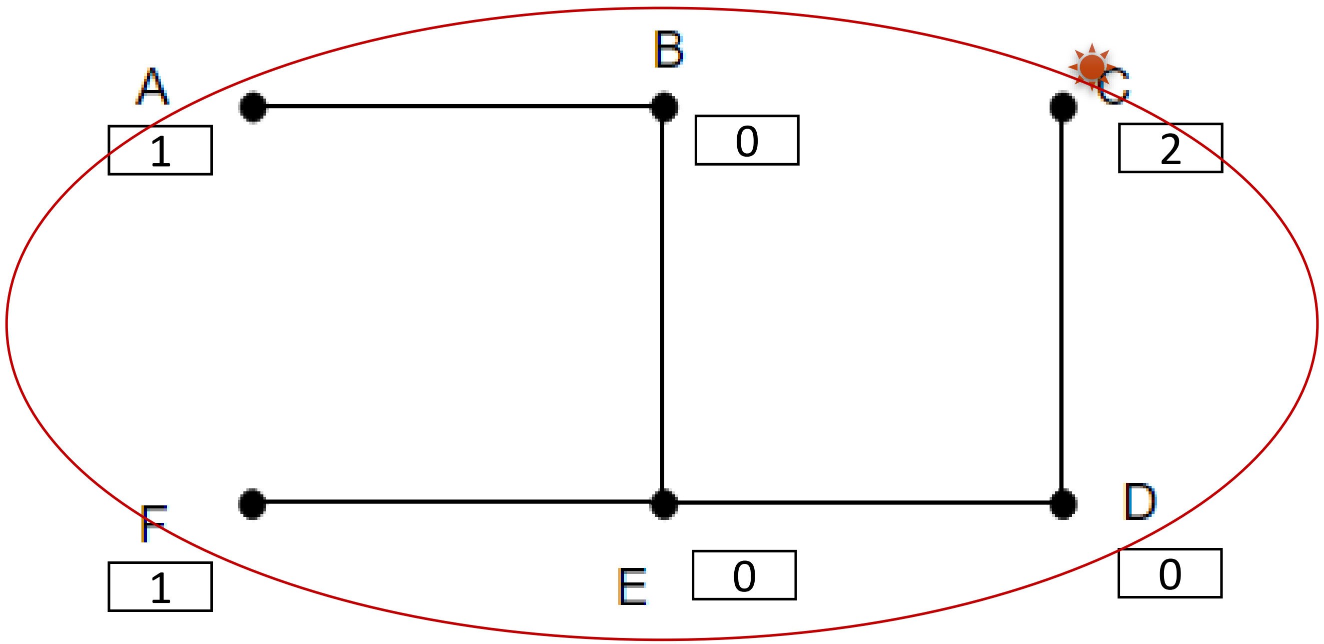


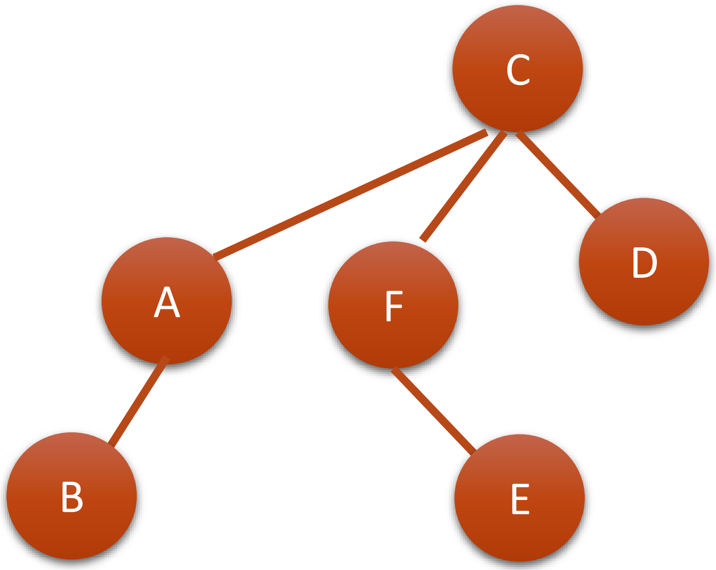


Union Find – Optimized Version

union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union

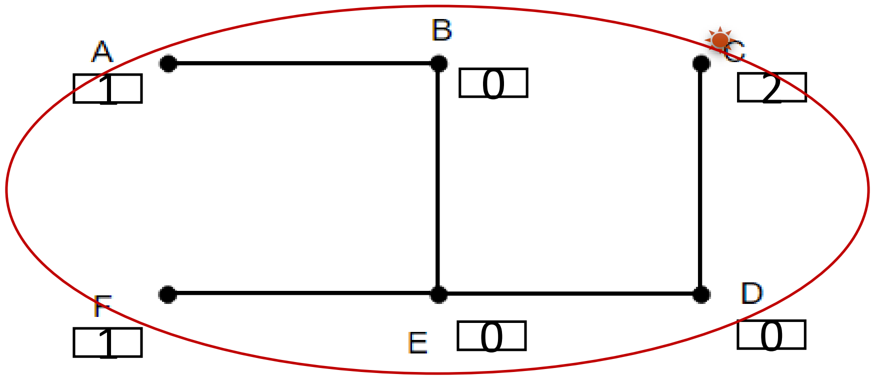


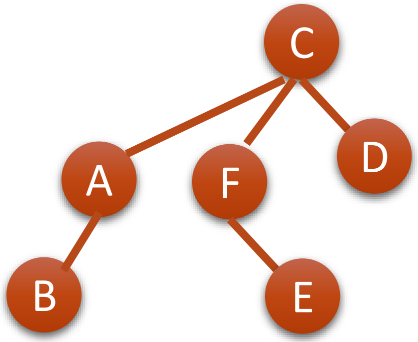


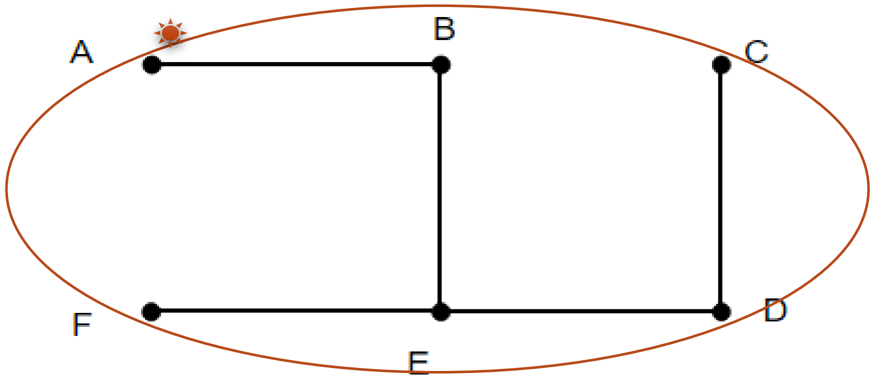
Union Find – Optimized Version

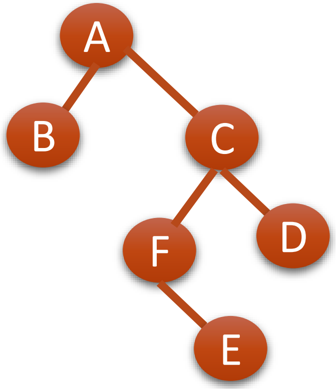
union(A, B)
union(C, D)
union(F, E)
union(D, E)
union(B, E)

union









Union Find – Optimized Version

Optimizations:

- Union by rank
- Path compression

Union Find – Optimized Version

Optimizations:

- Union by rank
- Path compression

    Flatten the structure of the tree whenever *findSet* is used.

Union Find – Optimized Version

Optimizations:

- Union by rank
- Path compression

  Flatten the structure of the tree whenever *findSet* is used.

```
findSet(n):
    if n.parent == n:
        return n
    return findSet(n.parent)
```

```
findSet(n):
    if n.parent != n:
        n.parent = findSet(n.parent)
     return n.parent
```

Union Find – Optimized Version

Amortized time per operation:

- O(a(n)), where a(n) is the inverse of the function n = f(x, x)
- f(x, x) is Ackermann function
- a(n) is less than 5 for all remotely practical values of n.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

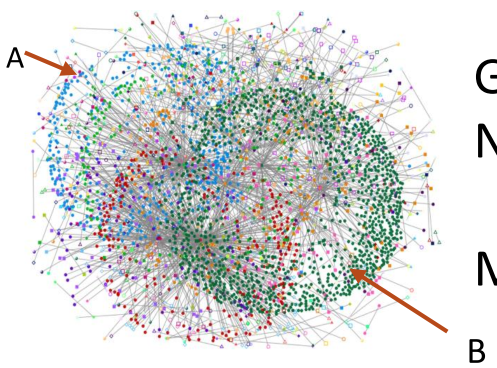
Union Find

Approaches

- Naïve Union Find
- Optimized Union Find

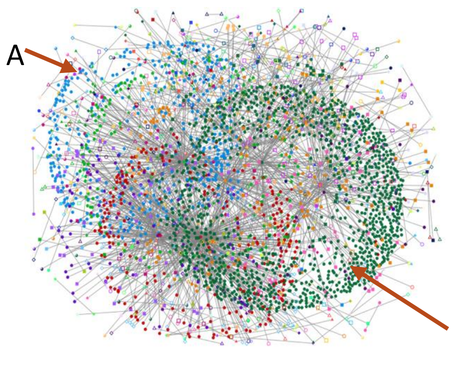
O(mn), where m stands for the number of operations

O(m* a(n)), where m stands for the number of operations



Graph: 5000 vertices
Naïve union find: 5 operations

Max steps: 25000



Graph: 5000 vertices
Optimized union find: 5 operations

Max steps: 25

Structure

- Union Find
  - Naïve Version
    - *Practice*
  - Optimized Version
    - *Practice*
- Cycle Detection
  - *Practice*
- Kruskal's algorithm

Q2. Number of Islands

A 2d grid map of m rows and n columns is initially filled with water. We may perform an *addLand* operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each *addLand* operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Q2. Number of Islands

Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]].

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \rightarrow \begin{matrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \rightarrow \begin{matrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \rightarrow \begin{matrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix}$$

1 1 2

Return [1, 1, 2]

Q2. Number of Islands

Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]].

0 0 0		1 0 0		1 1 0		1 1 0		1 1 1
0 0 0	→	0 0 0	→	0 0 0	→	0 0 1	→	0 0 1
0 0 0		0 0 0		0 0 0		0 0 0		0 0 0
		1		1		2		1

Return [1, 1, 2, 1]

Q2. Number of Islands

	1	
0	**1**	1
	0	

(1, 1)

Q2. Number of Islands

	1	
0	**1**	1
	0	

(1, 1)
Neighbors: (0, 1), (2, 1), (1, 0), (1, 2)

Q2. Number of Islands

	1	
0	**1**	1
	0	

(1, 1)
Neighbors: (0, 1), (2, 1), (1, 0), (1, 2)

(x, y)

Q2. Number of Islands

	1	
0	**1**	1
	0	

(1, 1)
Neighbors: (0, 1), (2, 1), (1, 0), (1, 2)

(x, y)
Neighbors: (x-1, y), (x+1, y), (x, y+1), (x, y-1)

Q2. Number of Islands

```
def numIslands2(self, m, n, positions):
        islands = Union()
        result = []
        for i, j in positions:
                islands.makeSet((i,j))
                neighbors = [(i+1,j), (i-1,j), (i,j+1), (i,j-1)]
                for x, y in neighbors:
                        if (x,y) in islands.table:
                                islands.union((x,y),(i,j))
                result.append(islands.count)
         return result
```

Structure

- Union Find
  - Naïve Version
    - *Practice*
  - Optimized Version
    - *Practice*
- Cycle Detection
  - *Practice*
- Kruskal's algorithm

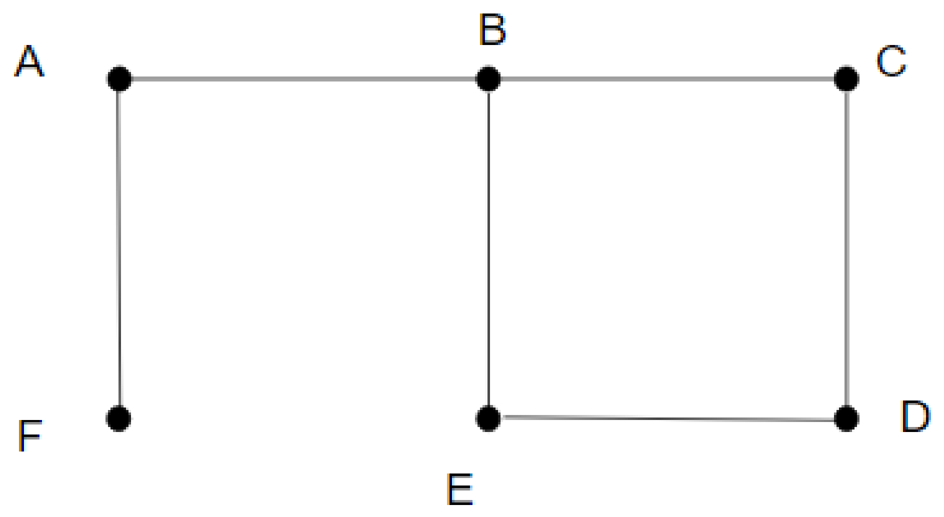
Cycle Detection

Cycle Detection

Cycle detection refers to the algorithmic problem of finding a cycle in a sequence of iterated function values.

Cycle Detection

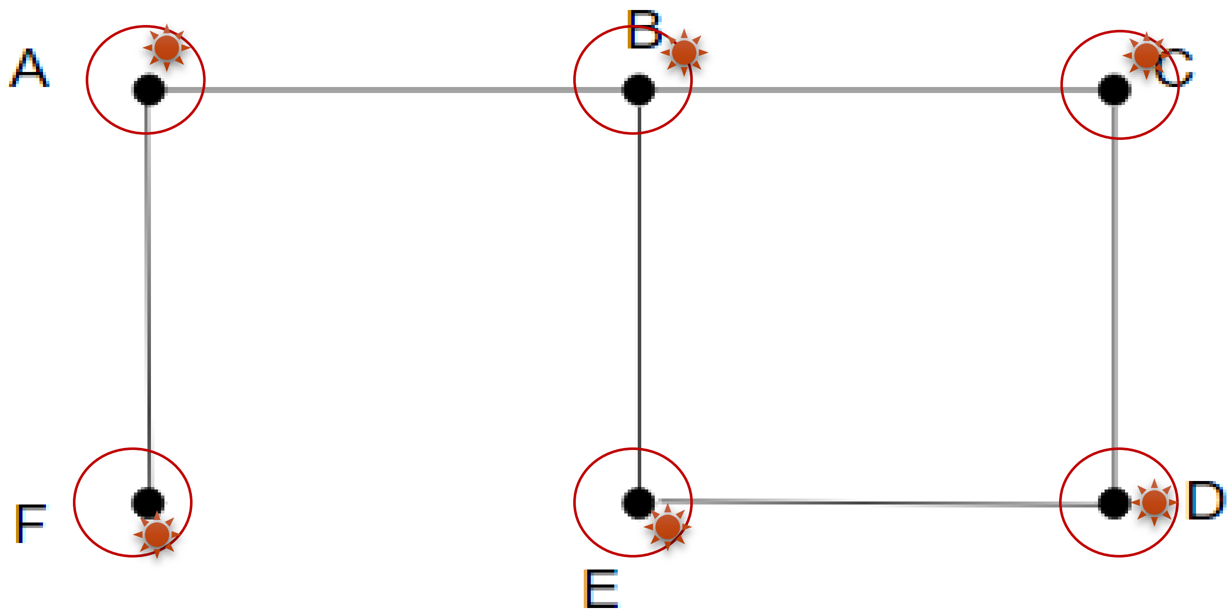
Cycle detection refers to the algorithmic problem of finding a cycle in a sequence of iterated function values.



Cycle Detection

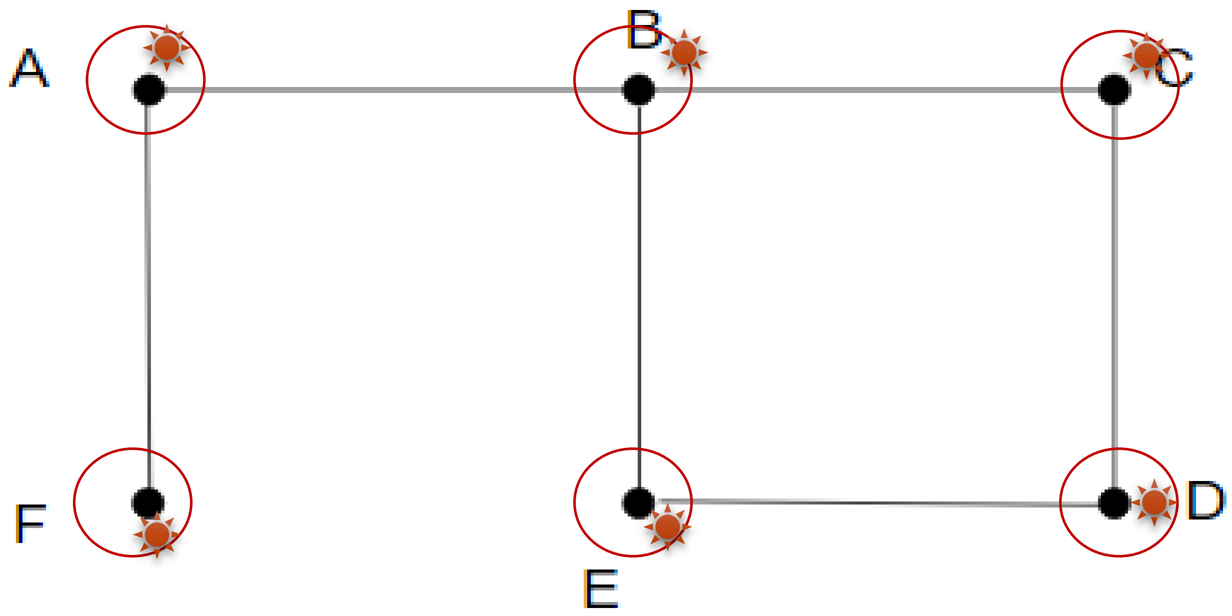
```
cycleDetect(G):
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E:
        if FIND-SET(u) ≠ FIND-SET(v):
            UNION(u, v)
        else:
            return True
    return False
```

Cycle Detection – Step 1



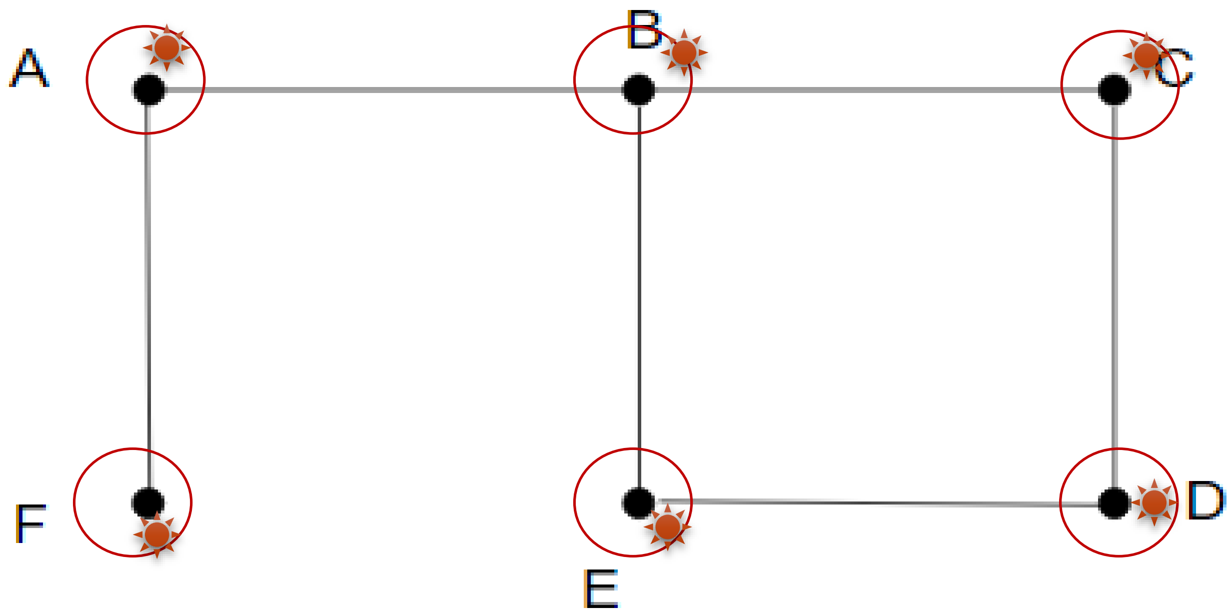
Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE



Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

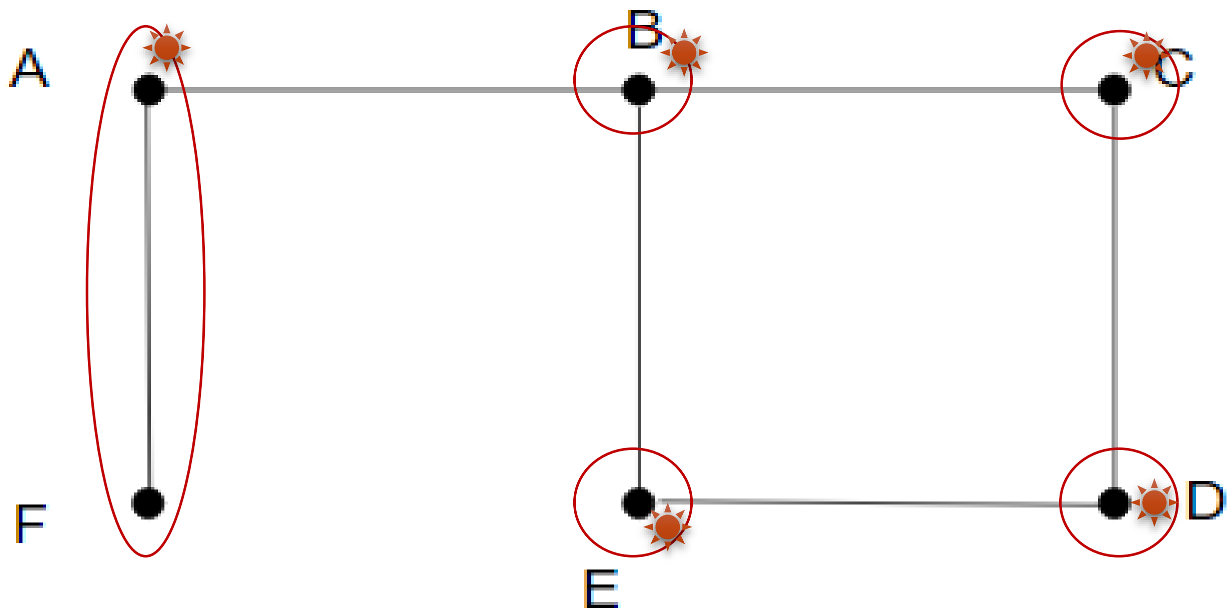


AF

findSet(A) **findSet(F)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

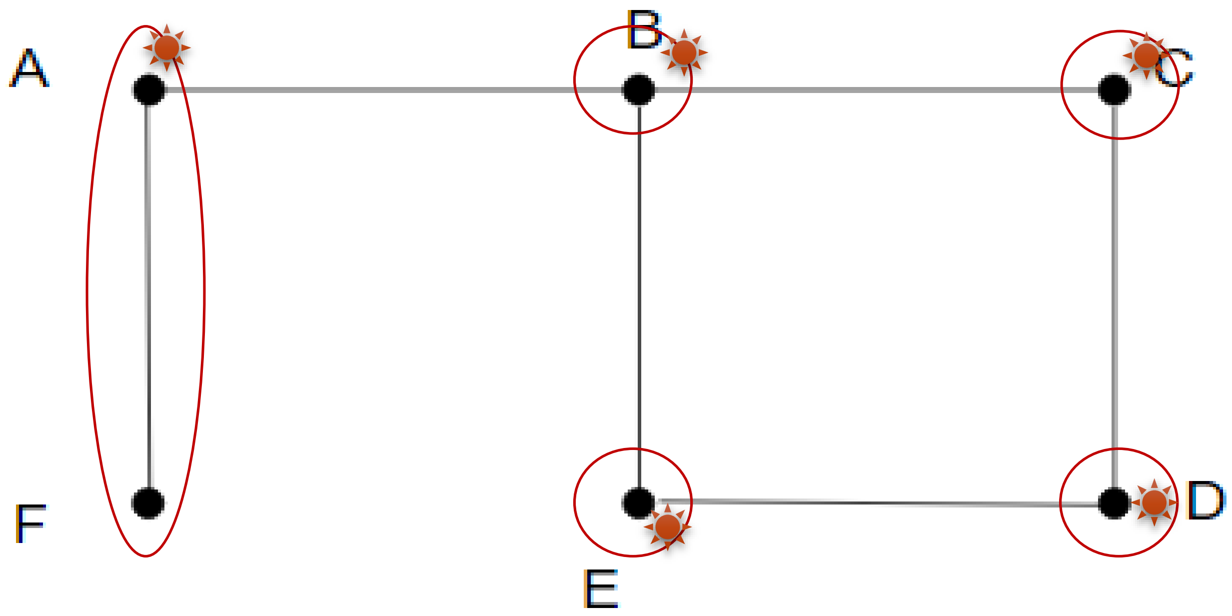


AF

findSet(A) **findSet(F)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

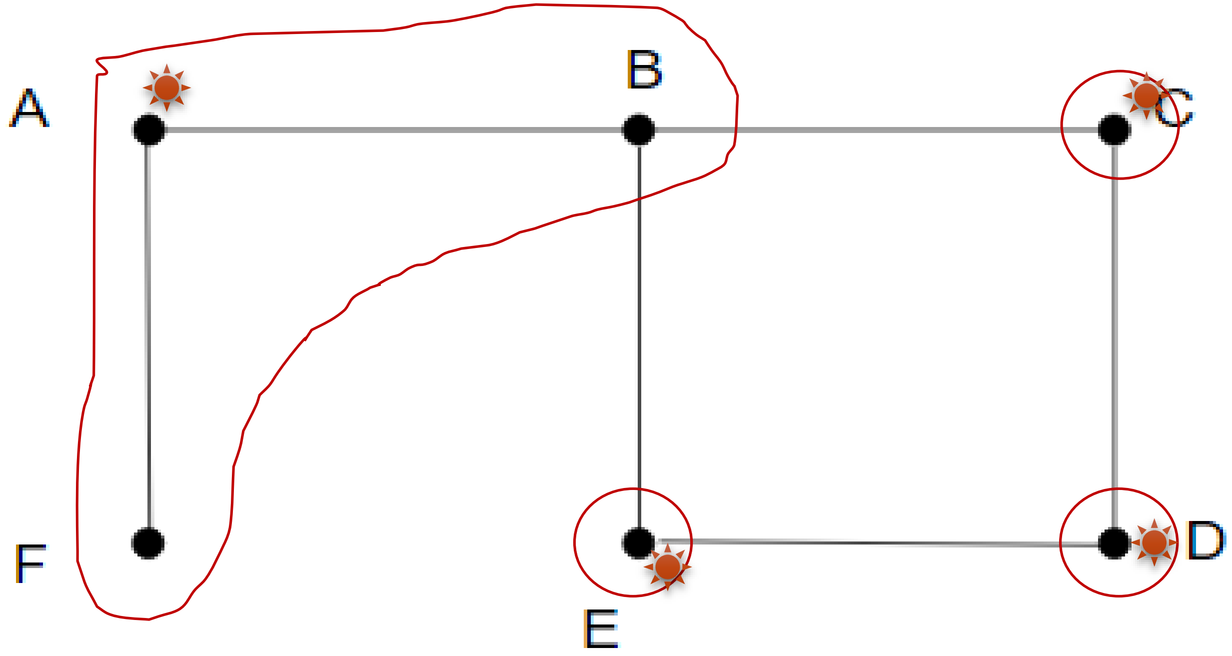


AB

findSet(A) **findSet(B)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

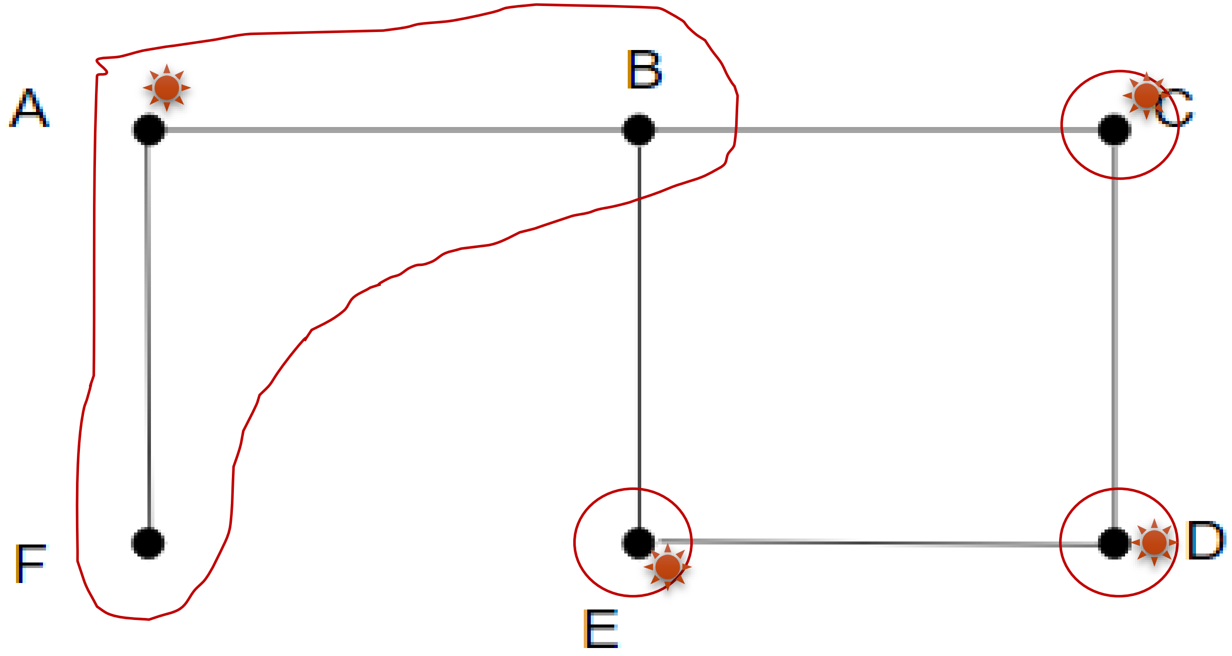


AB

findSet(A) **findSet(B)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

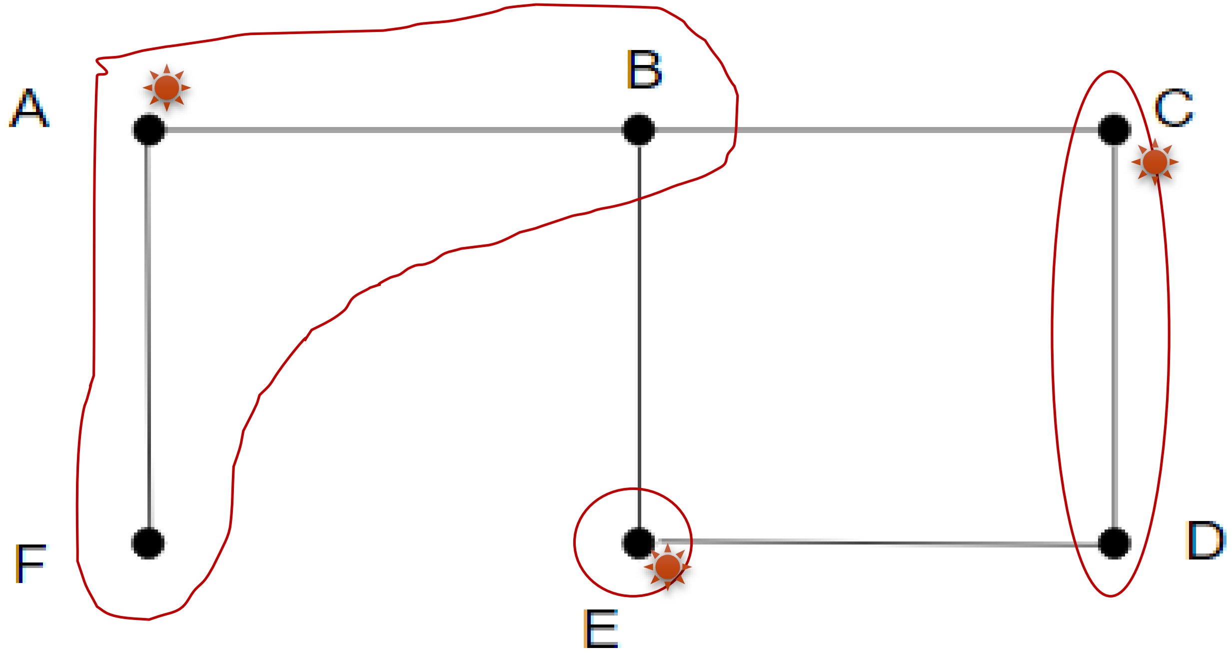


CD

findSet(C) **findSet(D)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

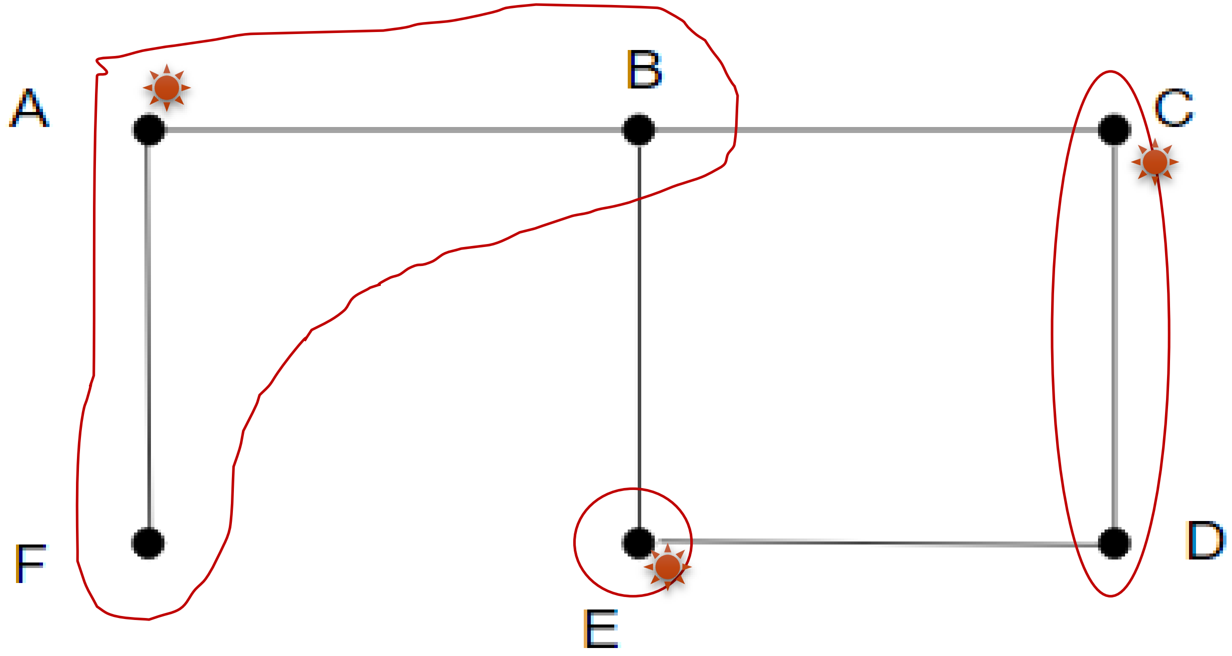


CD

findSet(C) **findSet(D)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

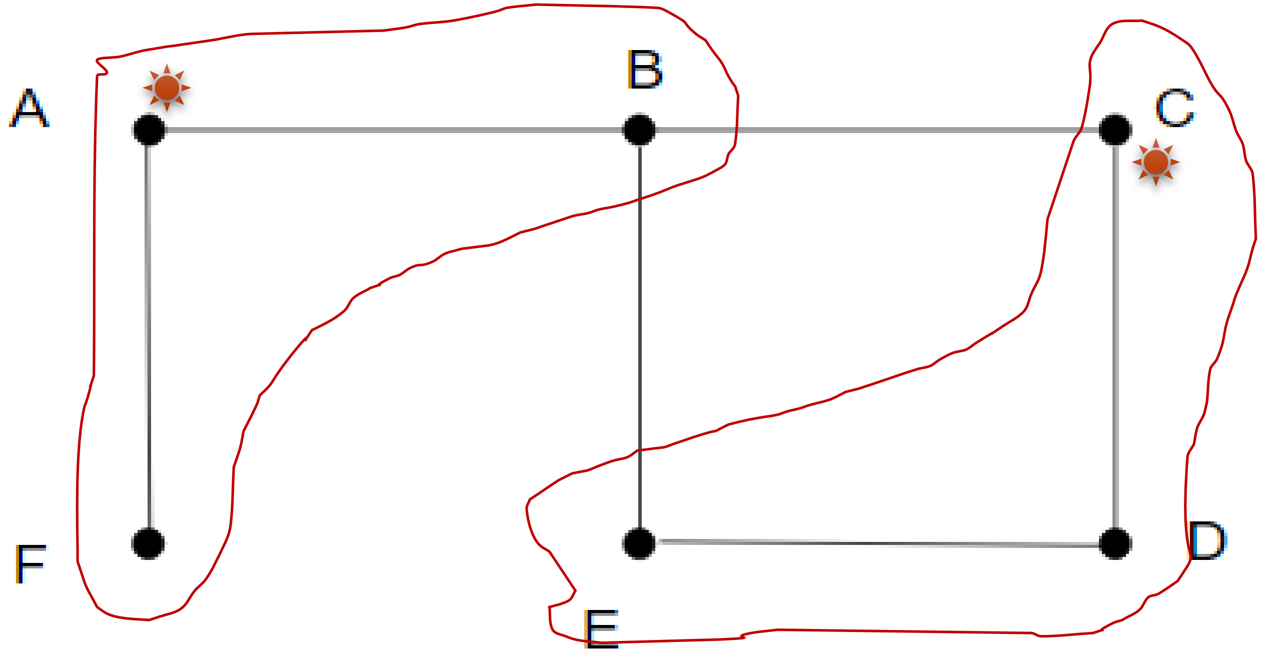


DE

findSet(D) **findSet(E)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

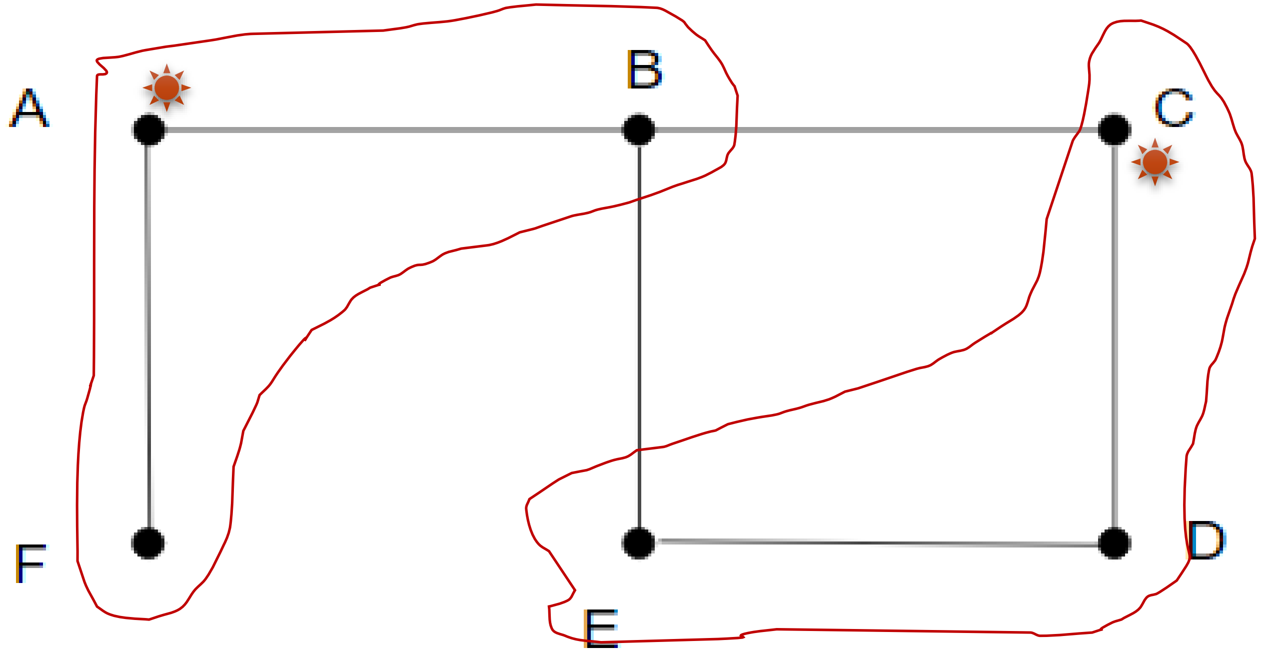


DE

findSet(D) **findSet(E)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

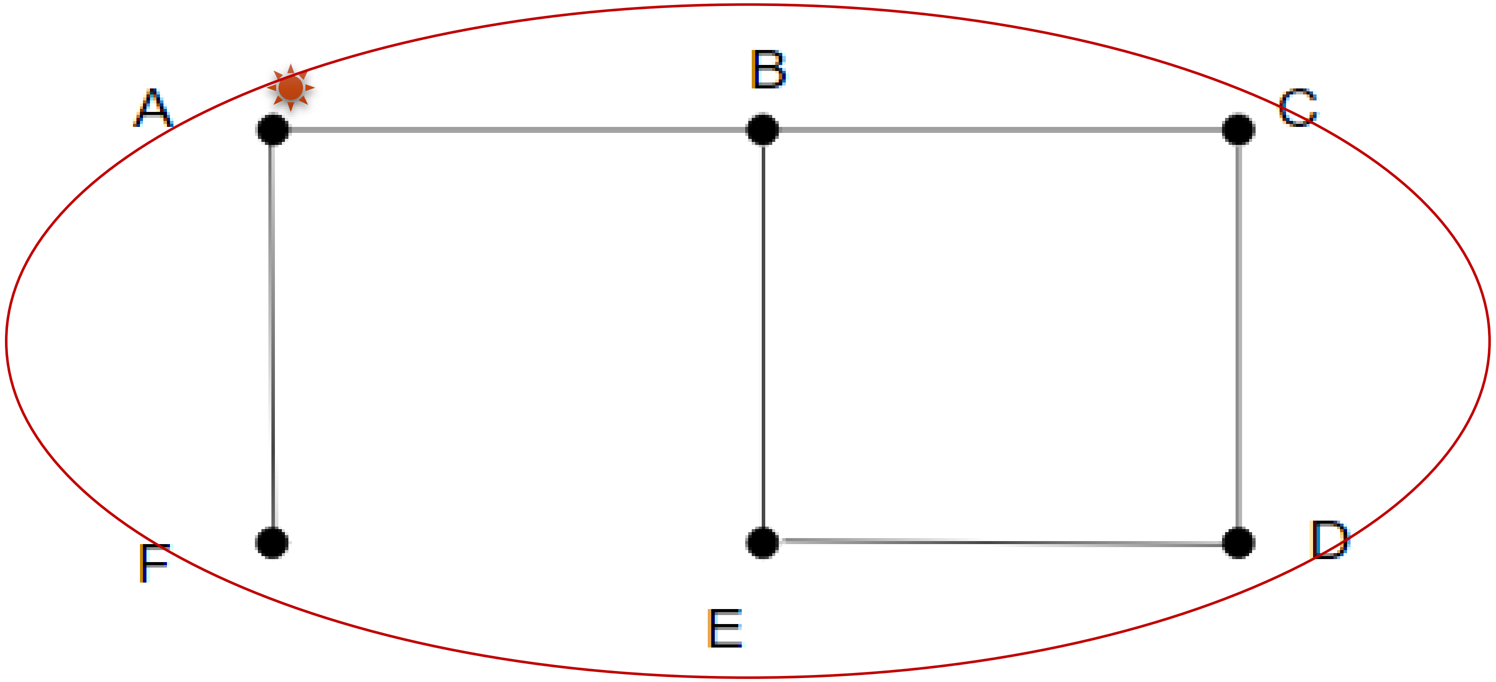


BC

findSet(B) **findSet(C)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

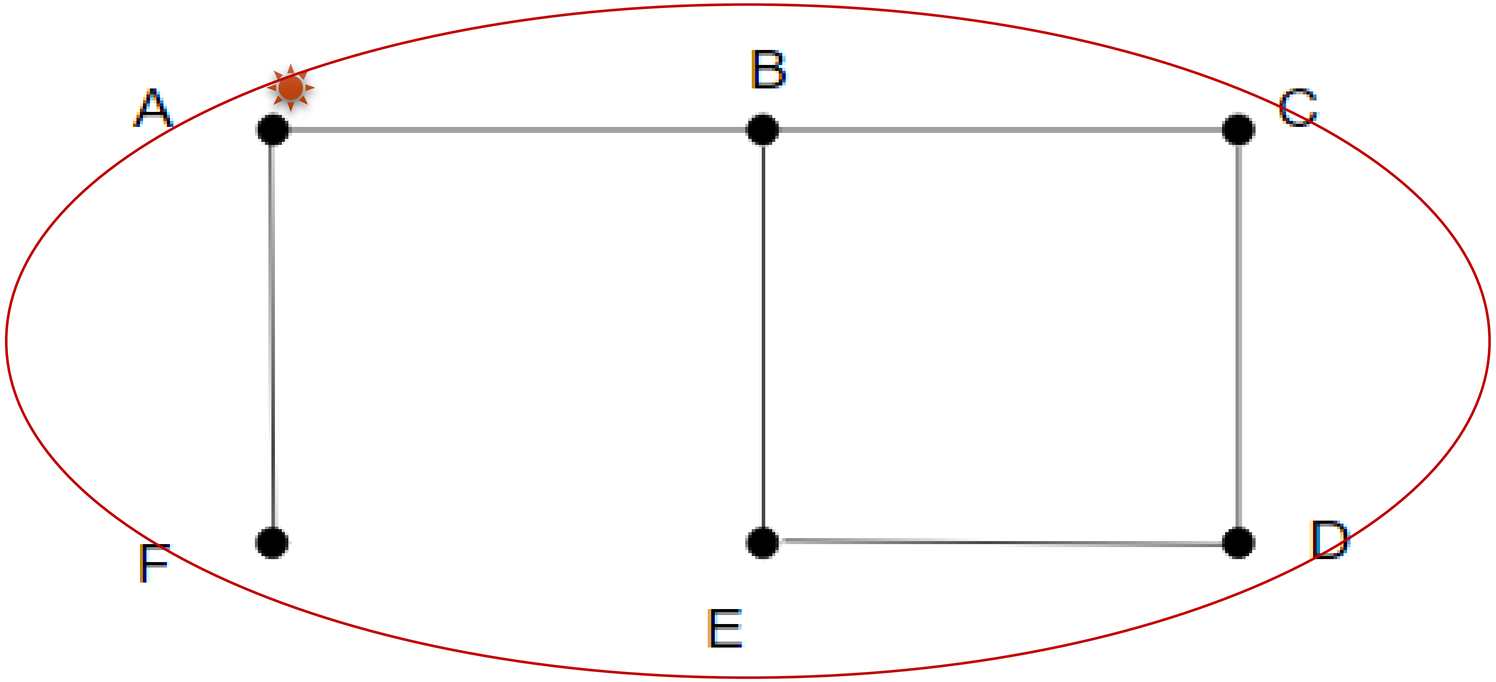


BC

findSet(B) **findSet(C)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE

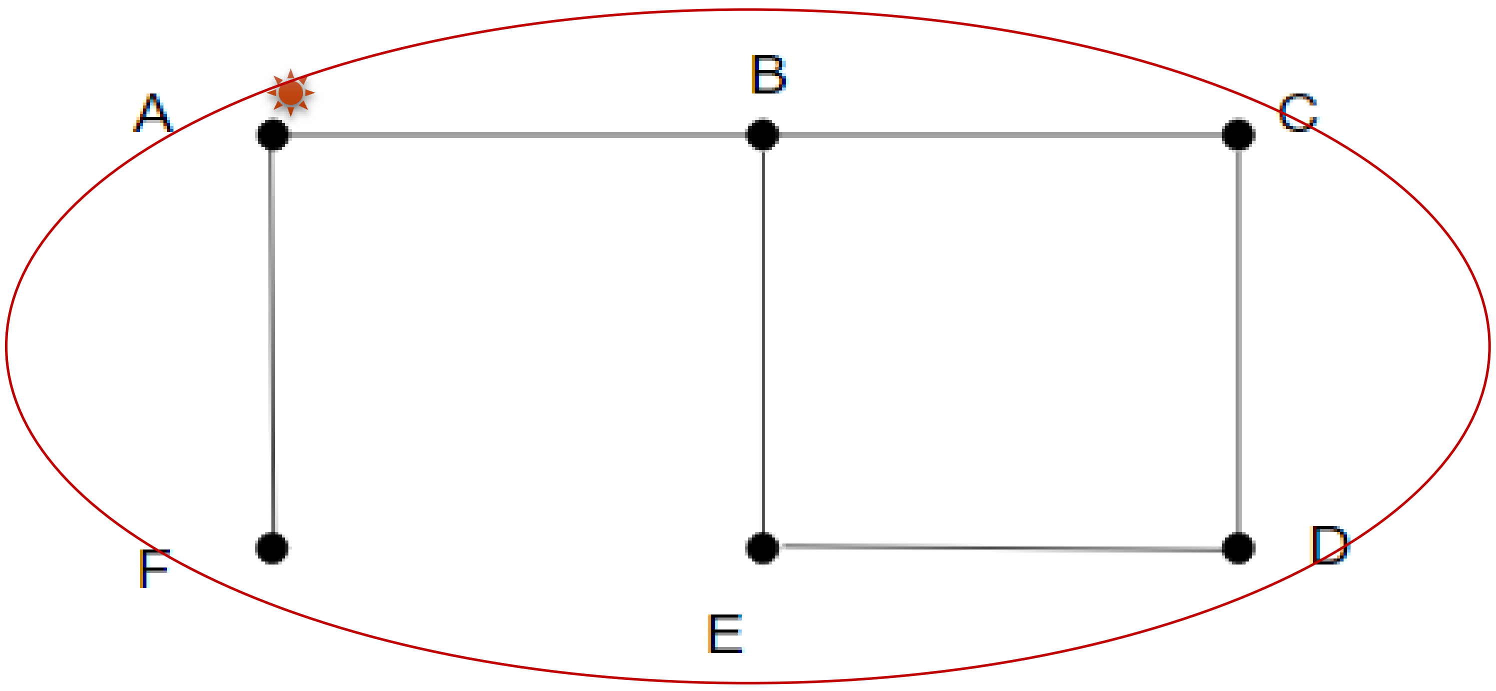


BE

findSet(B) **findSet(E)**

Cycle Detection – Step 2

AF, AB, CD, DE, BC, BE



BE

findSet(B) **findSet(E)**

Result: True

Cycle Detection – Time Complexity

```
cycleDetect(G):
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E:
        if FIND-SET(u) ≠ FIND-SET(v):
            UNION(u, v)
        else:
            return True
    return False
```

Cycle Detection – Time Complexity

```
cycleDetect(G):
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E:
        if FIND-SET(u) ≠ FIND-SET(v):
            UNION(u, v)
        else:
            return True
    return False
```

O(V)

O(E)

TOTAL:
O(V) + O(E)

Cycle Detection – Time Complexity

```
cycleDetect(G):
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E:
        if FIND-SET(u) ≠ FIND-SET(v):
            UNION(u, v)
        else:
            return True
    return False
```

O(V)

O(E)

TOTAL:

O(V)

Structure

- Union Find
  - Naïve Version
    - *Practice*
  - Optimized Version
    - *Practice*
- Cycle Detection
  - *Practice*
- Kruskal's algorithm

Q3. Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

Q3. Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

```
def validTree(self, n, edges):
        """
        :type n: int
        :type edges: List[List[int]]
        :rtype: bool
        """
```

Q3. Graph Valid Tree

Valid Tree:

1. If there are n nodes, there must be n-1 edges.
2. There is no loop.

Q3. Graph Valid Tree

```
def validTree(self, n, edges):
    if len(edges) != n-1: return False

    union = Union()
    for u, v in edges:
        union.makeSet(u)
        union.makeSet(v)

    for u, v in edges:
        if union.findSet(u) == union.findSet(v):
            return False
        else:
            union.union(u, v)
    return True
```

Structure

- Union Find
  - Naïve Version
    - *Practice*
  - Optimized Version
    - *Practice*
- Cycle Detection
  - *Practice*
- Kruskal's algorithm

Kruskal's algorithm

Kruskal's algorithm

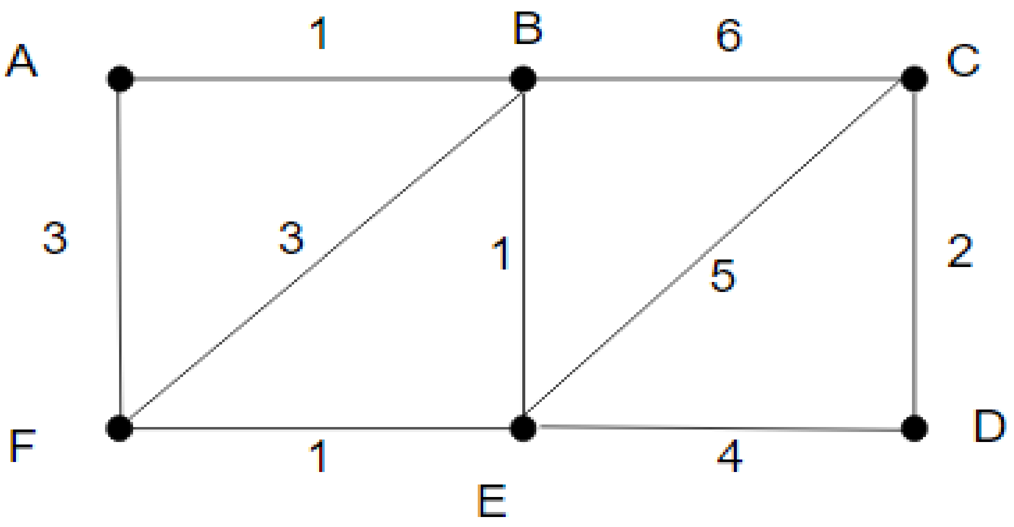
- *Weighted Graph*:

  A weighted graph refers to an edge-weighted graph, where edges have weights or values.

Kruskal's algorithm

- *Weighted Graph*:

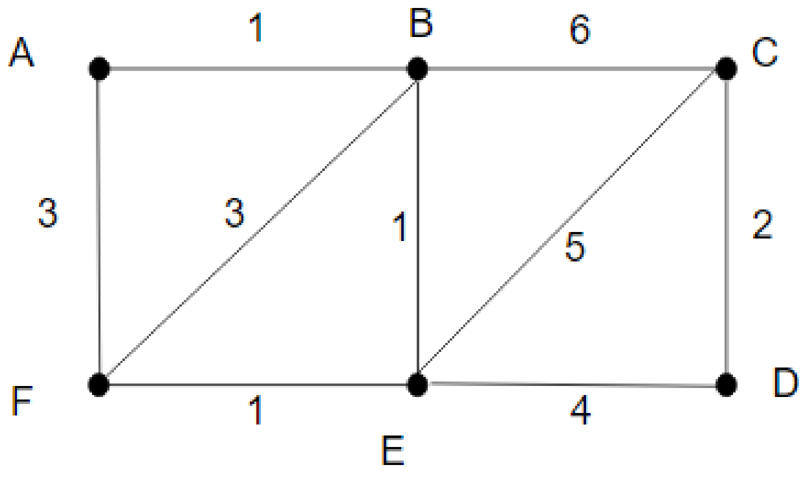
  A weighted graph refers to an edge-weighted graph, where edges have weights or values.



Kruskal's algorithm

- Minimum Spanning Tree:

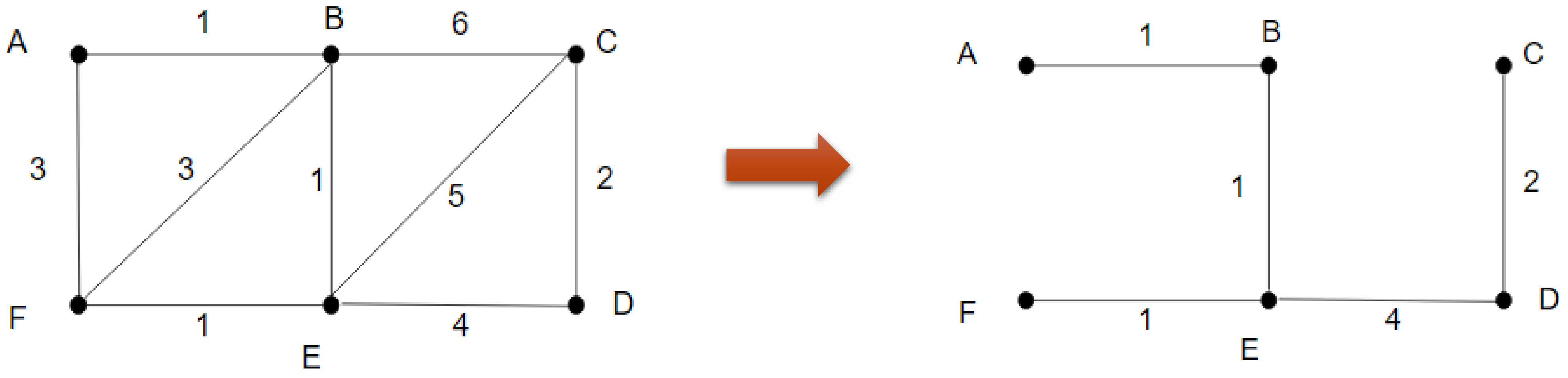
  A minimum spanning tree is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.



Kruskal's algorithm

- Minimum Spanning Tree:

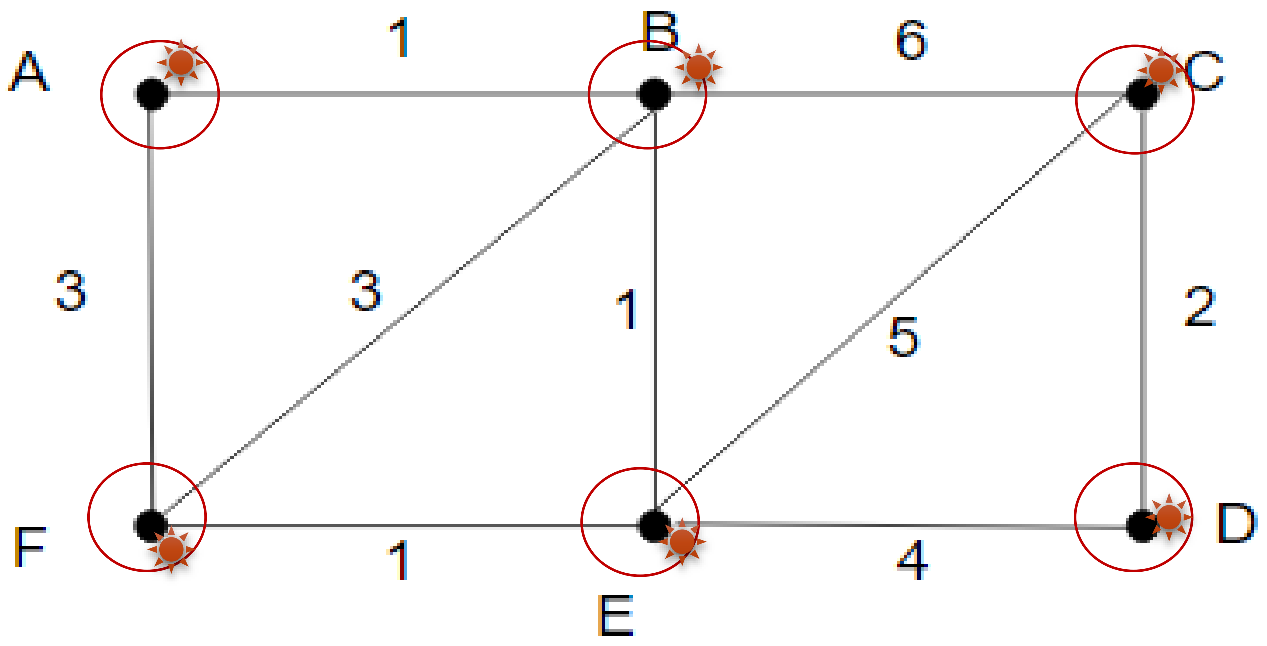
  A minimum spanning tree is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.



Kruskal's algorithm

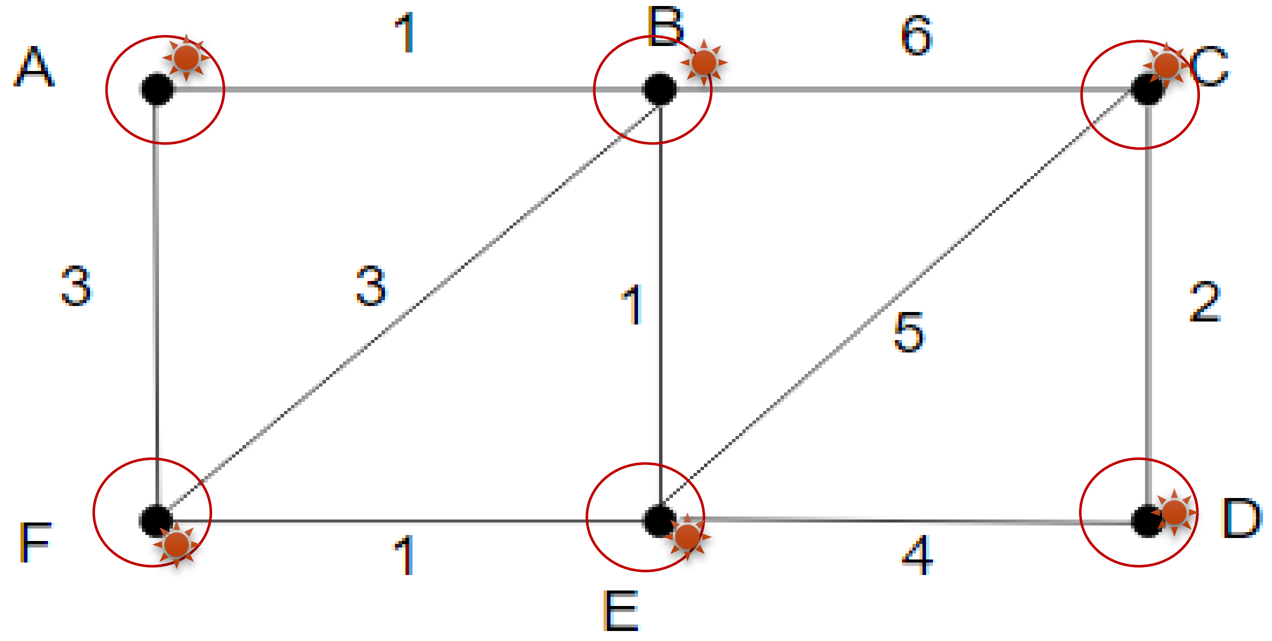
```
KRUSKAL(G):
    Result = ∅
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by increasing order of weight(u, v):
        if FIND-SET(u) ≠ FIND-SET(v):
            Result = Result ∪ {(u, v)}
            UNION(u, v)
    return Result
```

Kruskal's algorithm – Step 1



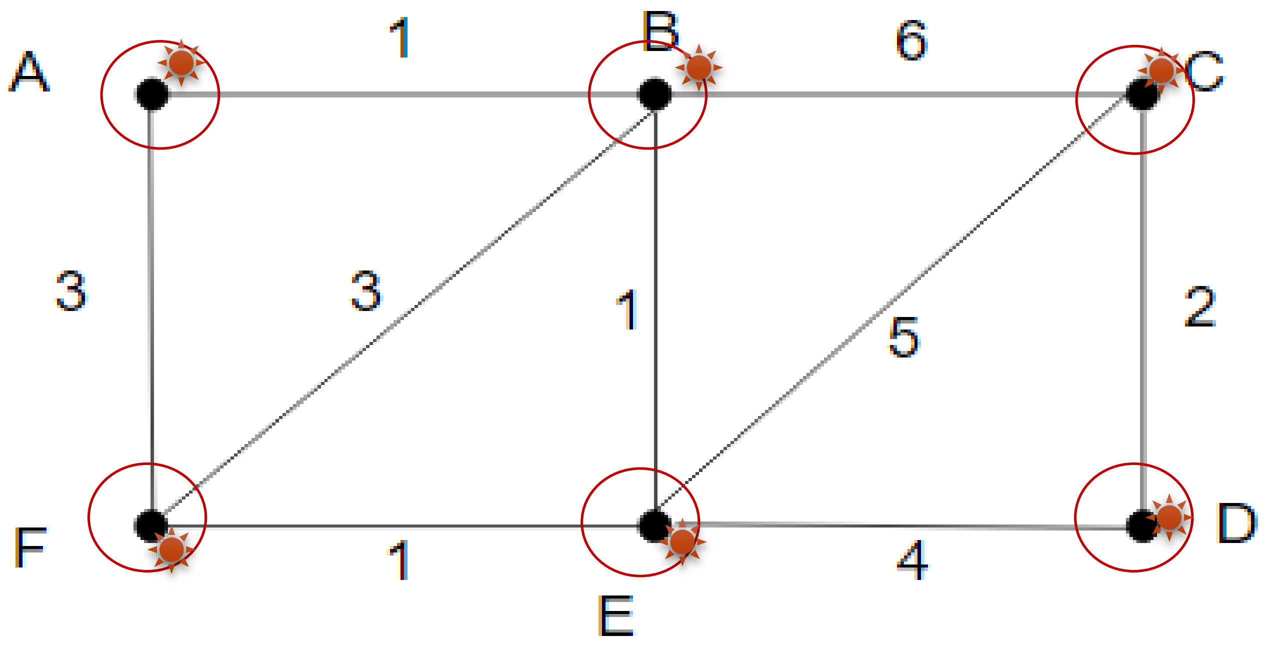
Kruskal's algorithm – Step 2

AB, BE, EF, CD, BF, DE, CE, BC



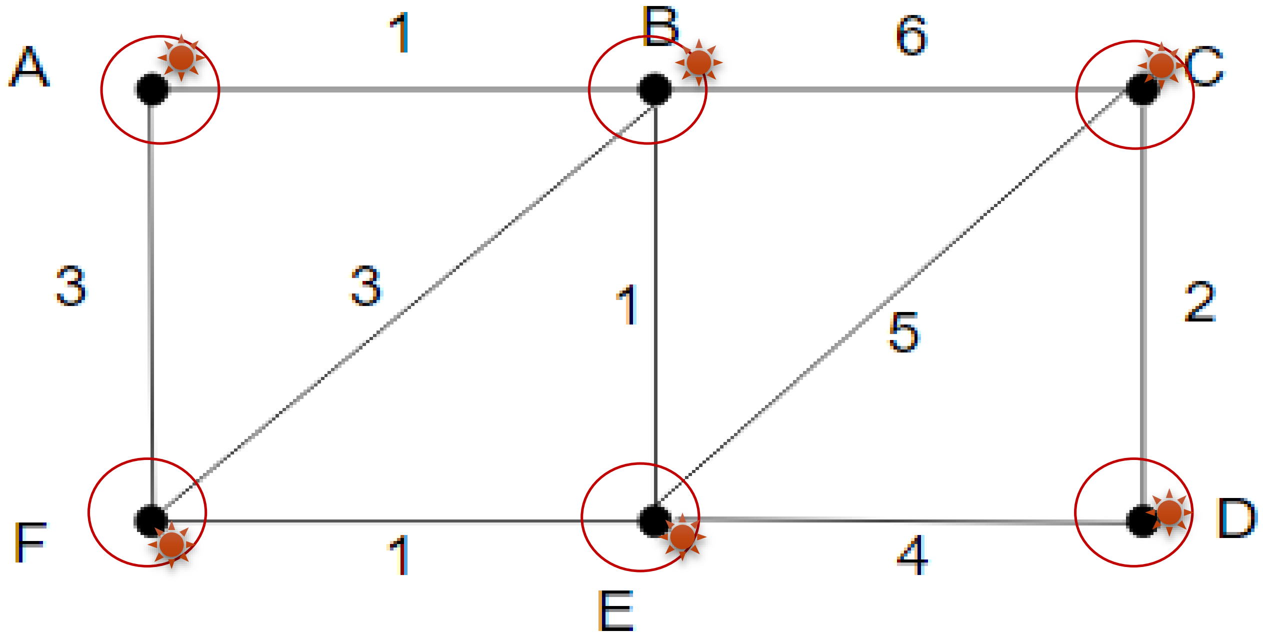
Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC

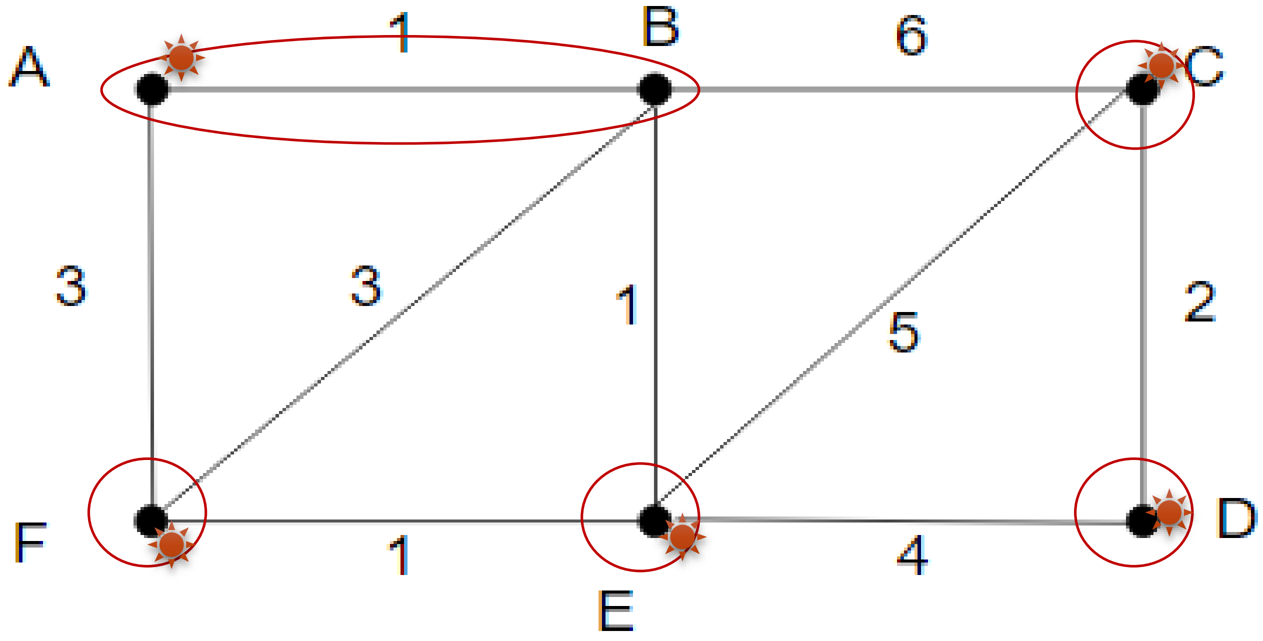


AB

findSet(A) **findSet(B)**

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



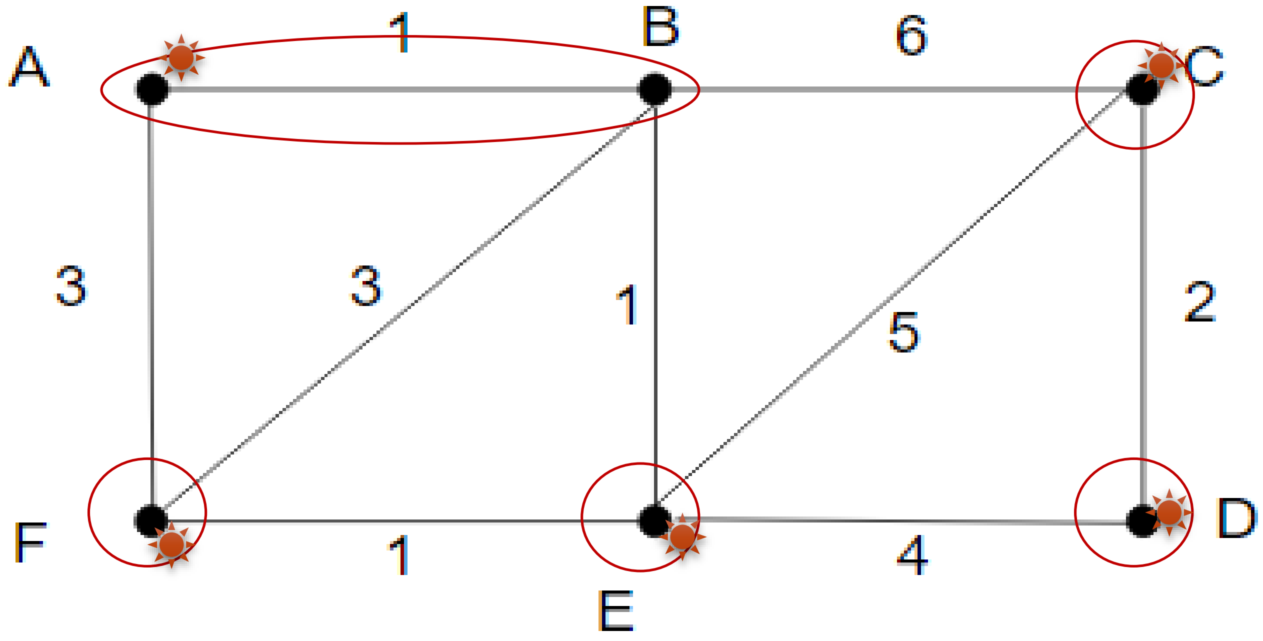
AB

findSet(A) **findSet(B)**

Result: AB

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



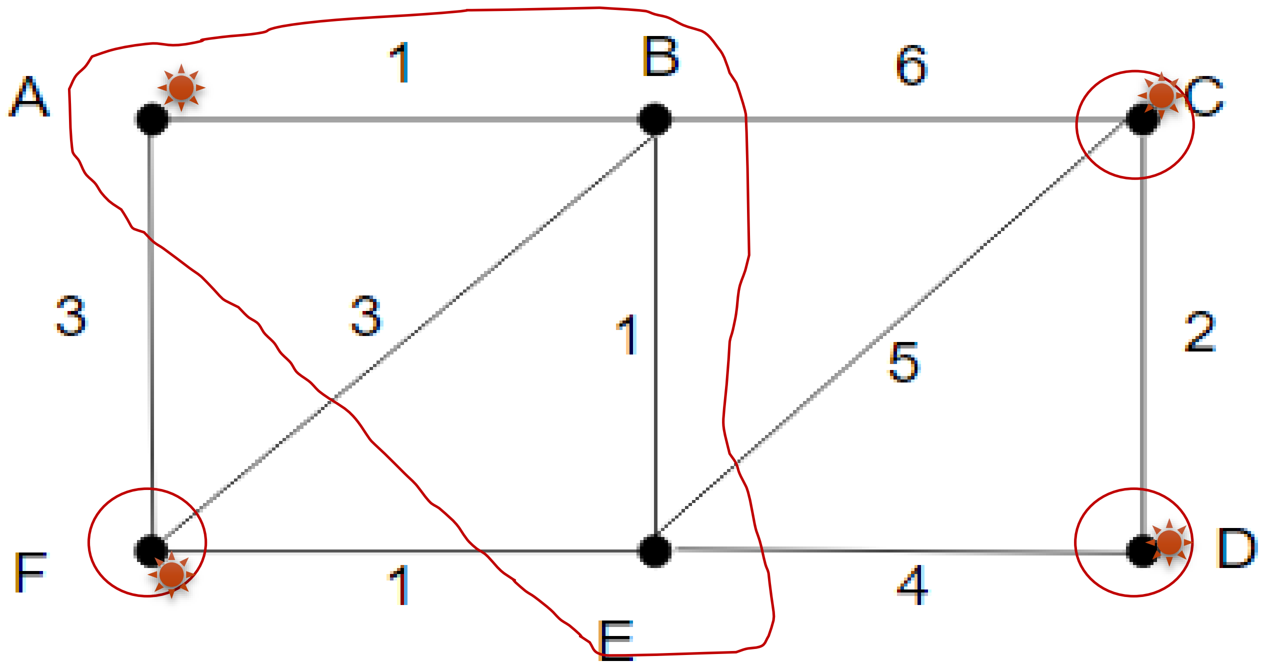
BE

findSet(B) **findSet(E)**

Result: AB

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



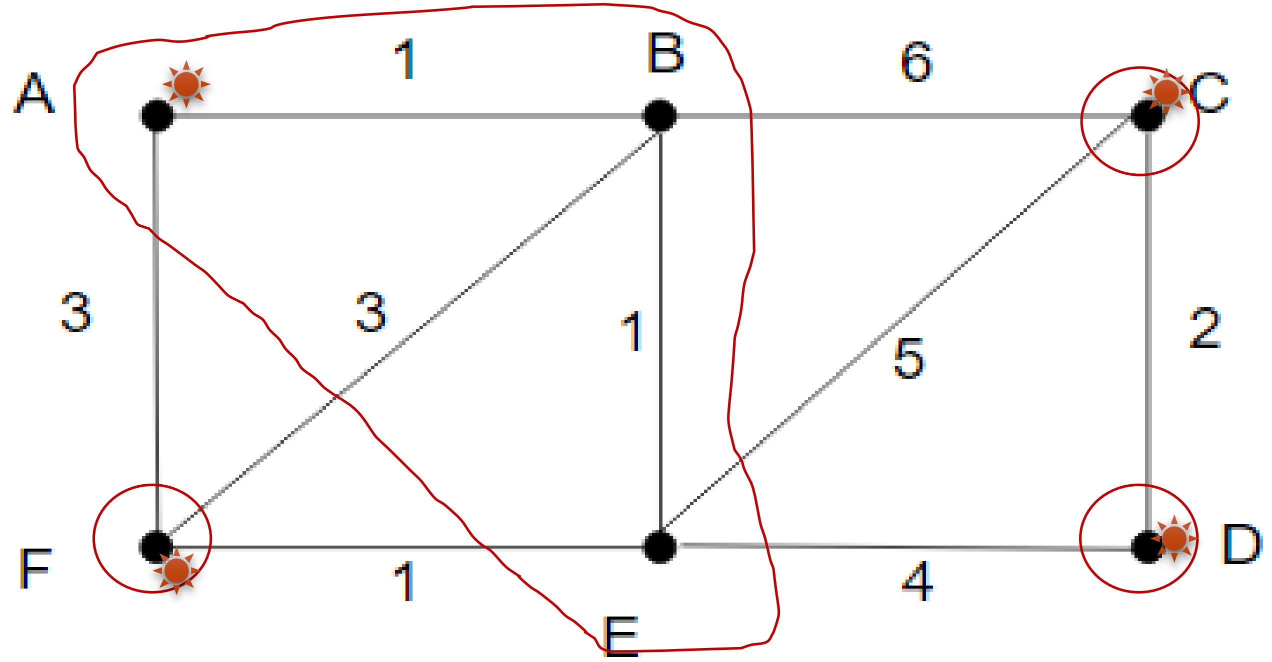
BE

findSet(B) **findSet(E)**

Result: AB, BE

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



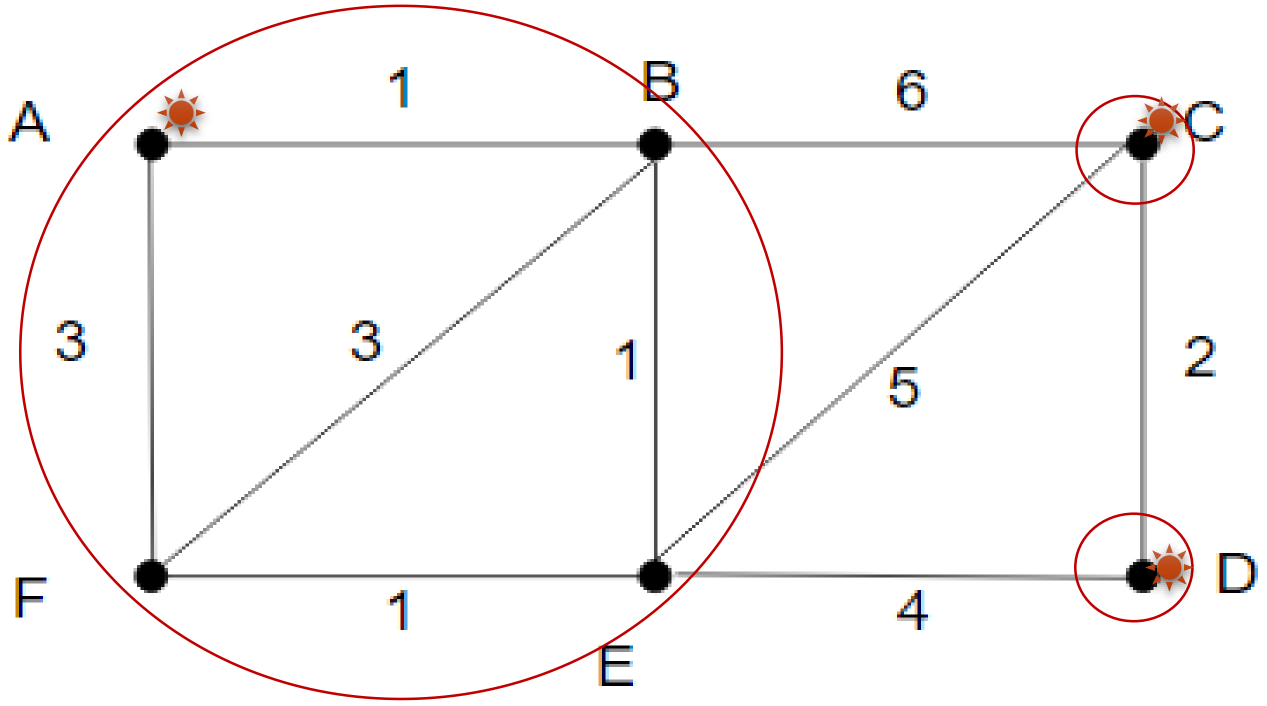
EF

findSet(E) **findSet(F)**

Result: AB, BE

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



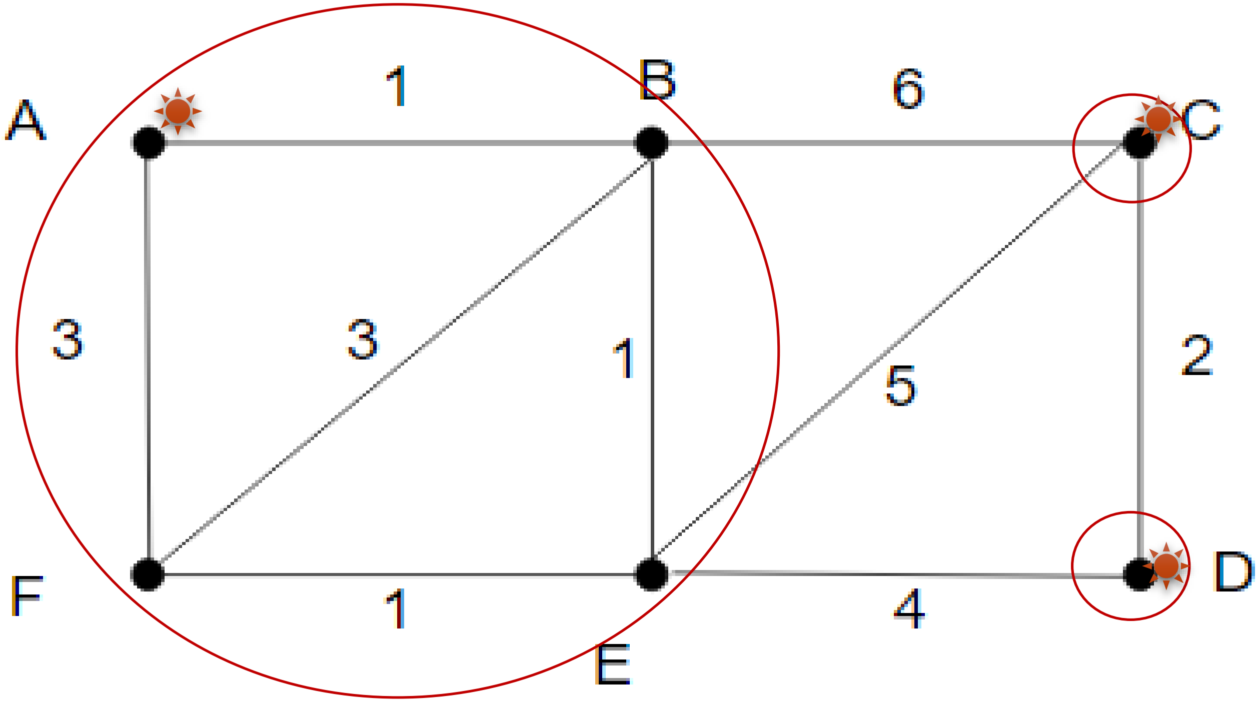
EF

findSet(E) **findSet(F)**

Result: AB, BE, EF

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



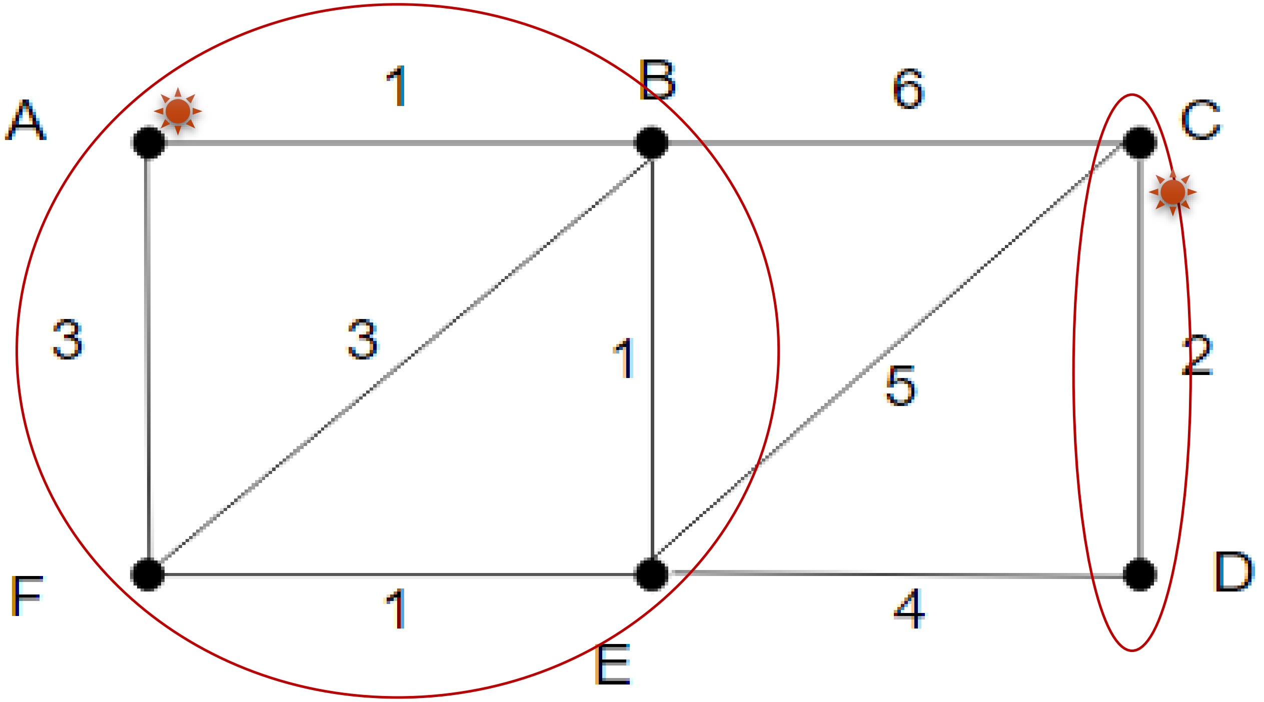
CD

findSet(C) **findSet(D)**

Result: AB, BE, EF

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



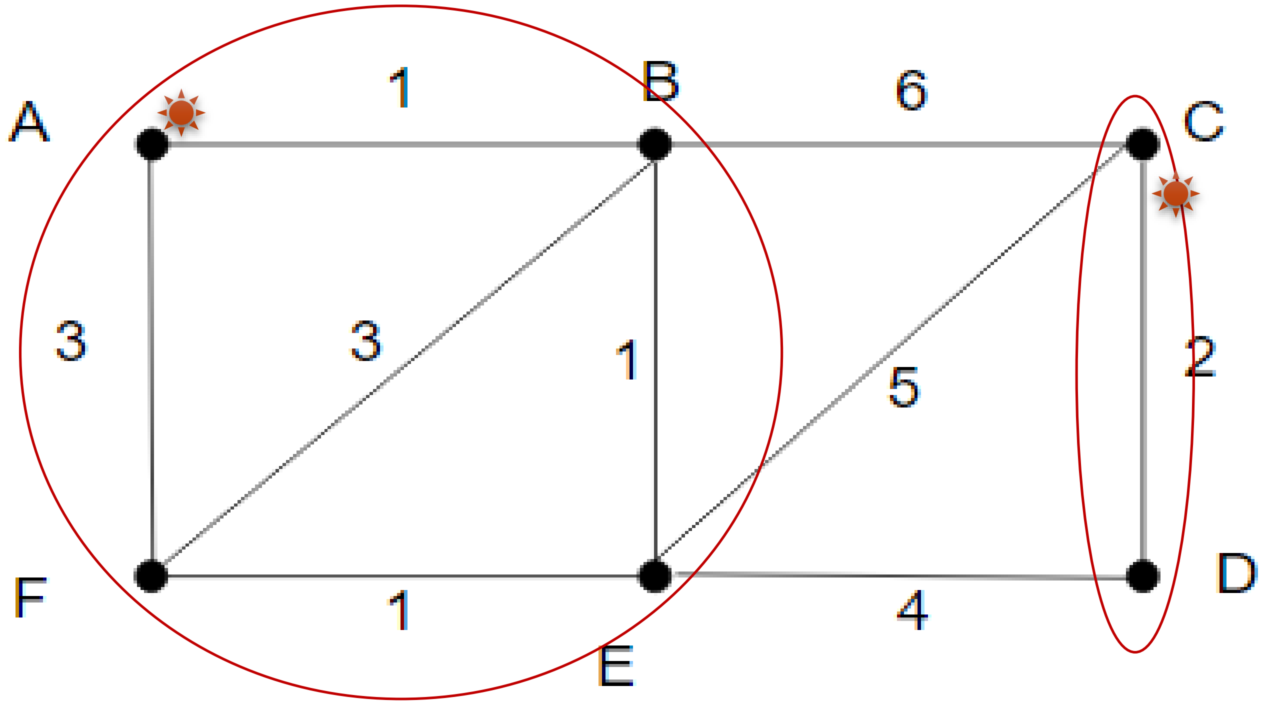
CD

findSet(C) **findSet(D)**

Result: AB, BE, EF, CD

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



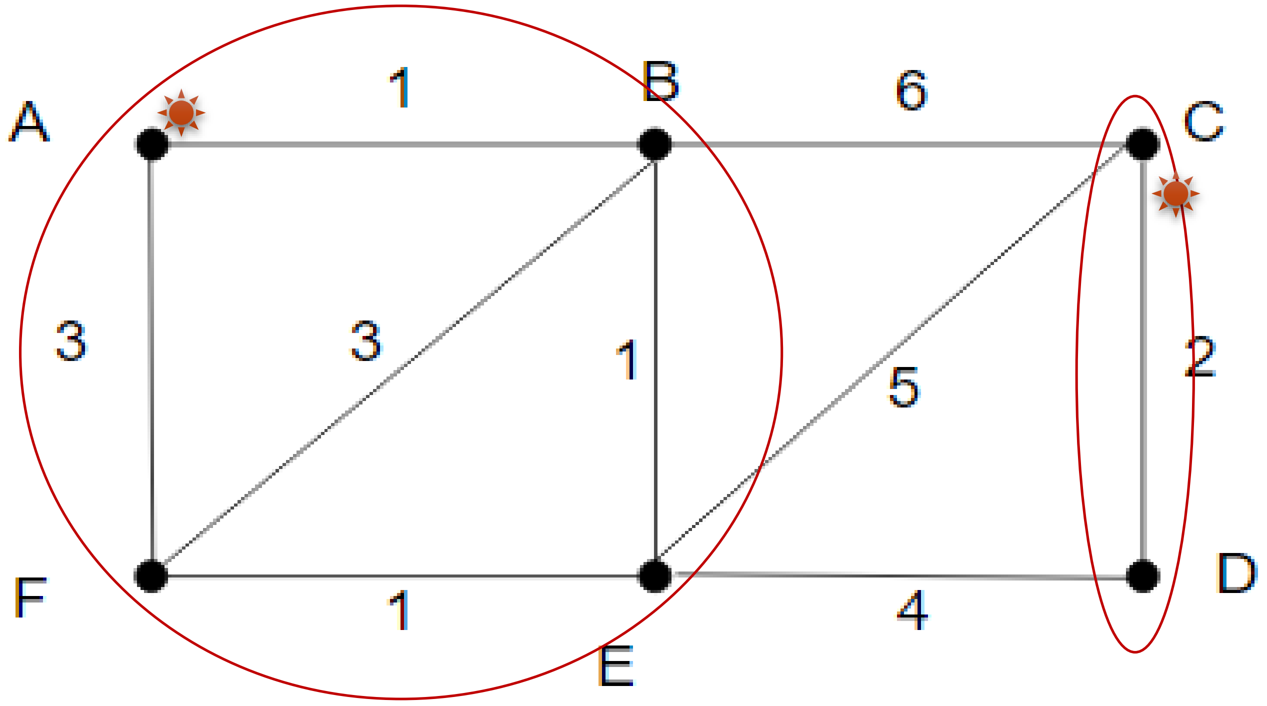
BF

findSet(B) **findSet(F)**

Result: AB, BE, EF, CD

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



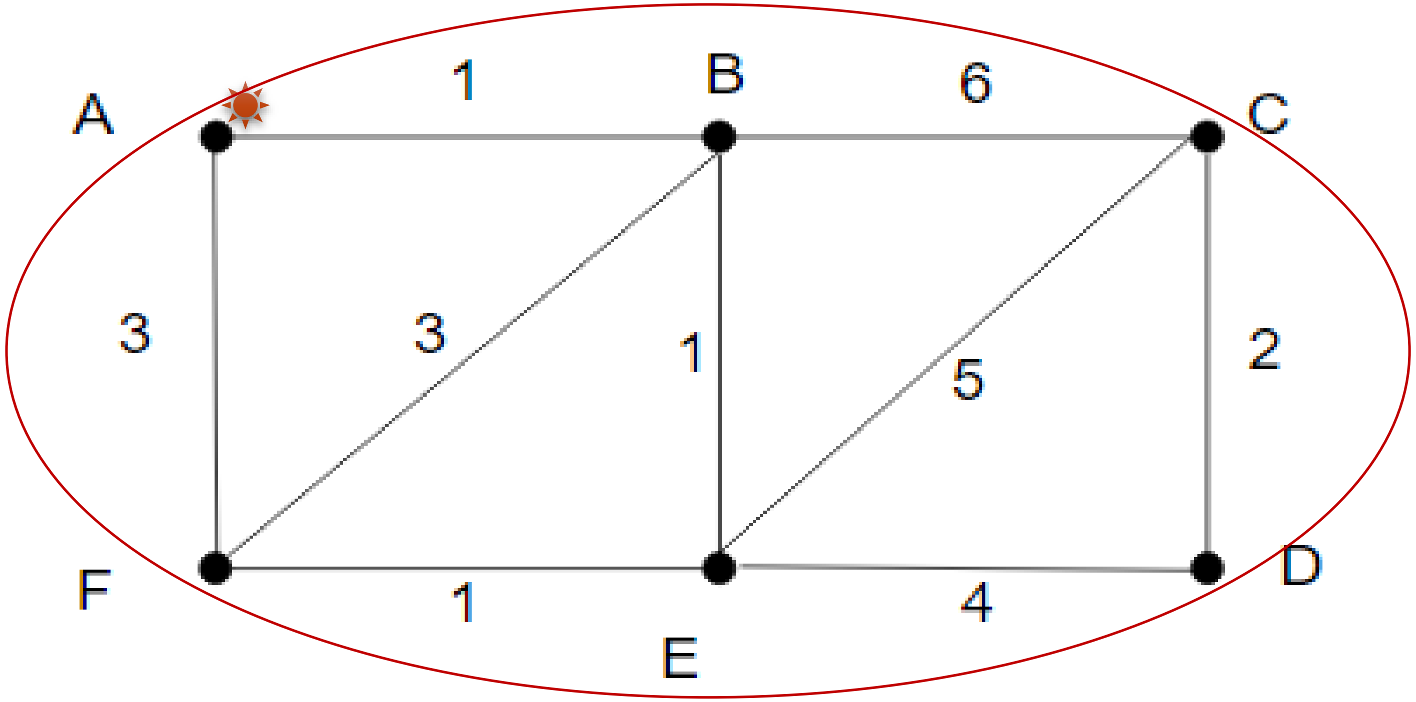
DE

findSet(D) **findSet(E)**

Result: AB, BE, EF, CD

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



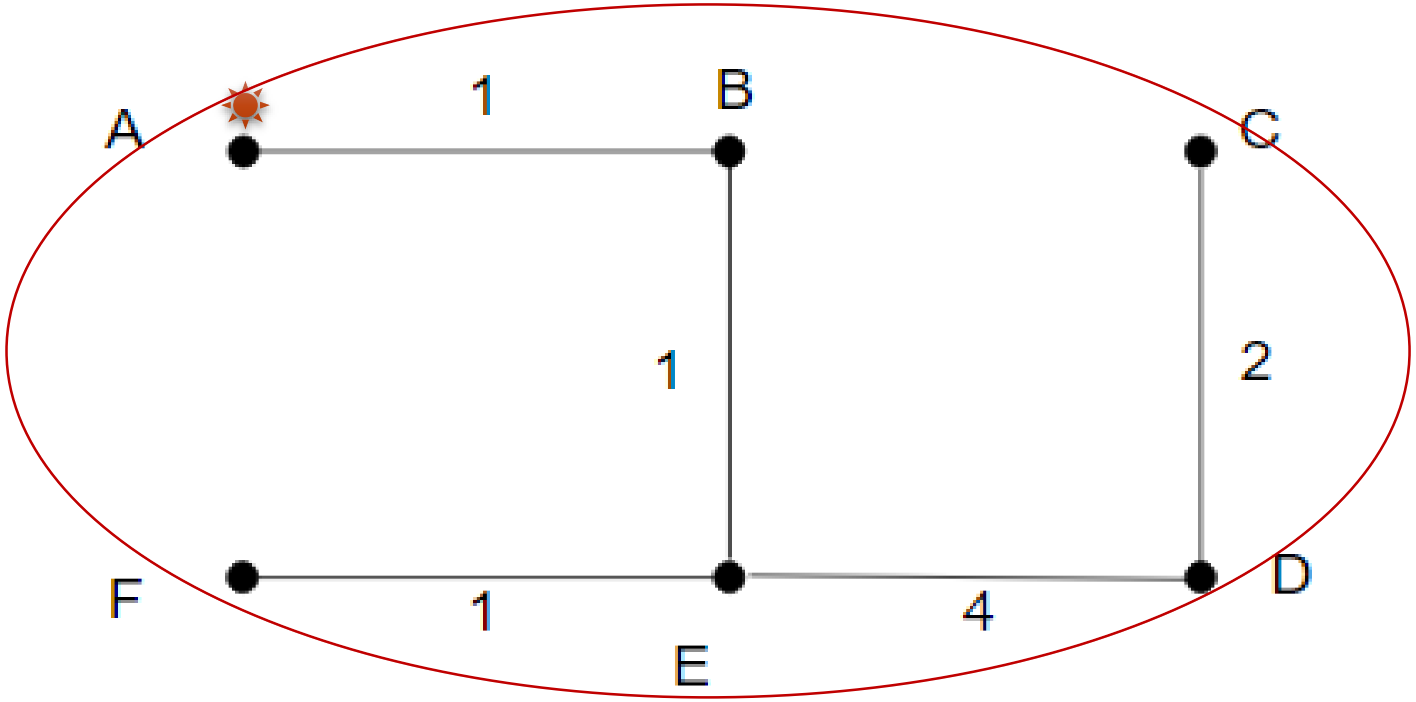
DE

findSet(D) **findSet(E)**

Result: AB, BE, EF, CD, DE

Kruskal's algorithm – Step 3

AB, BE, EF, CD, BF, DE, CE, BC



DE

findSet(D) **findSet(E)**

Result: AB, BE, EF, CD, DE

Kruskal's algorithm – Time Complexity

```
KRUSKAL(G):
    Result = ∅
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by increasing order of weight(u, v):
        if FIND-SET(u) ≠ FIND-SET(v):
            Result = Result ∪ {(u, v)}
            UNION(u, v)
    return Result
```

Kruskal's algorithm – Time Complexity

```
KRUSKAL(G):
    Result = ∅
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by increasing order of weight(u, v):
        if FIND-SET(u) ≠ FIND-SET(v):
            Result = Result ∪ {(u, v)}
            UNION(u, v)
    return Result
```

O(V)

O(E log E)

O(E)

TOTAL:

O(V) + O(E log E) + O(E)

Kruskal's algorithm – Time Complexity

```
KRUSKAL(G):
    Result = ∅
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by increasing order of weight(u, v):
        if FIND-SET(u) ≠ FIND-SET(v):
            Result = Result ∪ {(u, v)}
            UNION(u, v)
    return Result
```

O(V)

O(E log E)

O(E)

TOTAL:

O(E log E)

neohao@uga.edu

Thanks!