CSCI 2270-305 Recitation 04/02: Graphs (Review)

Varad Deshmukh

Logistics

- Assignment 8 is due this Sunday.
  - Click on **Attempt** and **Submit All and Finish** when you are done!

- Midterm 2 on 04/10 at 5 pm:
  - MCQs to be completed within 1 hour.
  - Coding questions due by midnight.
  - I will be covering hash tables, and do a midterm review in the next recitation.

- Office Hours Zoom link: https://cuboulder.zoom.us/j/975589526
  - Thursday: 10:45 to 11:45 am, 2-3 pm
  - Friday: 9-11 am

Recitation Format

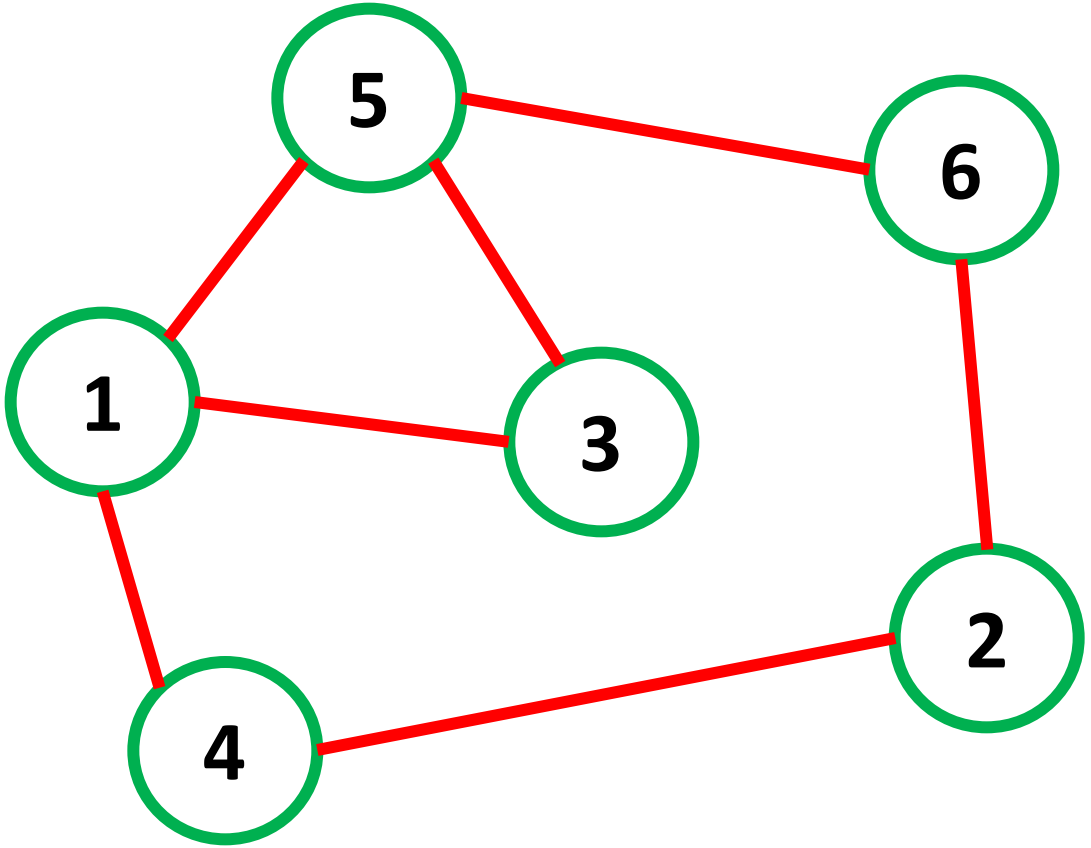
- Participation: If you have questions, please raise your hand (zoom feature), or PM Elizabeth/Aazer and they will prompt me.
  - If there is no response, just unmute yourself and speak up!

- Recitation attendance is not compulsory.
- Submit your recitation exercise on Moodle by Sunday (Recitation 11).
- We will hang around to help you with the exercise (breakout rooms).

- Recommended that you keep your video on, since we are already socially distanced.

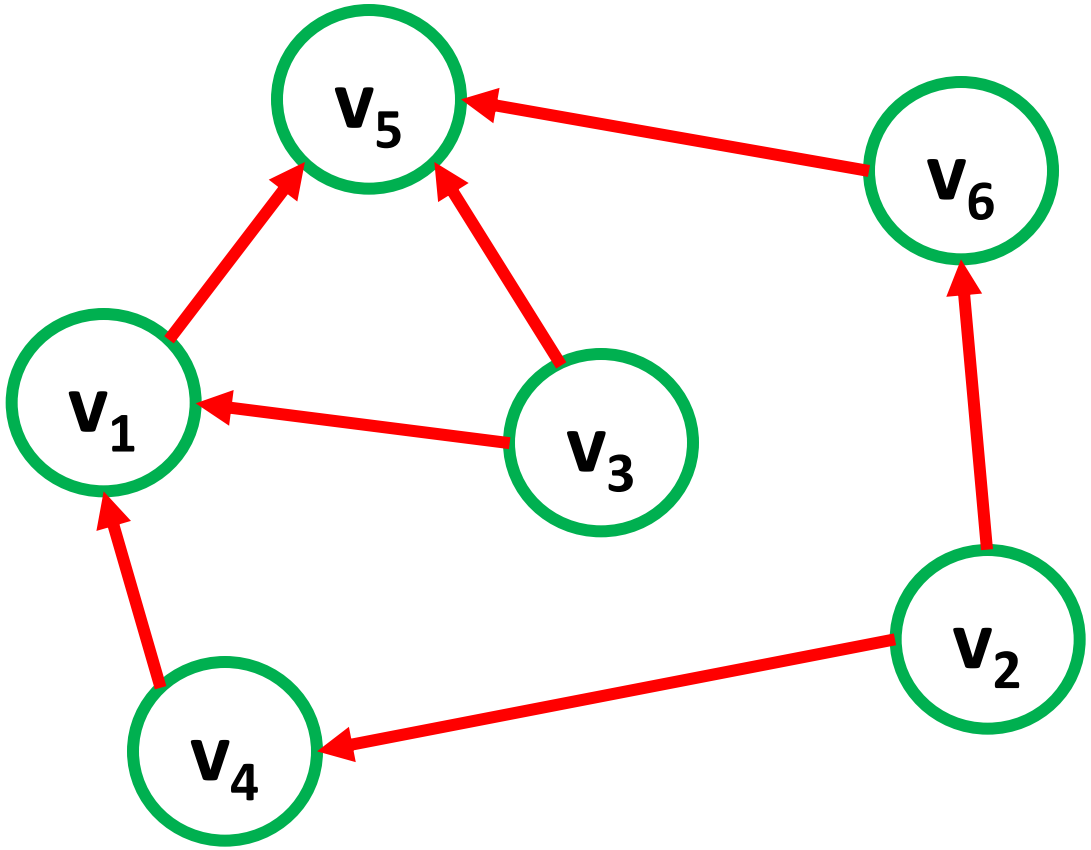
What is a Graph

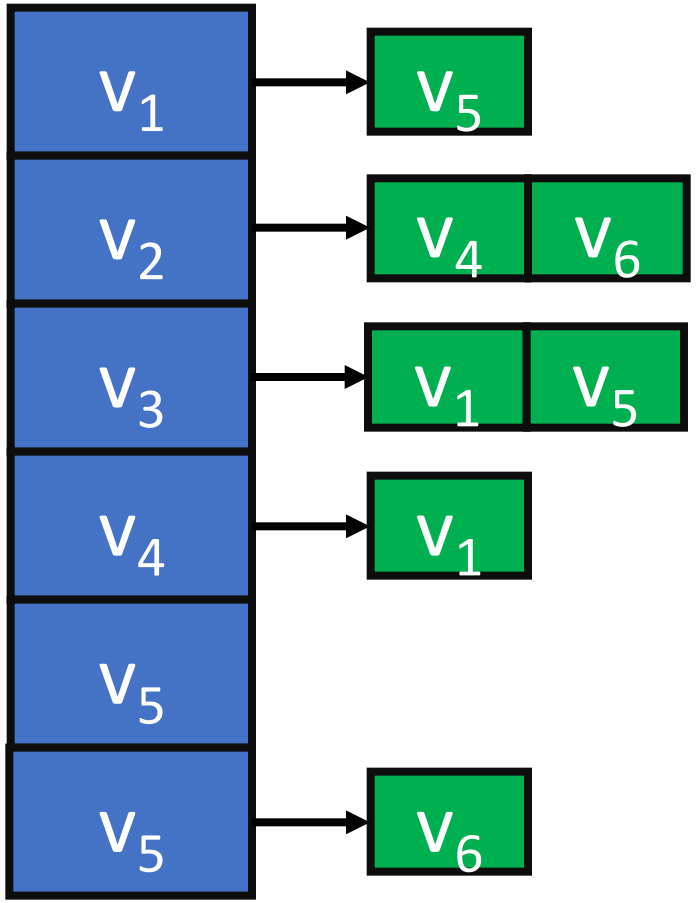
- Graph is a collection of a vertices (nodes) connected to each other via edges.



This graph has 6 nodes and 7 edges.

Adjacency List





Implementing a Graph

```
struct adjVertex{
    vertex *v;
};

struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

- How do you store list of vertices?

  `std::vector<vertex*> vertices;`

- How do you access the key of the second vertex?

- How do you access the visited flag of the third vertex?

Implementing a Graph

```
struct adjVertex{
    vertex *v;
};

struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

- How do you store list of vertices?

  `std::vector<vertex*> vertices;`

- How do you access the key of the second vertex?

  `vertices[2]->key`

- How do you access the visited flag of the third vertex?

  `vertices[3]->visited`

Implementing a Graph

```
struct adjVertex{
    vertex *v;
};

struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

- How do you access the adjacency list of a vertex 3?
- adj its itself a list.
- How do you access the second vertex in the adj list of vertex 3?
- How do you check if that vertex was visited?

Implementing a Graph

```
struct adjVertex{
    vertex *v;
};

struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

- How do you access the adjacency list of a vertex 3?

  `vertices[3]->adj`

- adj its itself a list.

- How do you access the second vertex in the adj list of vertex 3?

- How do you check if that vertex was visited?

Implementing a Graph

```
struct adjVertex{
    vertex *v;
};

struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

- How do you access the adjacency list of a vertex 3?

  `vertices[3]->adj`

- adj its itself a list.

- How do you access the second vertex in the adj list of vertex 3?

  `vertices[3]->adj[2].v;`

- How do you check if that vertex was visited?

Implementing a Graph

```
struct adjVertex{
    vertex *v;
};

struct vertex{
    int key;
    bool visited = false;
    int distance = 0;
    vertex *pred = NULL; // predecessor
    std::vector<adjVertex> adj;
};
```

- How do you access the adjacency list of a vertex 3?

```
vertices[3]->adj
```

- adj its itself a list.

- How do you access the second vertex in the adj list of vertex 3?

```
vertices[3]->adj[2].v;
```

- How do you check if that vertex was visited?

```
vertex *n = vertices[3];
n->adj[2].v->visited;
```

Graph Traversals

- Depth First Search
  - At each vertex, recursively visit or process each neighbor until all its neighbors are processed.

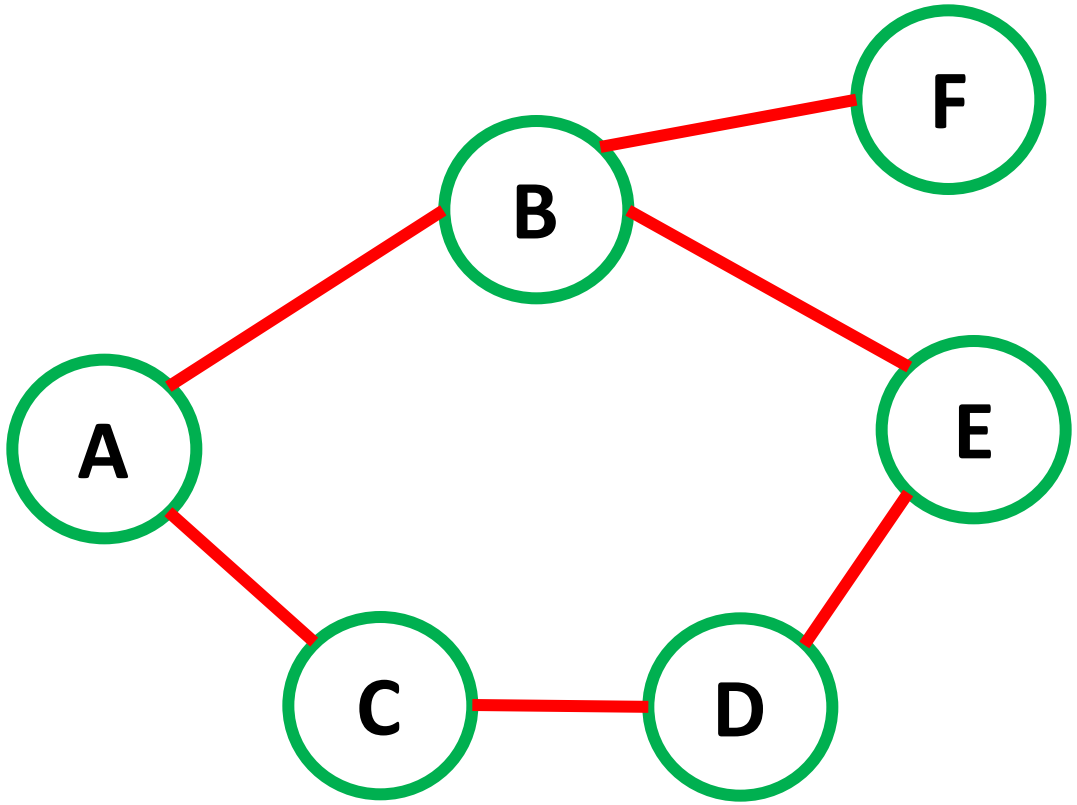
- Breadth First Search
  - At each vertex, visit or process all of its neighbors first, before moving on to their unvisited neighbors.

Some Graph Traversal Applications

- Finding the shortest path between two vertices.
- Finding the number of connected components in the graph.
- Finding if a cycle exists in the graph.

Finding the shortest path (walkthrough)

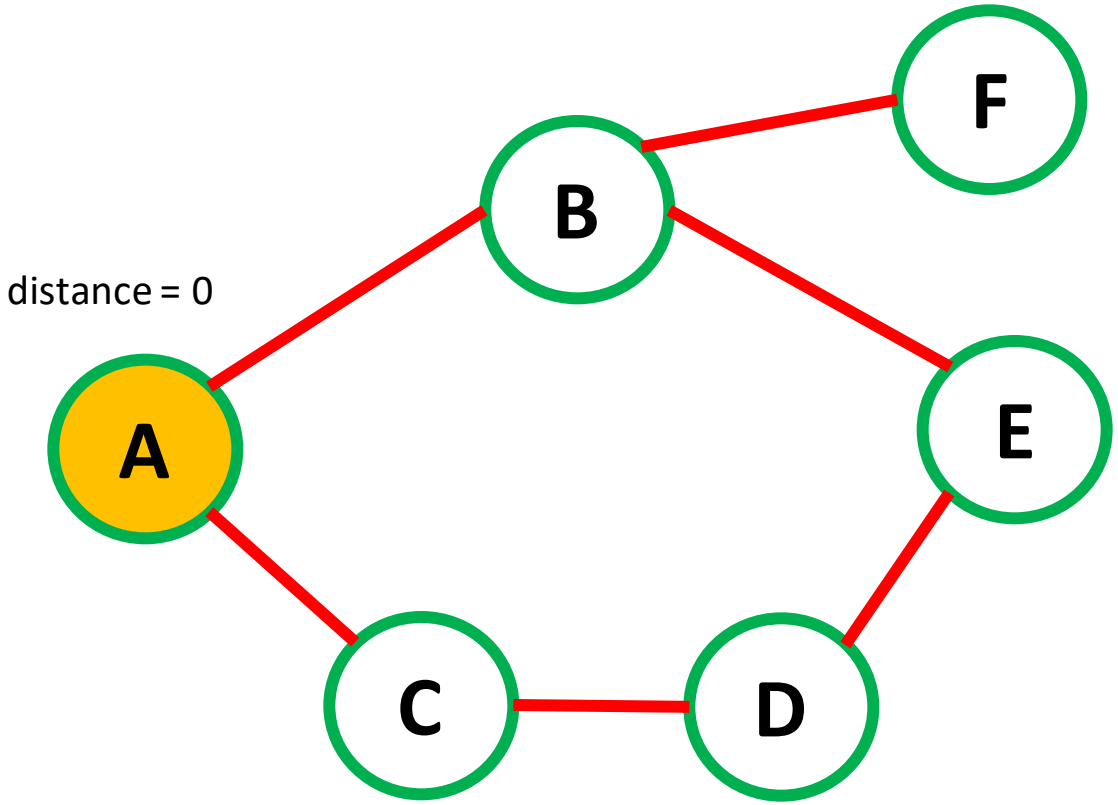
- Shortest path between **A** and **E**



Queue

Finding the shortest path (walkthrough)

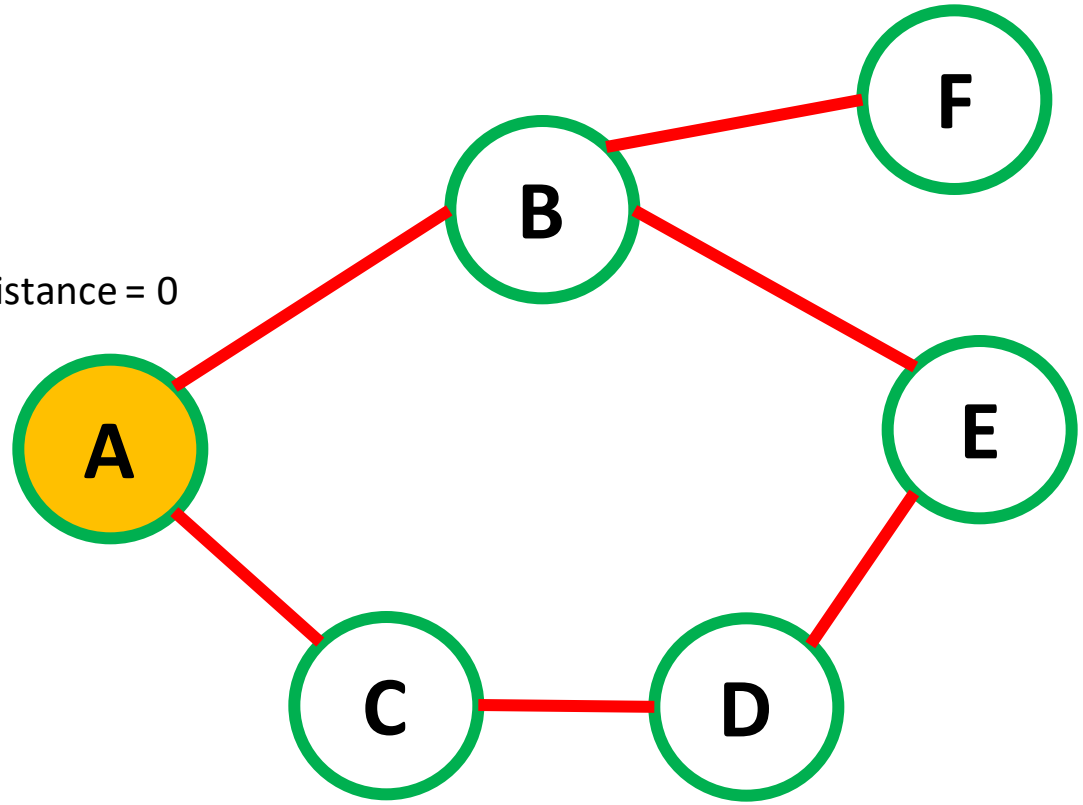
- Shortest path between **A** and **E**



Queue

Finding the shortest path (walkthrough)

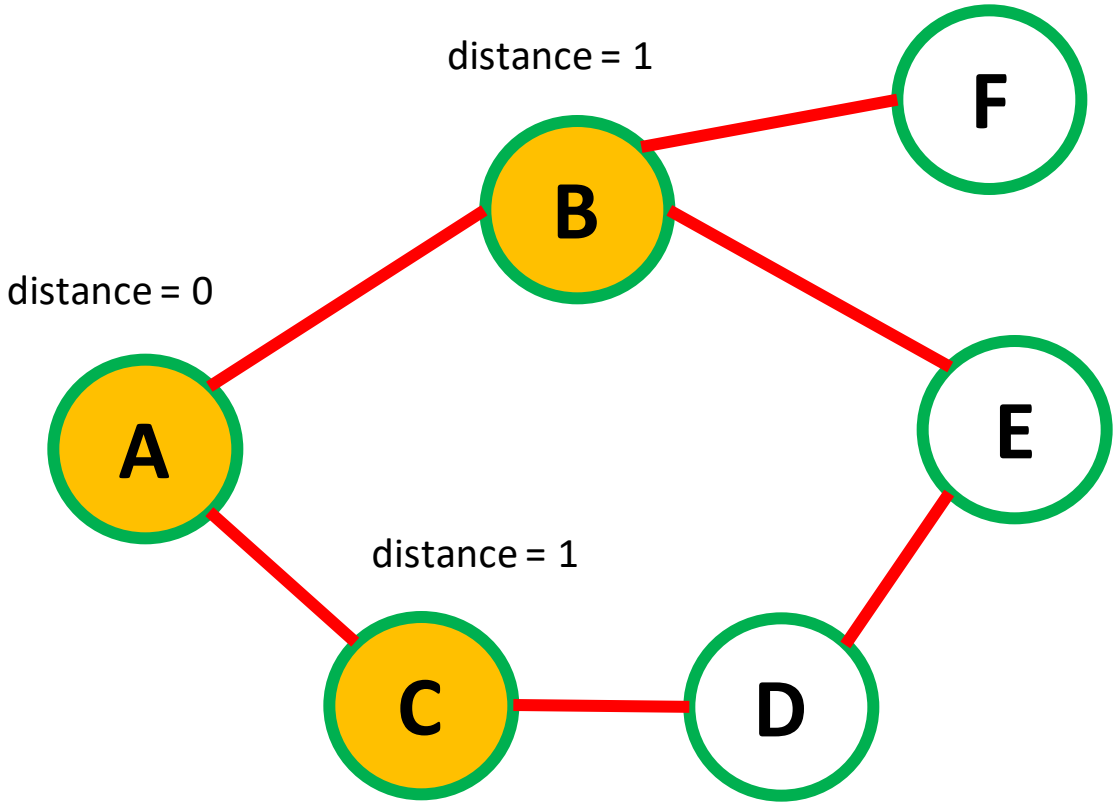
- Shortest path between **A** and **E**



Queue

Finding the shortest path (walkthrough)

- Shortest path between **A** and **E**

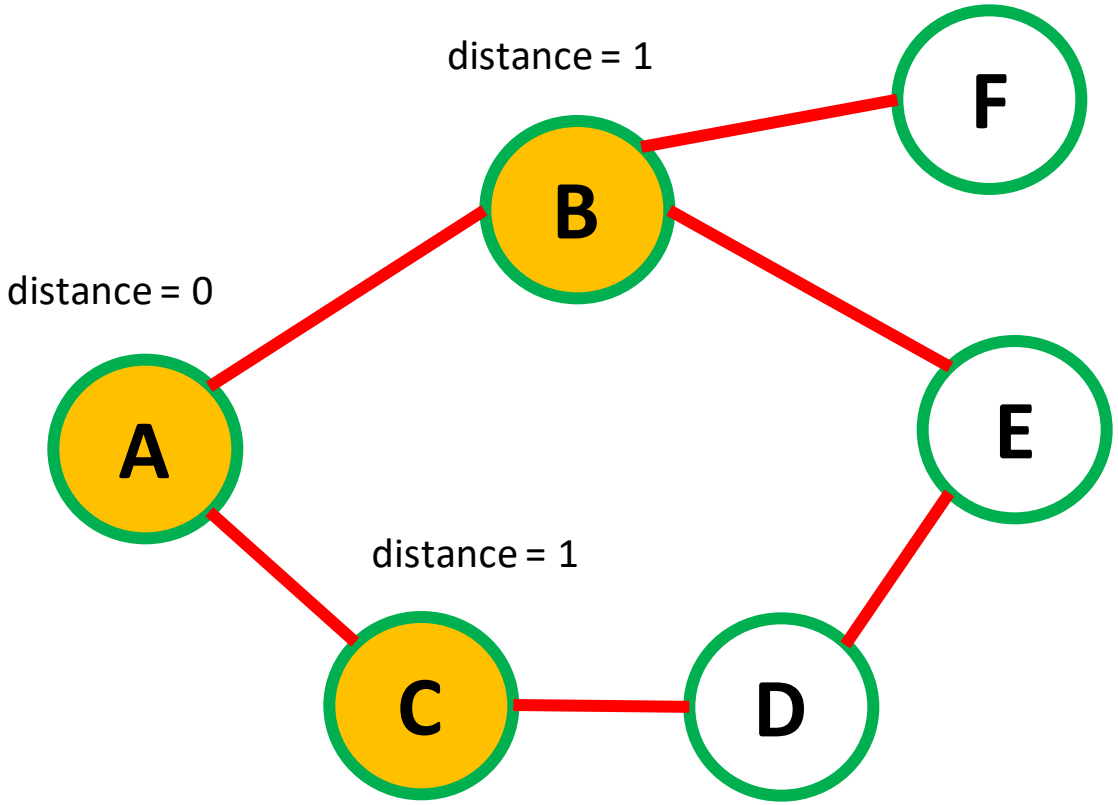


Queue

The order of processing is flexible

Finding the shortest path (walkthrough)

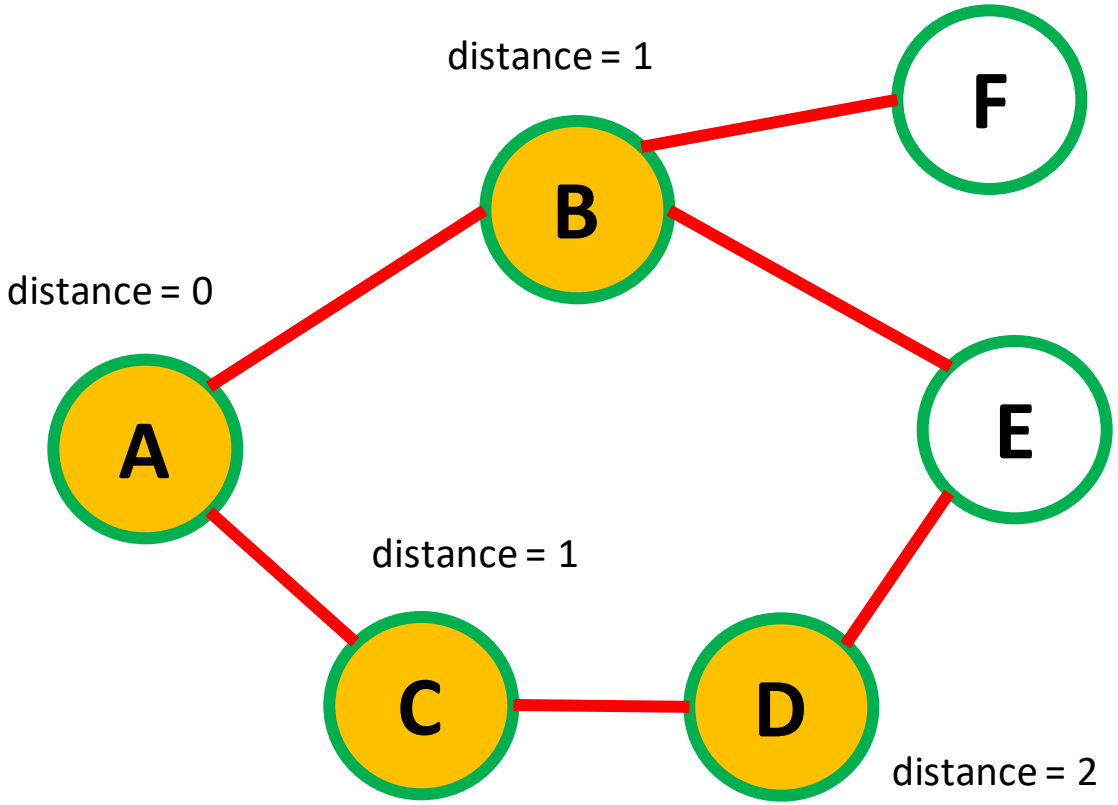
- Shortest path between **A** and **E**



Queue

Finding the shortest path (walkthrough)

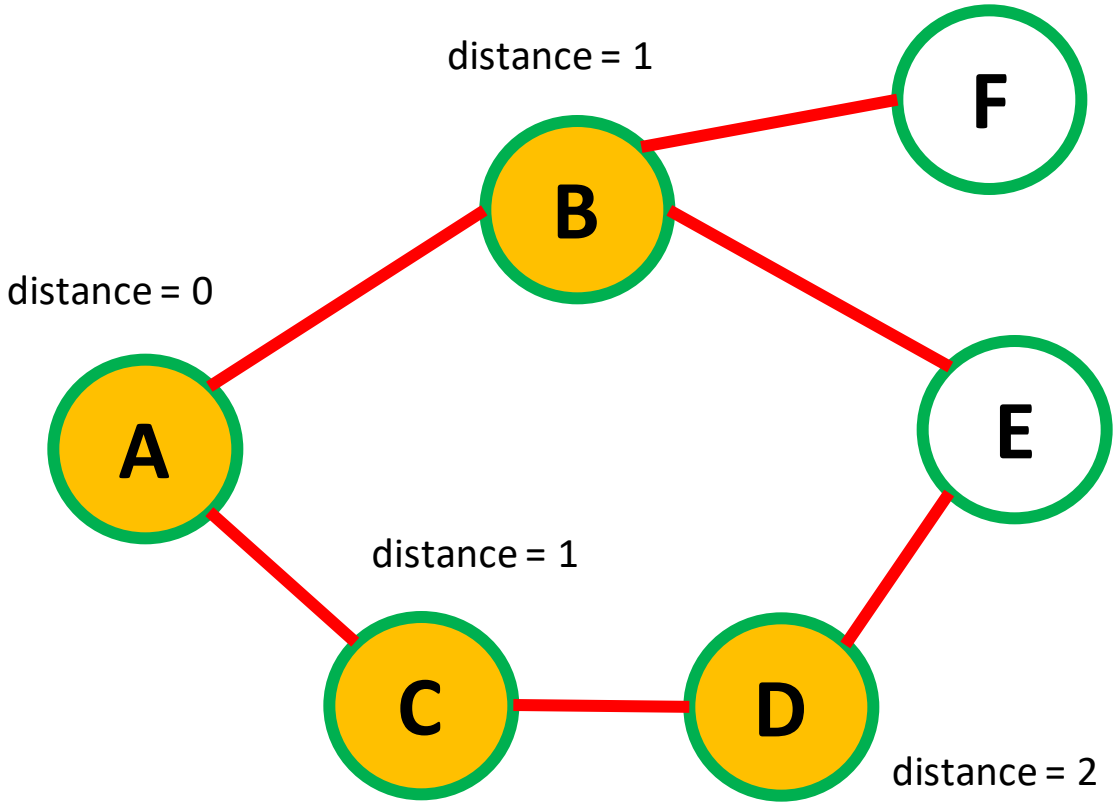
- Shortest path between **A** and **E**



Queue

Finding the shortest path (walkthrough)

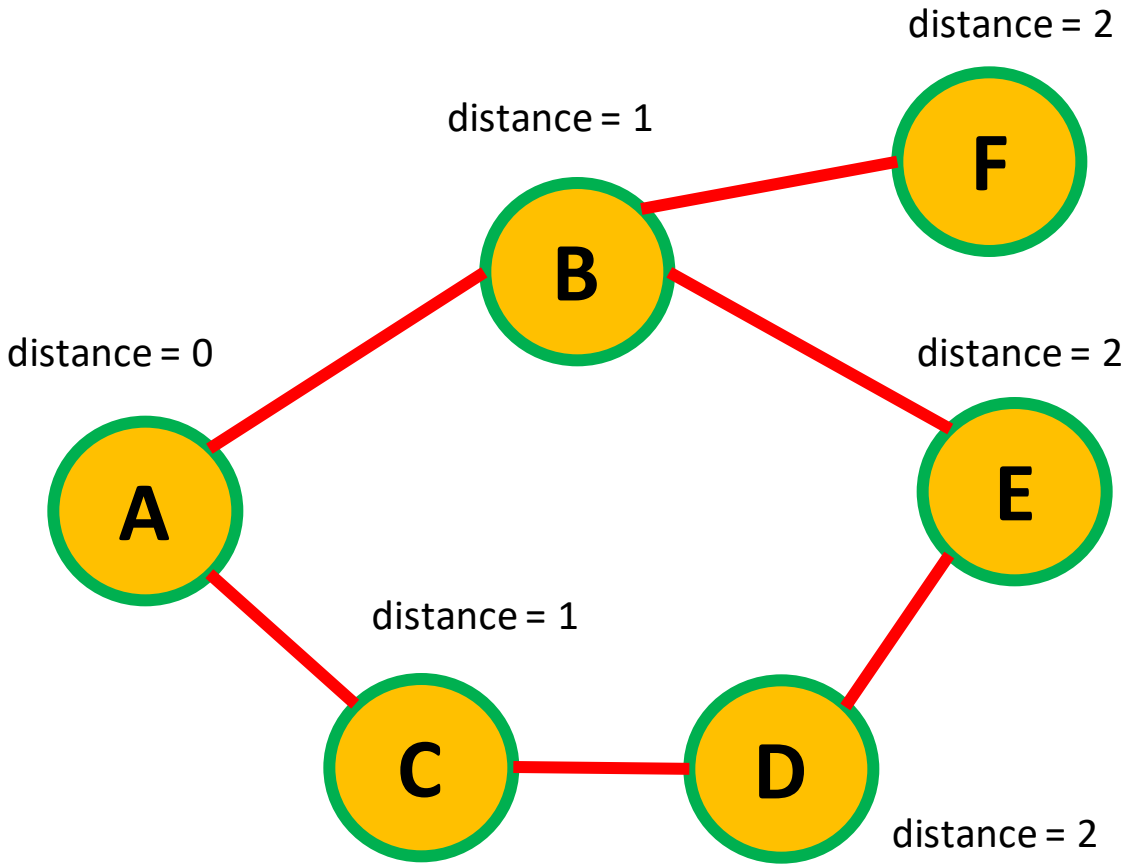
- Shortest path between **A** and **E**



Queue

Finding the shortest path (walkthrough)

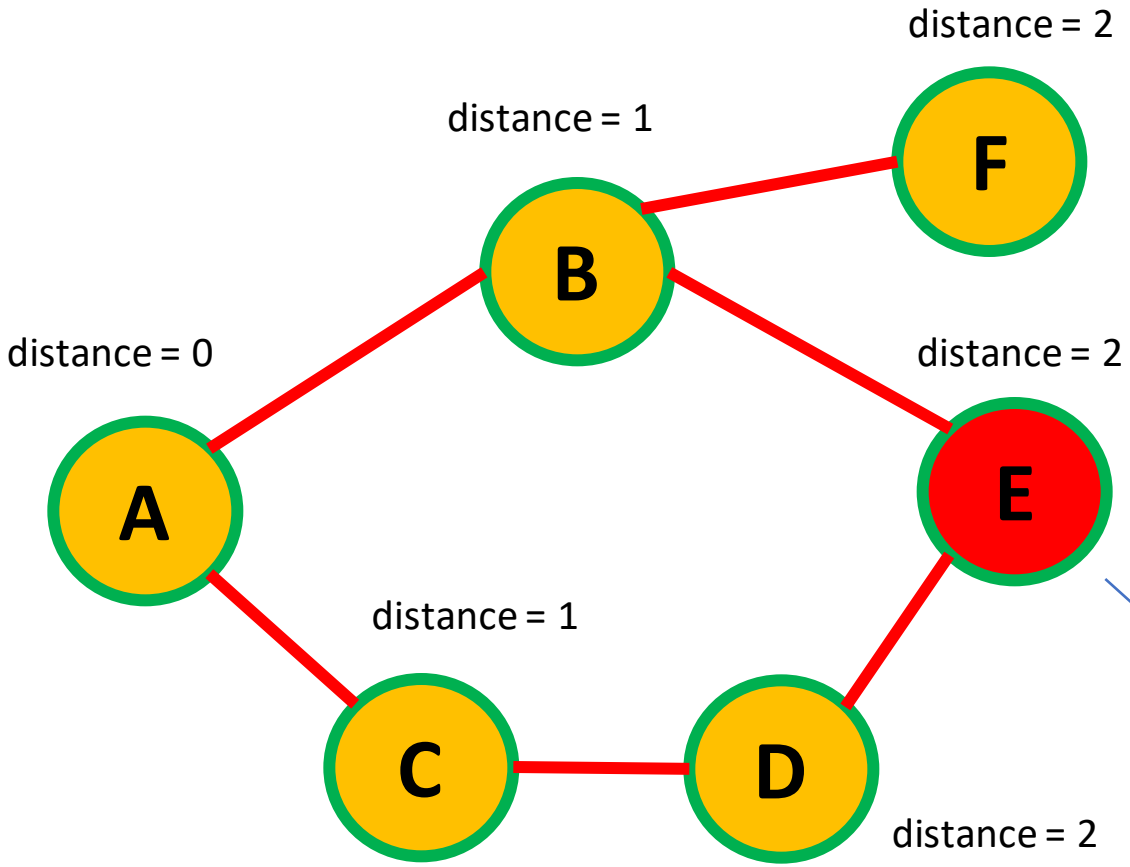
- Shortest path between **A** and **E**



Queue

Finding the shortest path (walkthrough)

- Shortest path between **A** and **E**: Distance = 2.

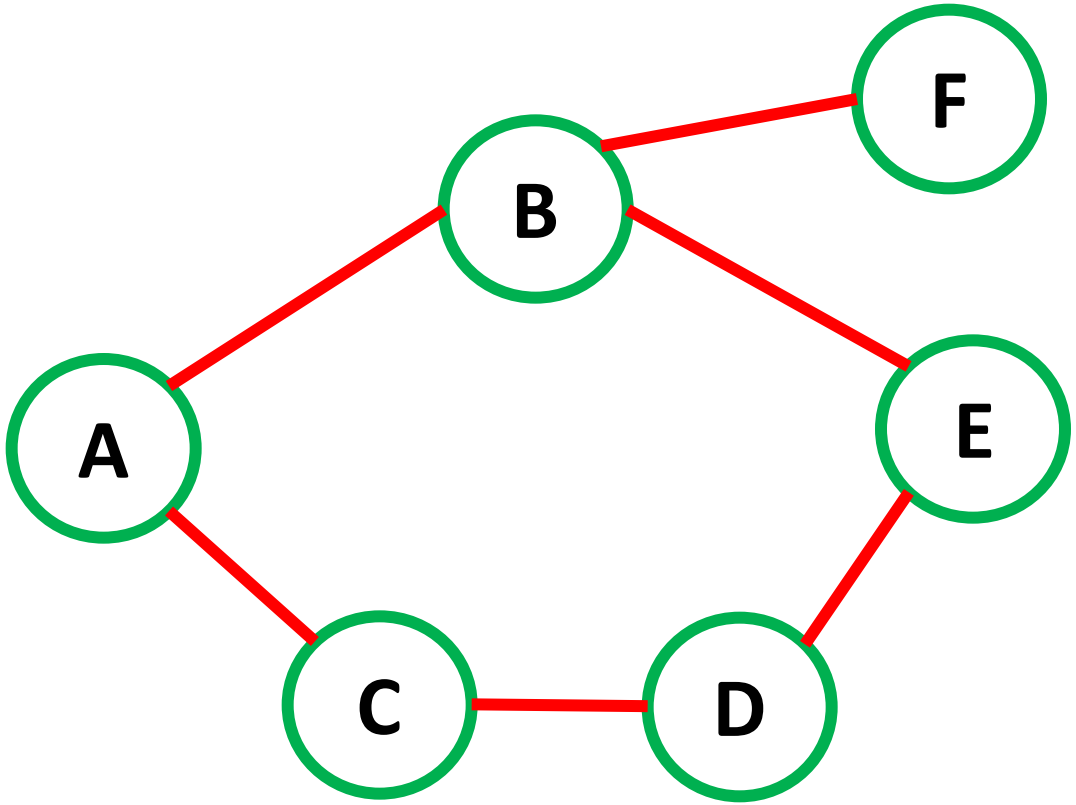


Queue

Target found!

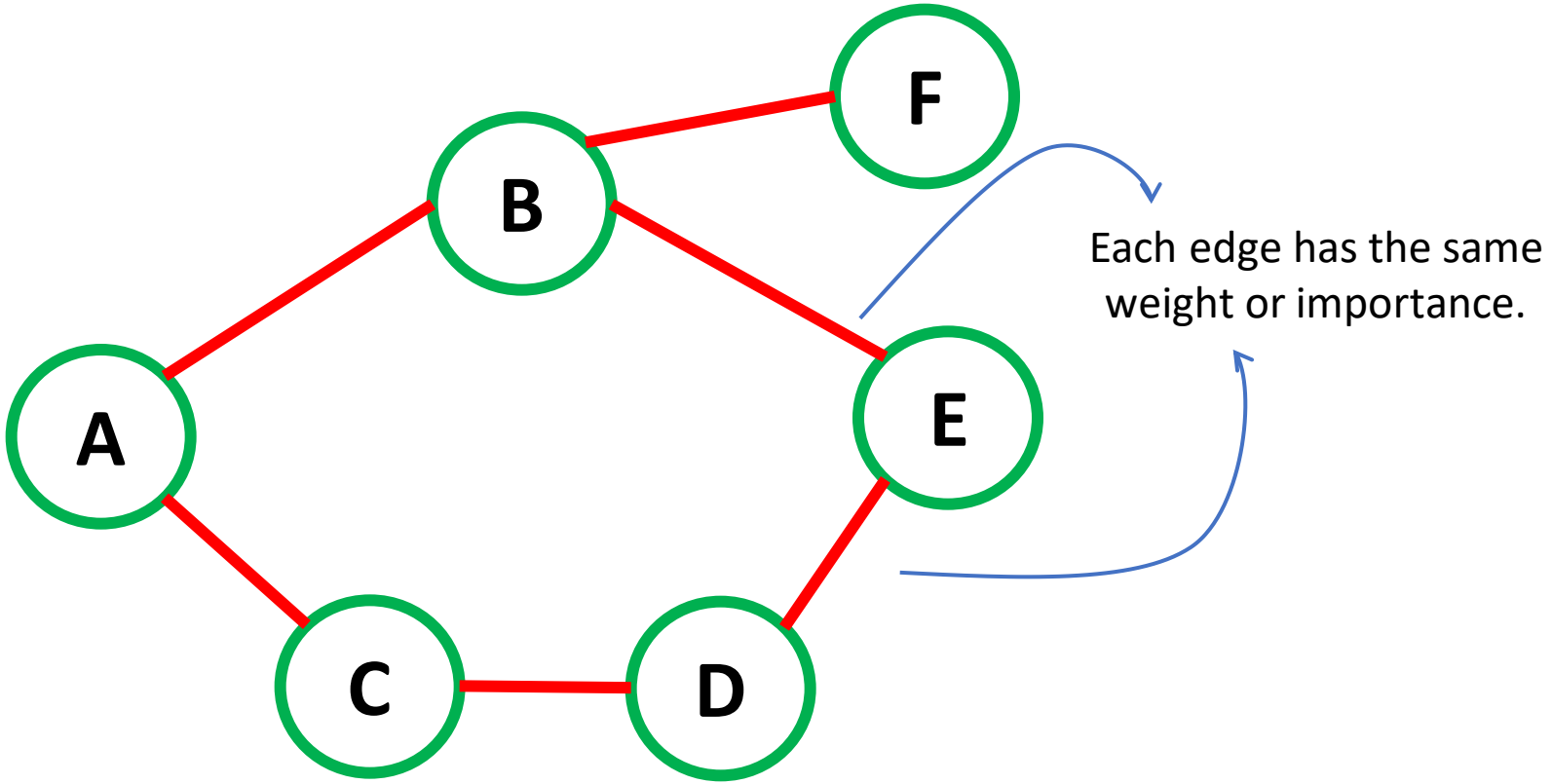
So is BFS good enough?

- Only if you are dealing with unweighted graphs.

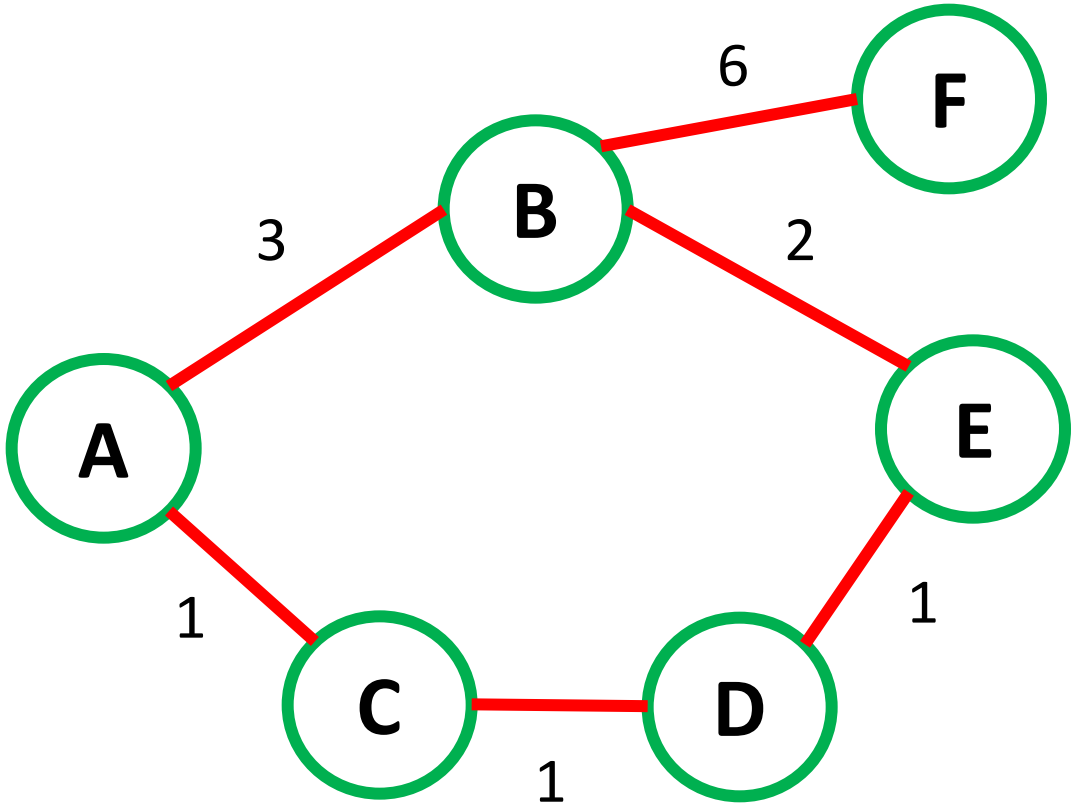


So is BFS good enough?

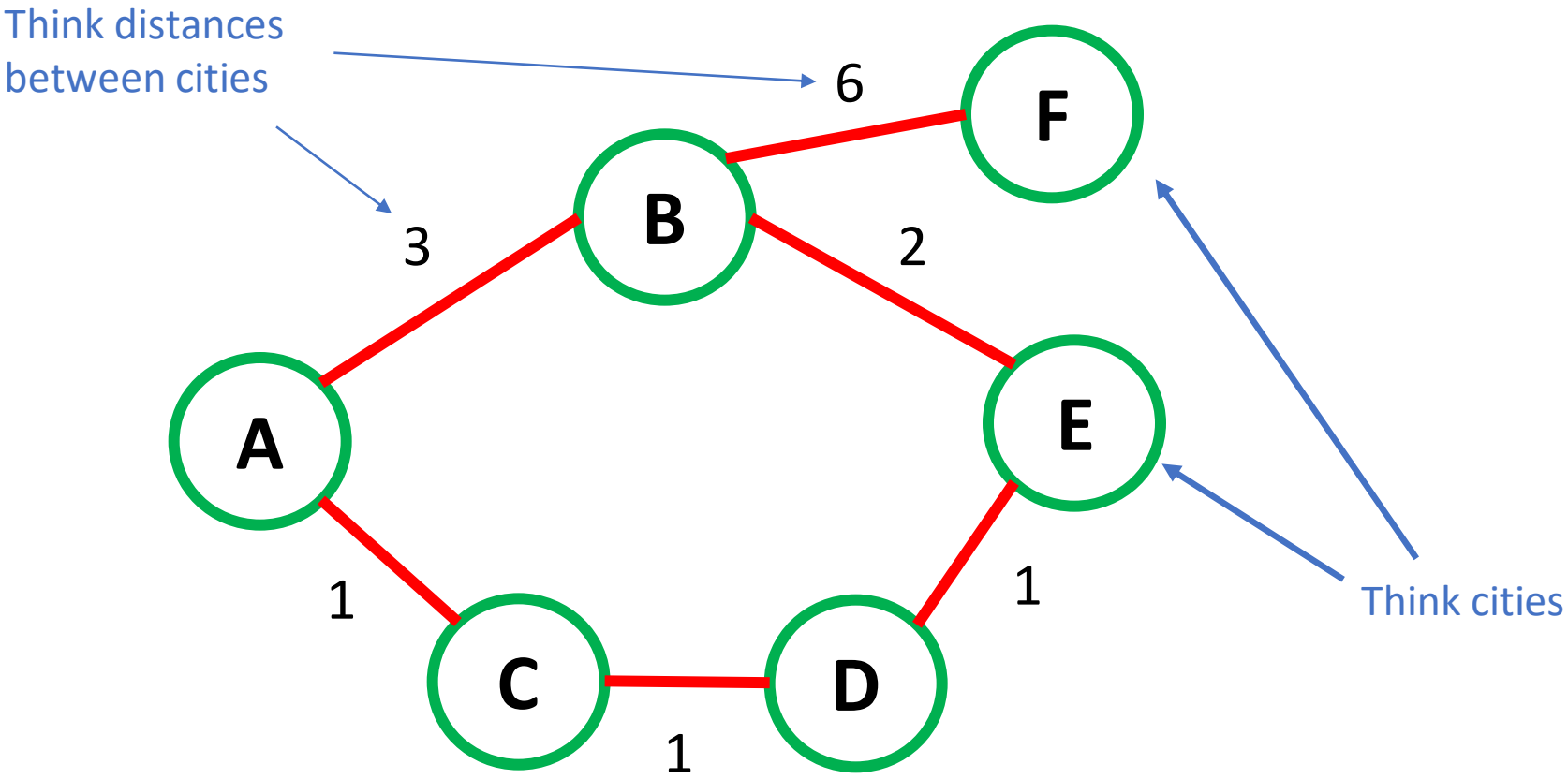
- Only if you are dealing with unweighted graphs.



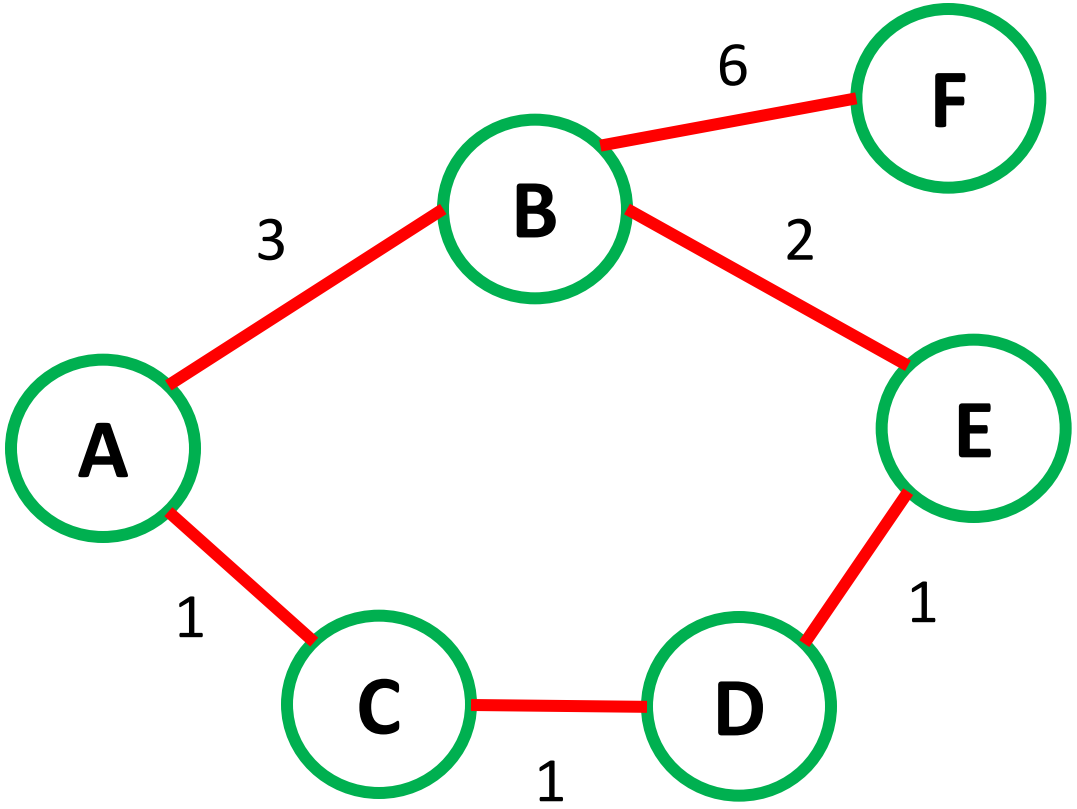
Enter Weighted Graphs



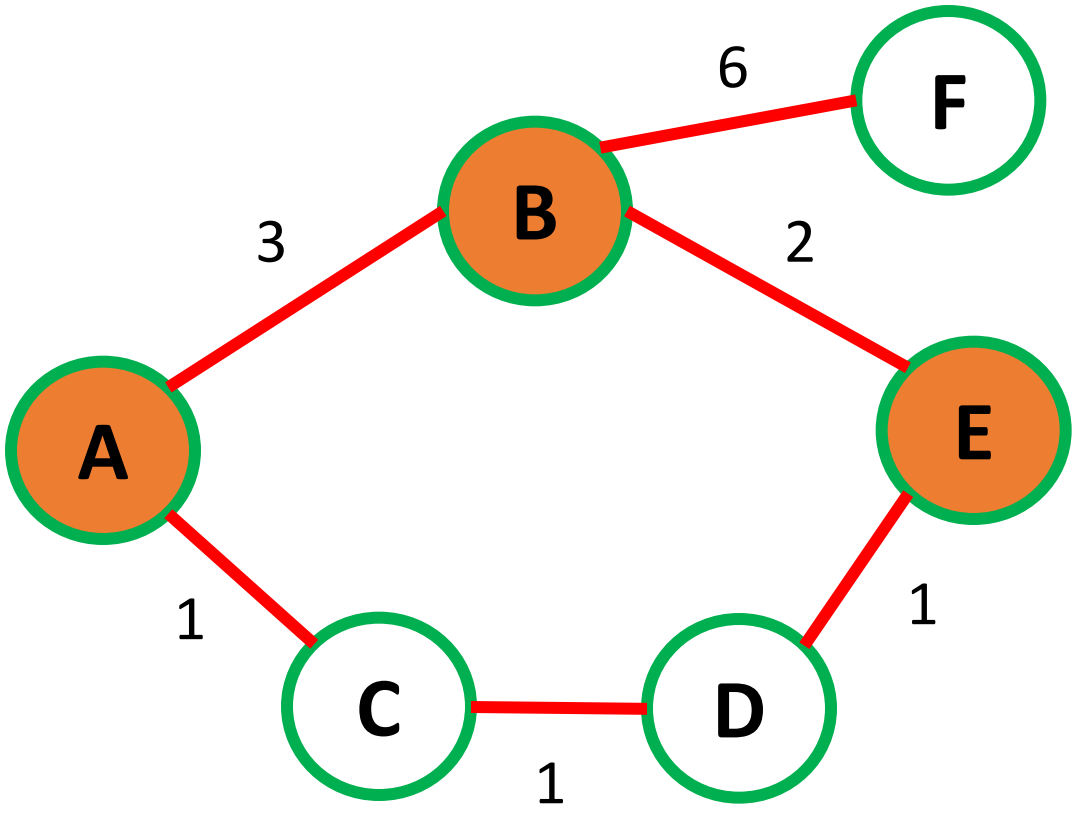
Enter Weighted Graphs



BFS on Weighted Graphs

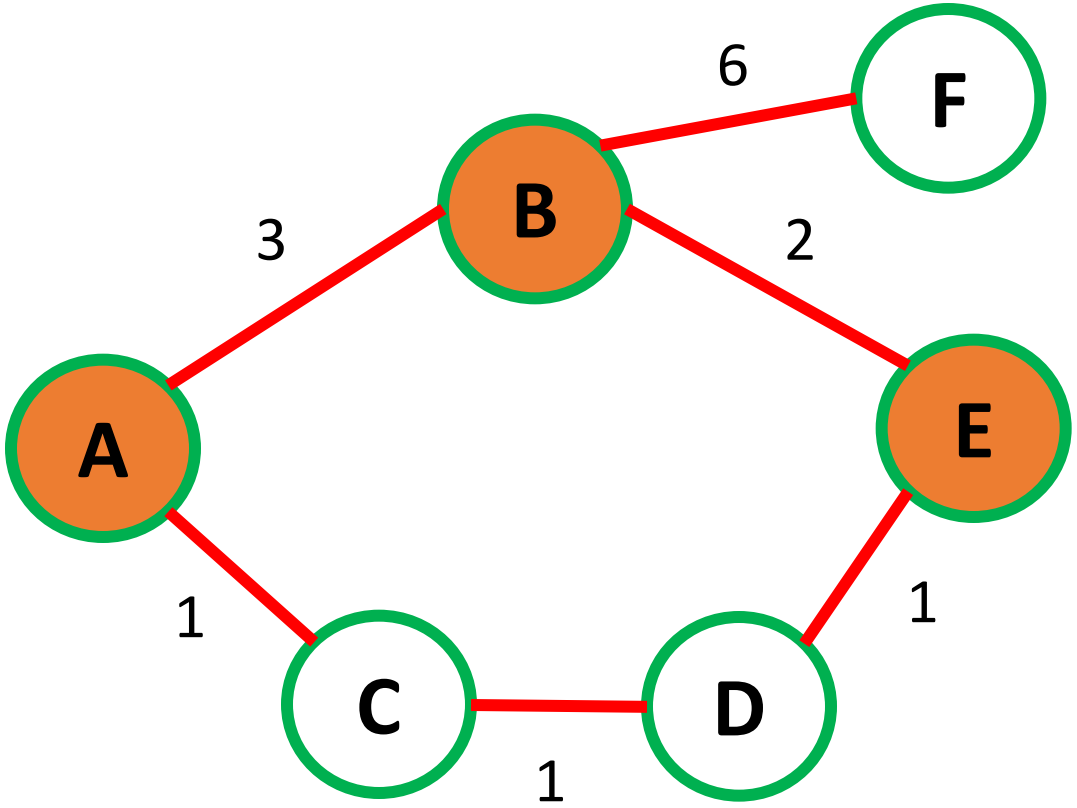


BFS on Weighted Graphs



- BFS gives ABE as the shortest path (Same algorithm as before).
- What is the path distance for ABE?
- Is it the shortest path from A to E?

BFS on Weighted Graphs



- BFS gives ABE as the shortest path (Same algorithm as before).
- What is the path distance for ABE?

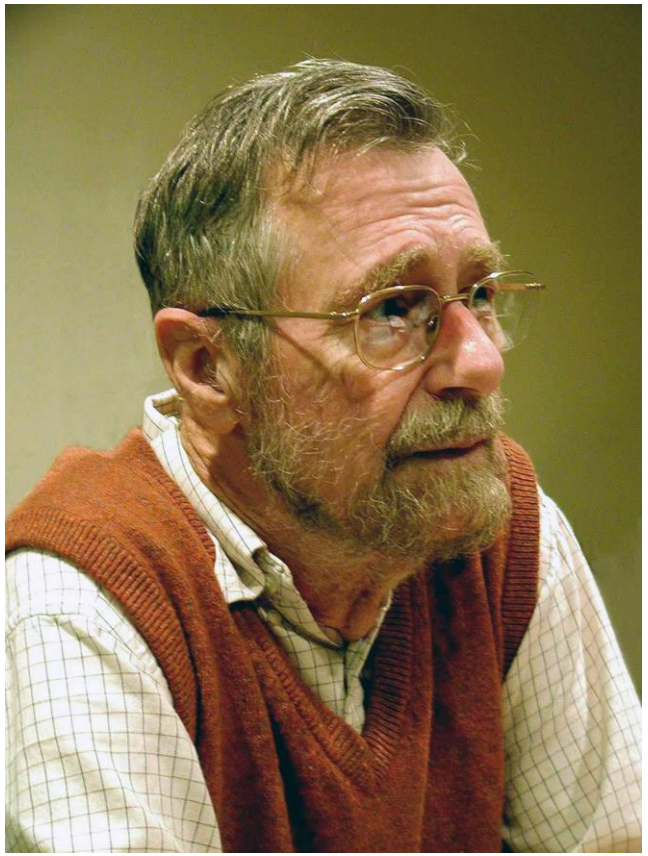
  5
- Is it the shortest path from A to E?

  No, the shortest path is ACDE with path length 3.

Dijkstra’s Algorithm

Sigismund Dijkstra (Witcher 3: The Wild Hunt)

Dijkstra's Algorithm



Sorry Witcher fans, it's actually named after this character – Edsger Dijkstra.

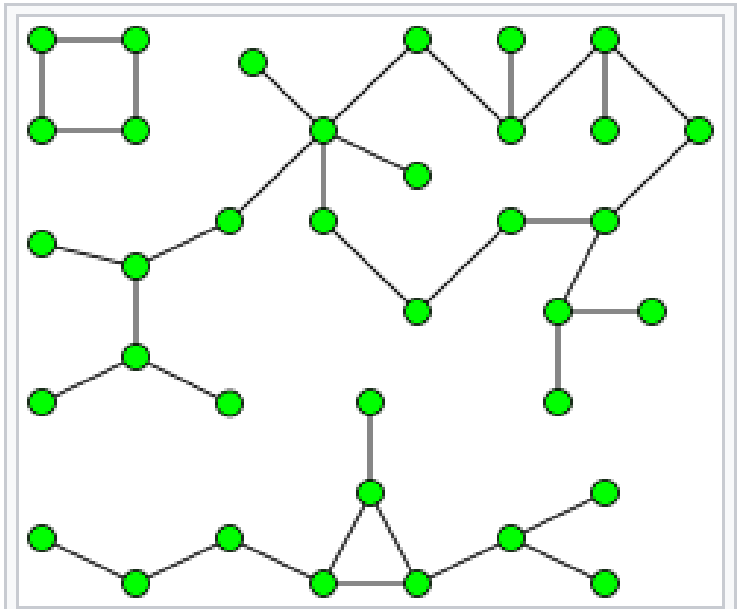
"Computer science is no more about computers than astronomy is about telescopes."
- Edsger Dijkstra

Dijkstra's Algorithm

- Allows you to find the shortest path in weighted graphs.
- Iteratively keeps track updates the shortest path from each node to the source.
- Will be covered in detail in the class, but I will review in the next recitation.

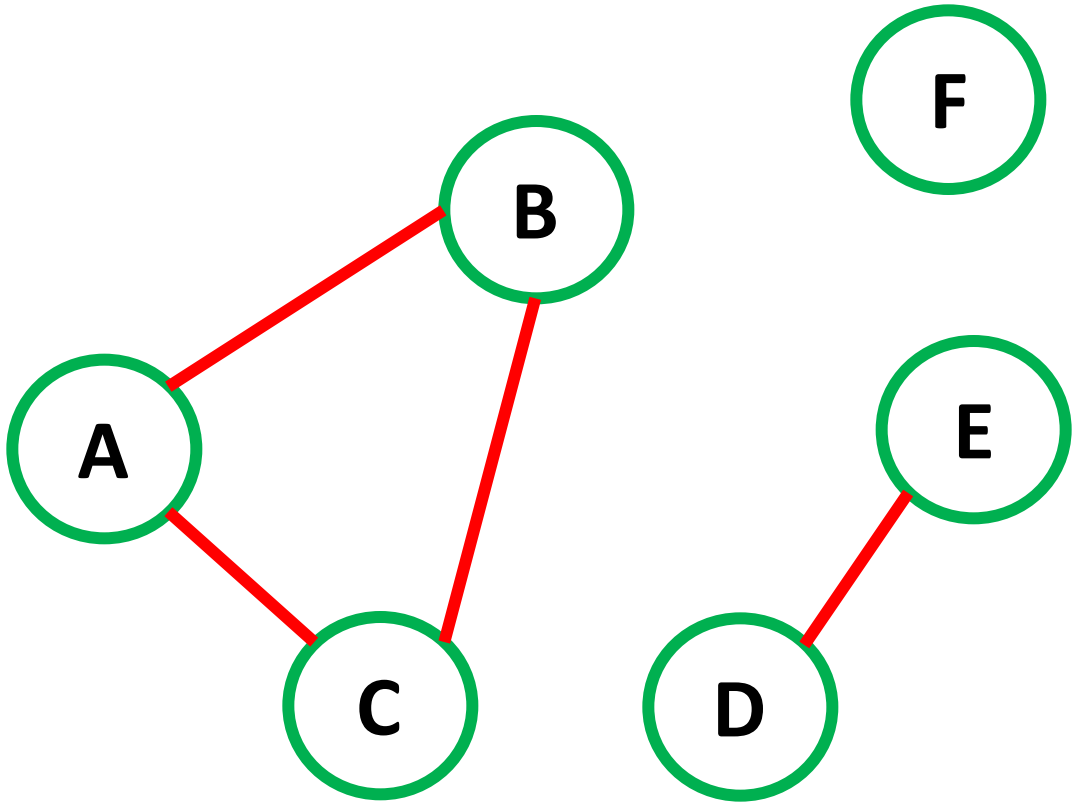
What are connected components?

- Individual pieces of the graph which are disconnected from each other.
- More formally, a connected component is a subgraph of the original graph, with a set of vertices connected by a path to each other.
- Think islands in an ocean.

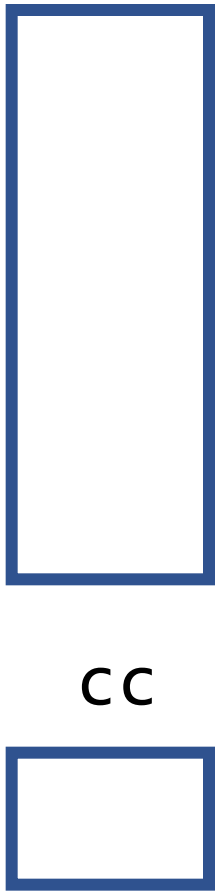


connected components using DFS (walkthrough)

- Find the total number of components



Stack

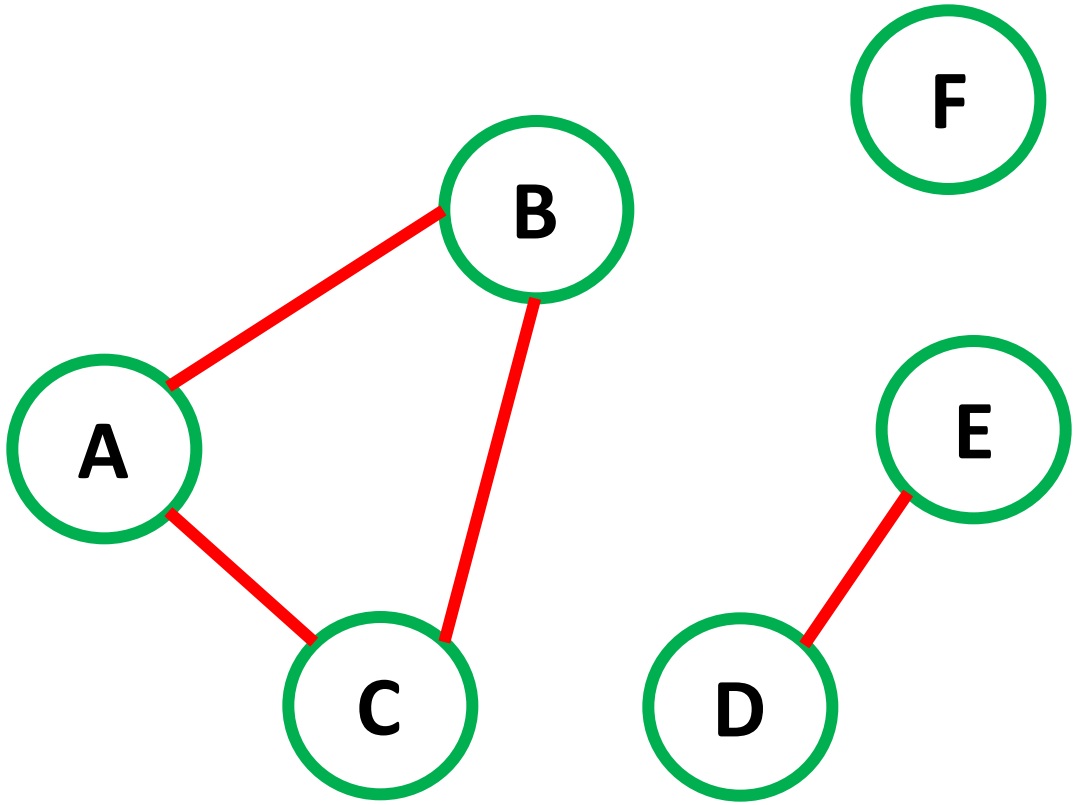


cc


```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
    for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack

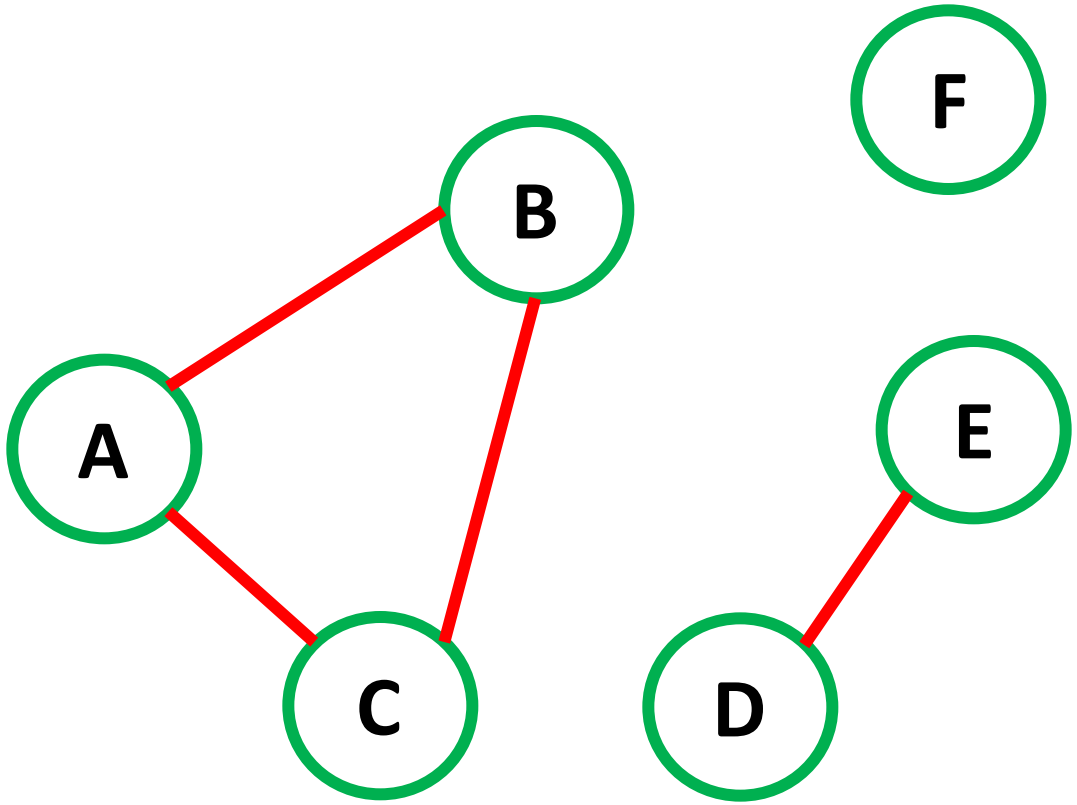
cc

0

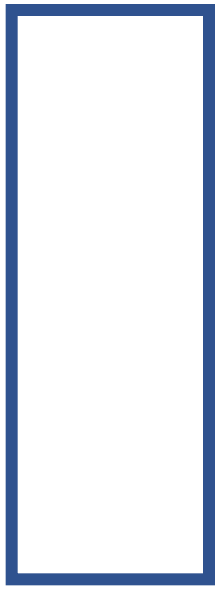
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
    for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



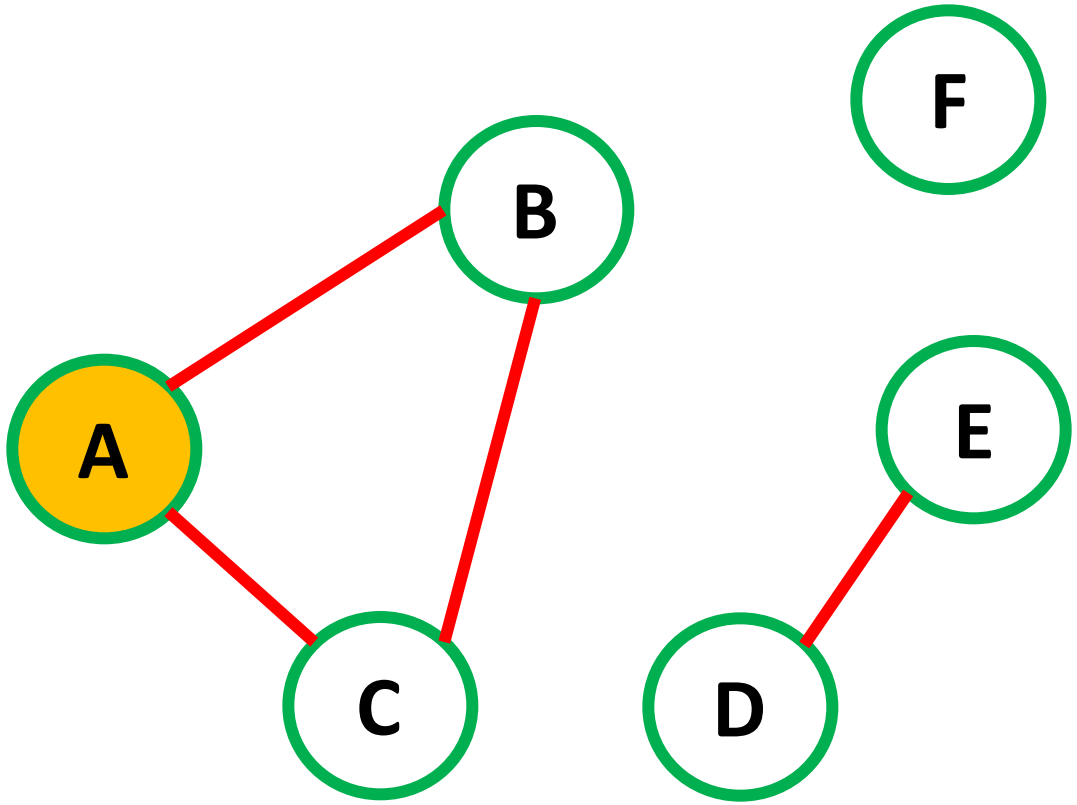
cc

0

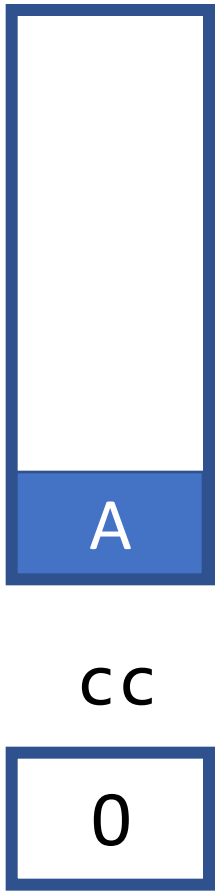
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



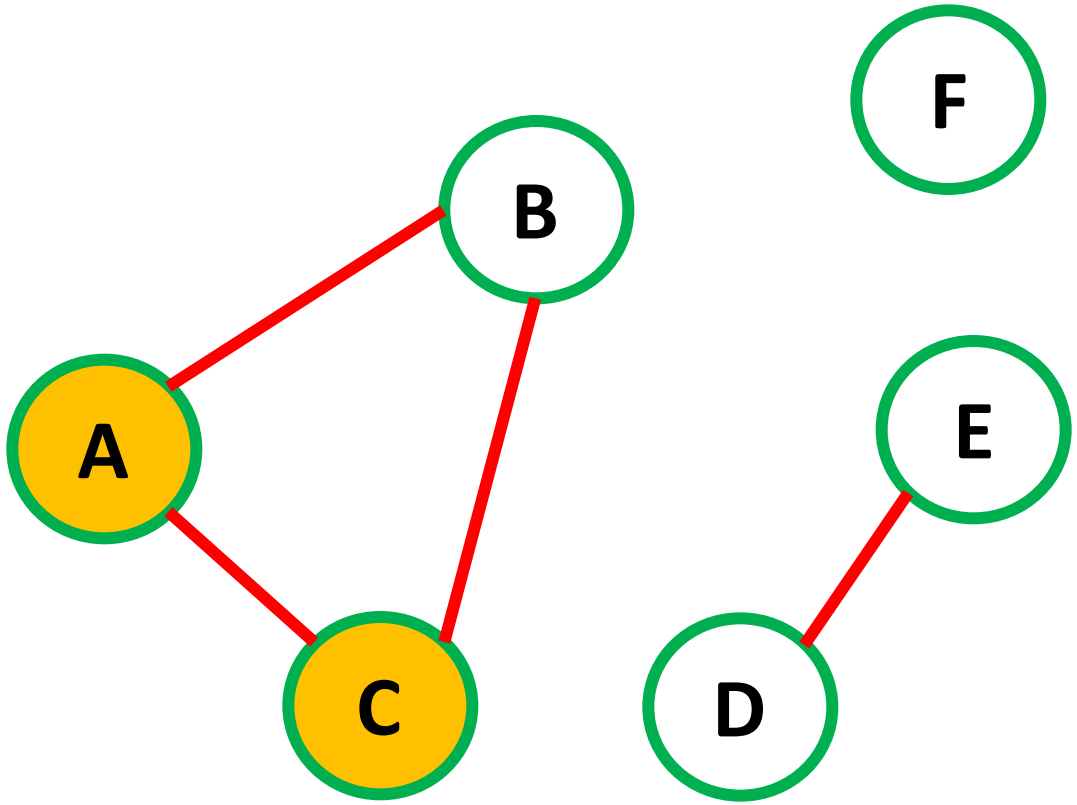
Stack



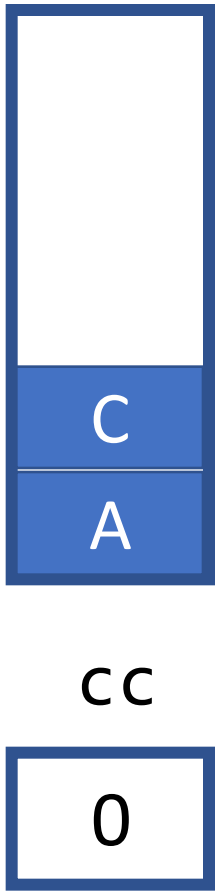

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack

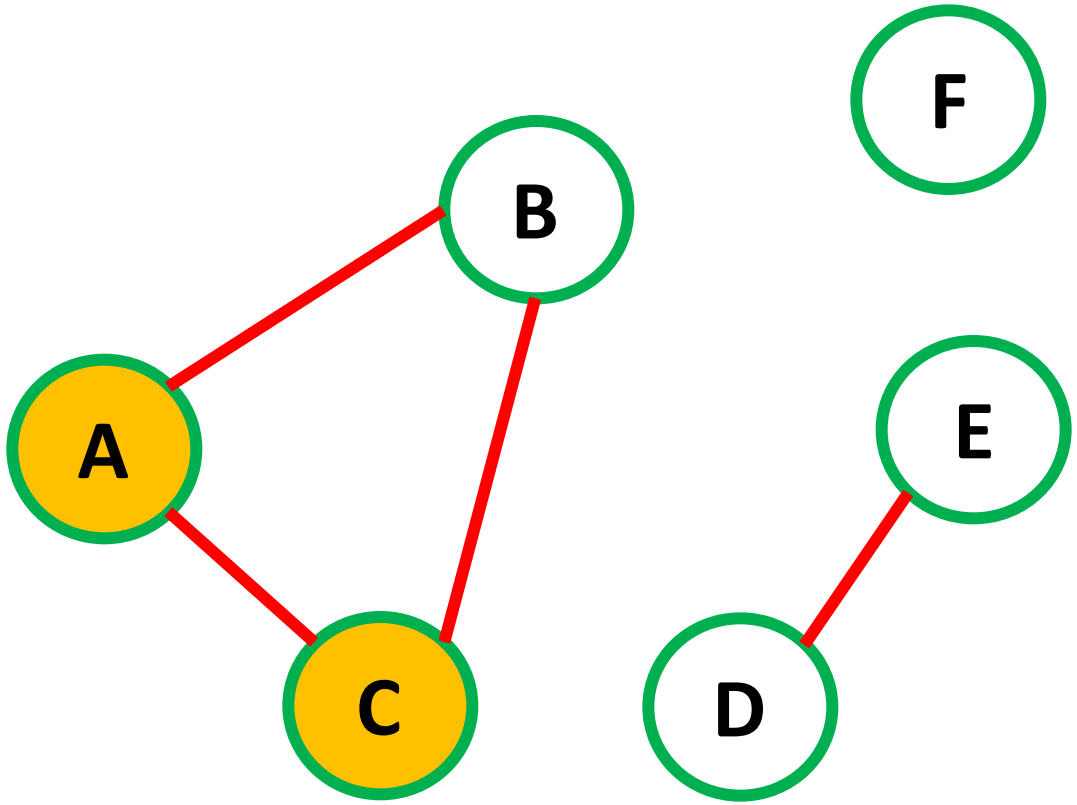


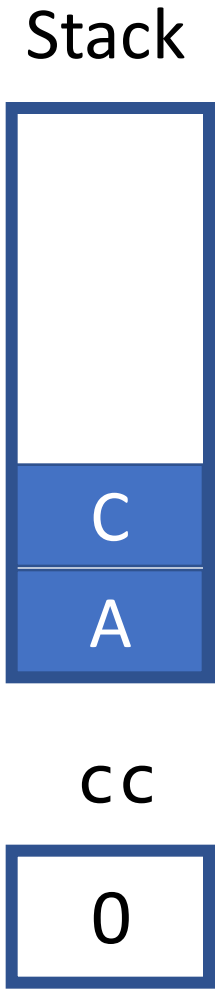
cc


```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components

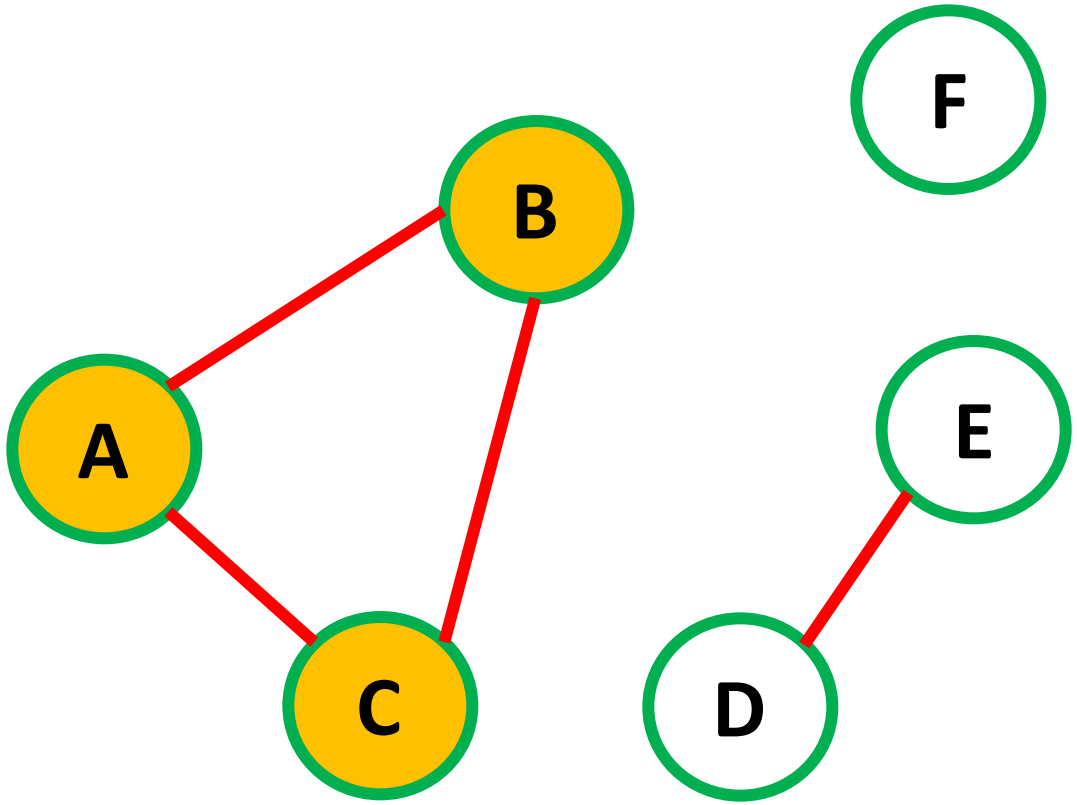


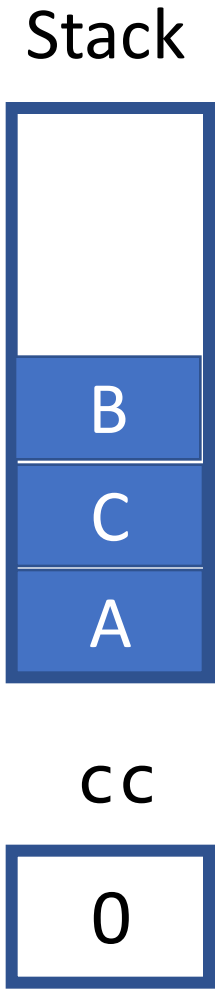



```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components

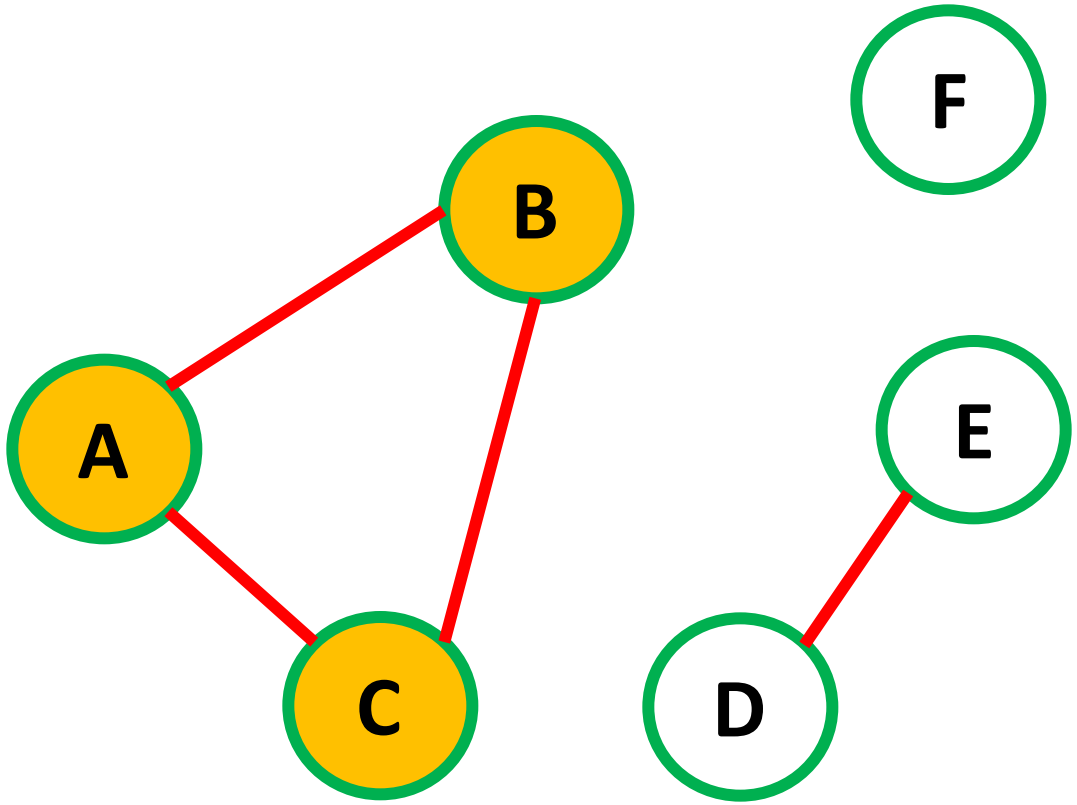


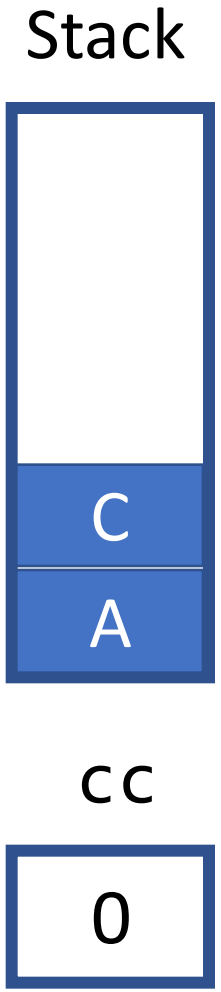



```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components

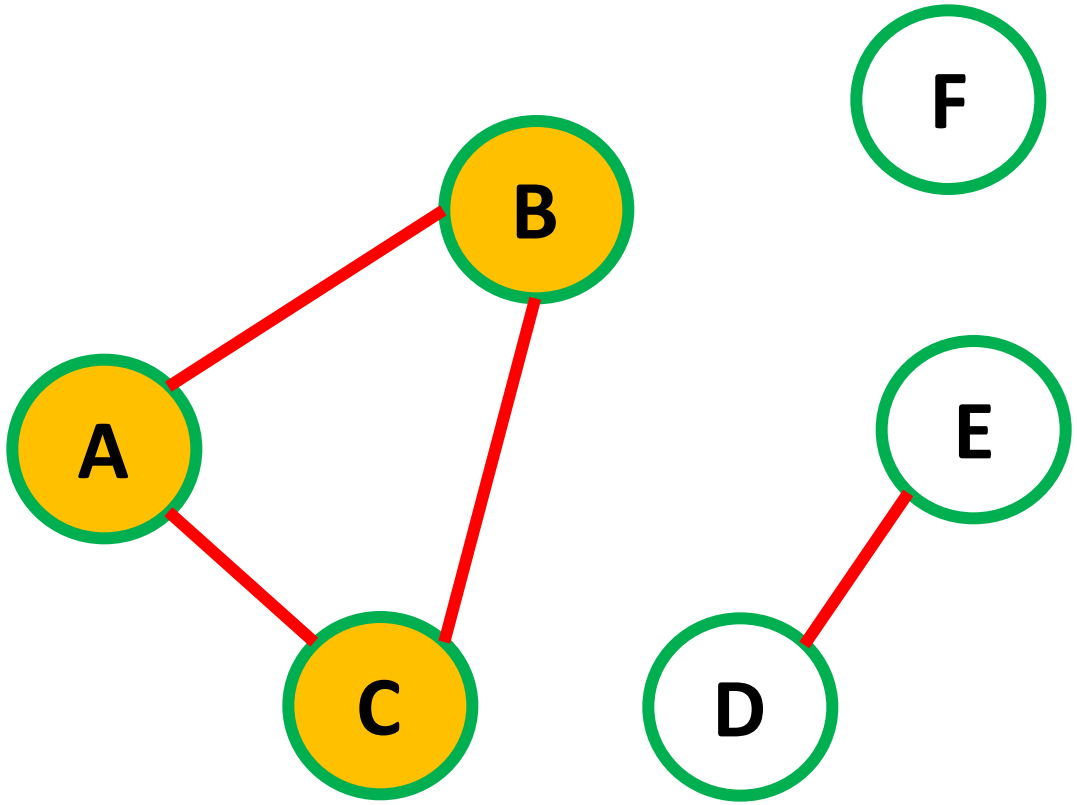


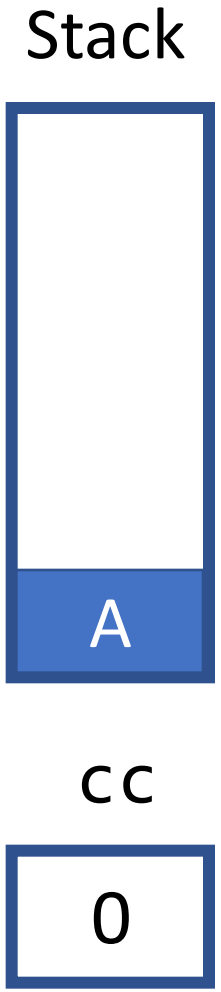



```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components

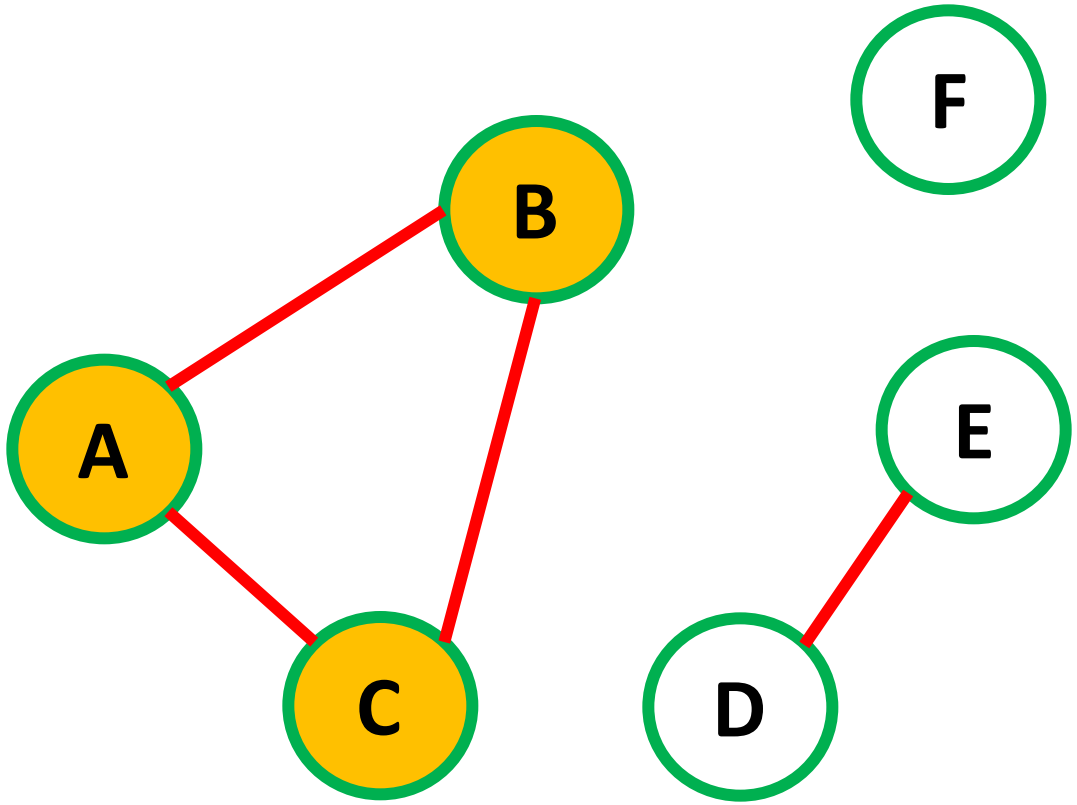





```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack

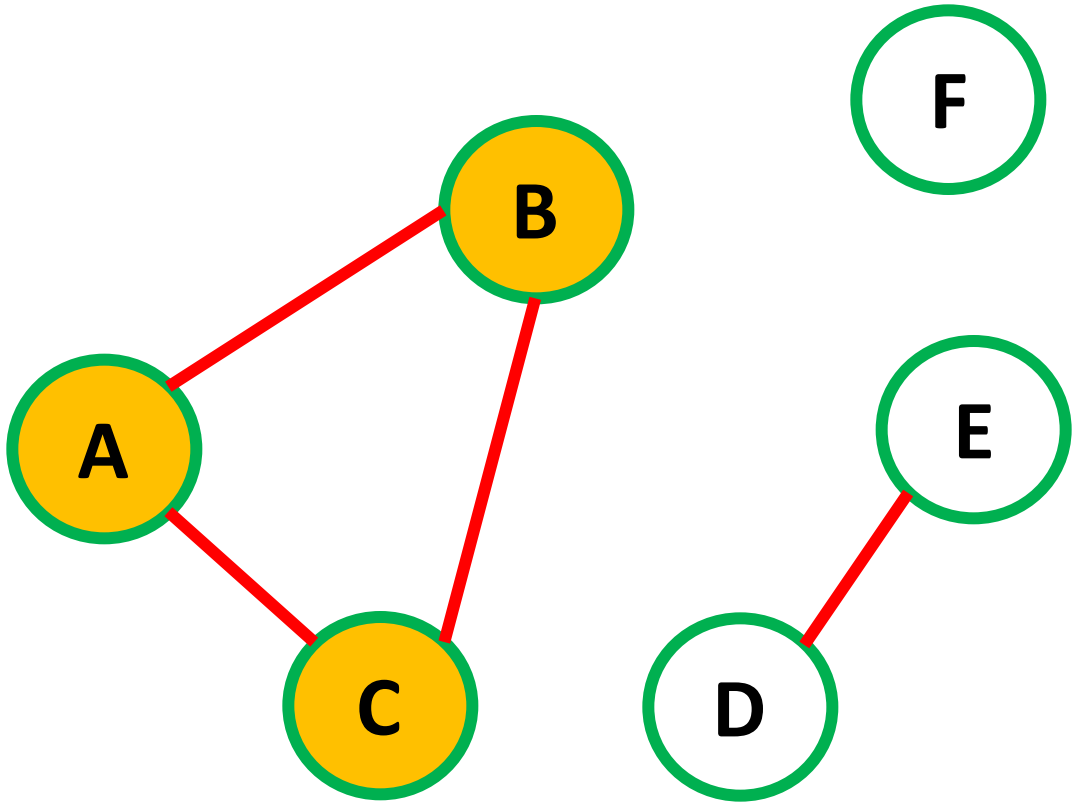
cc

0

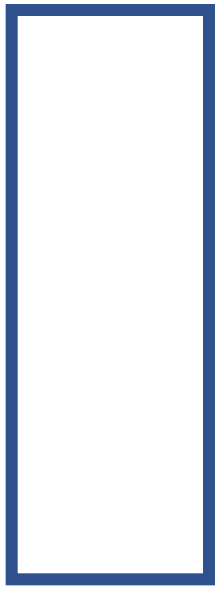
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



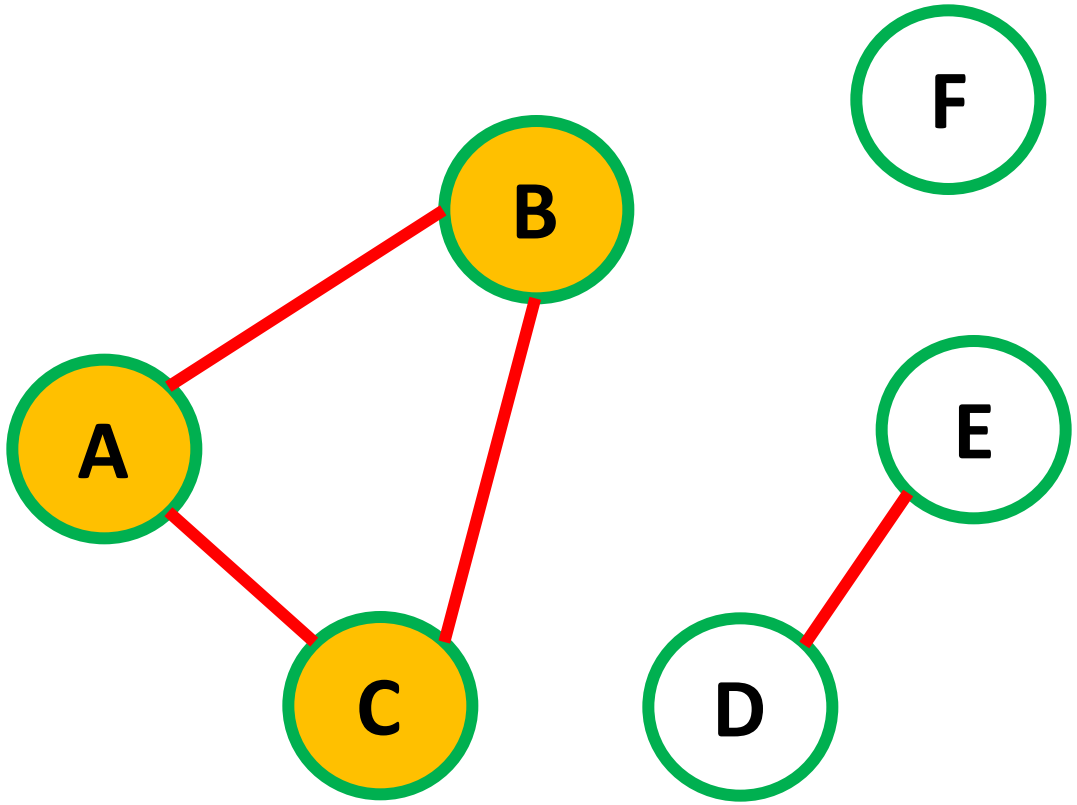
cc

1

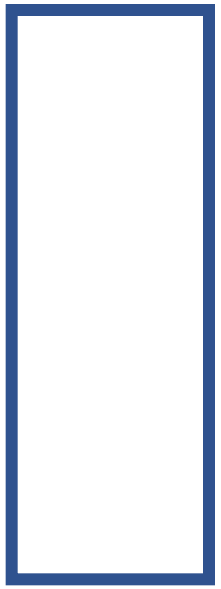
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=A for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



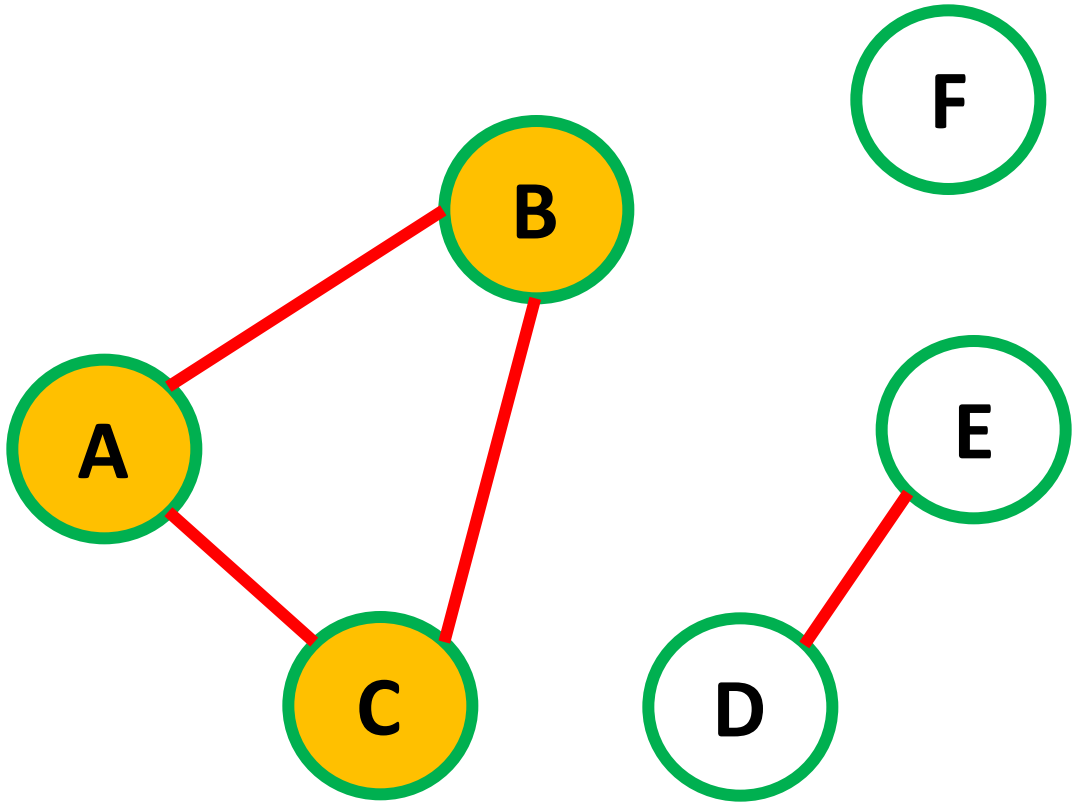
cc

1

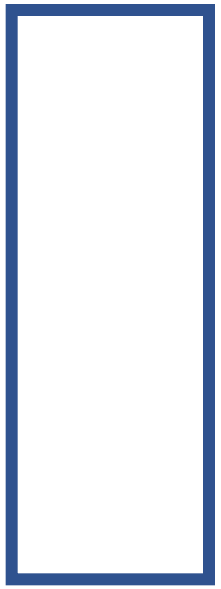
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=B for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



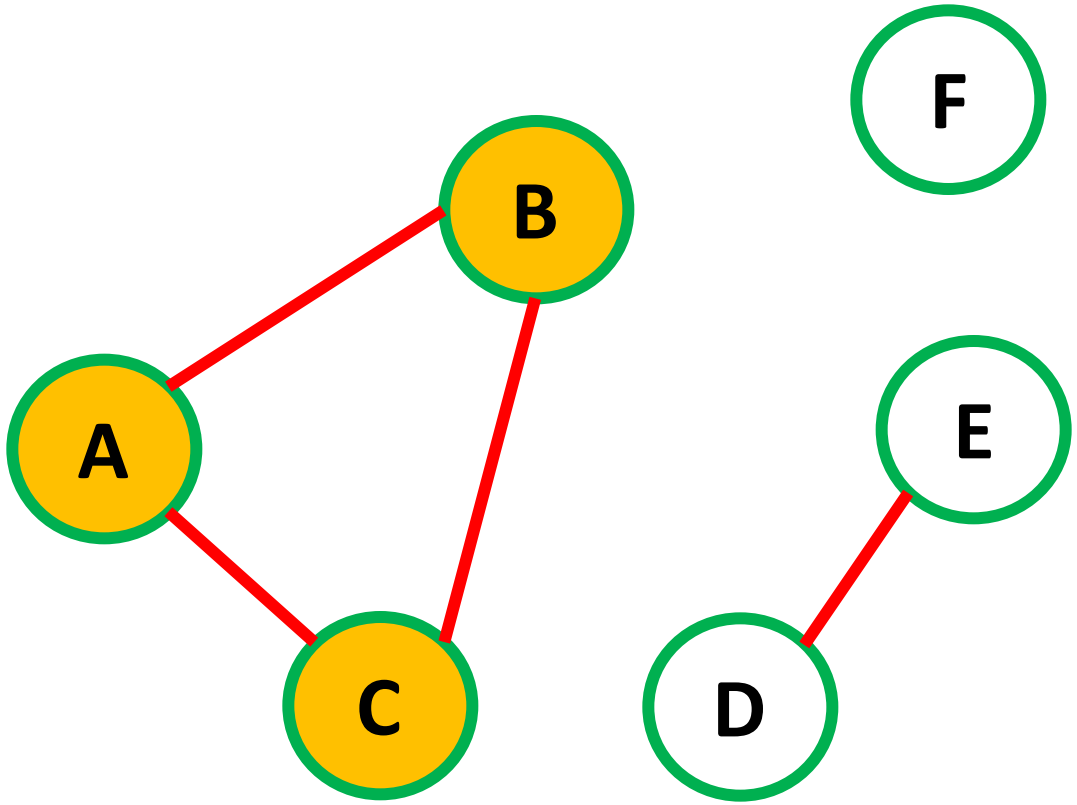
cc

1

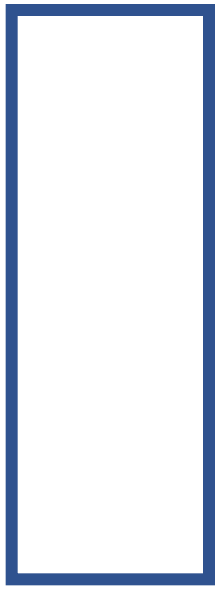
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=C for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



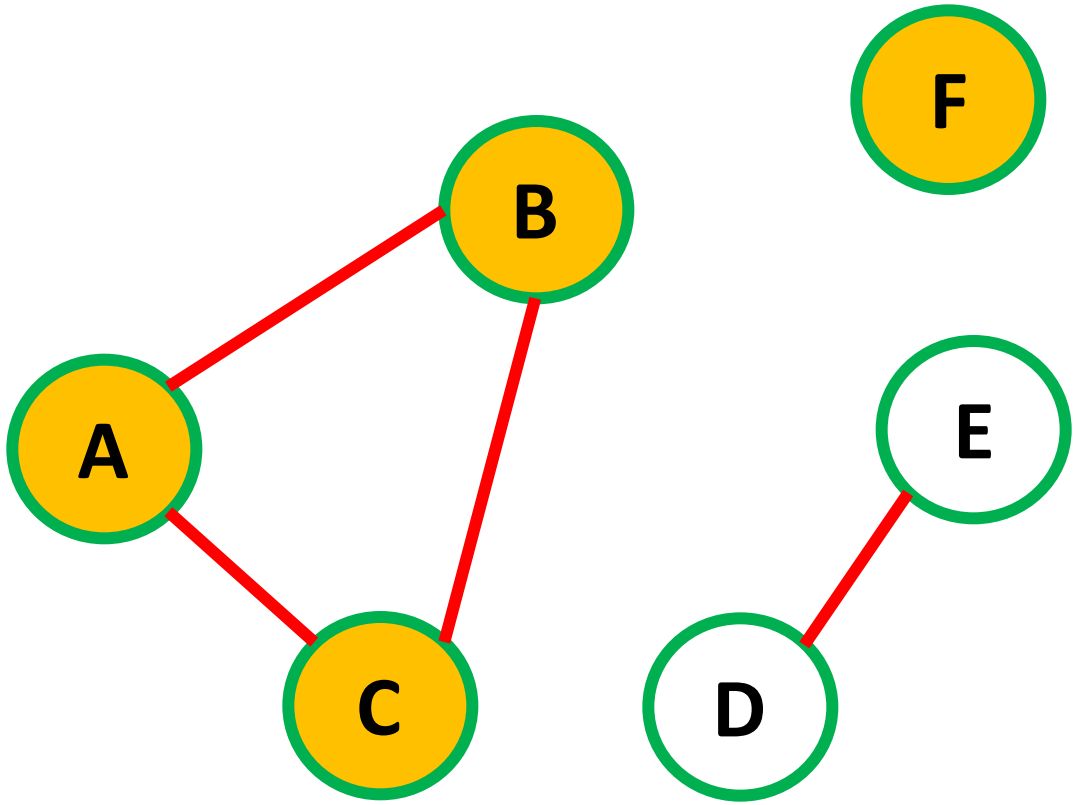
cc

1

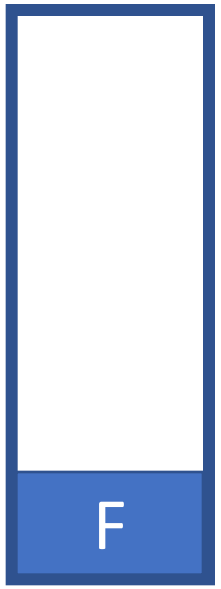
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=F for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

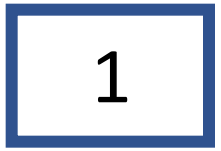
- Find the total number of components



Stack



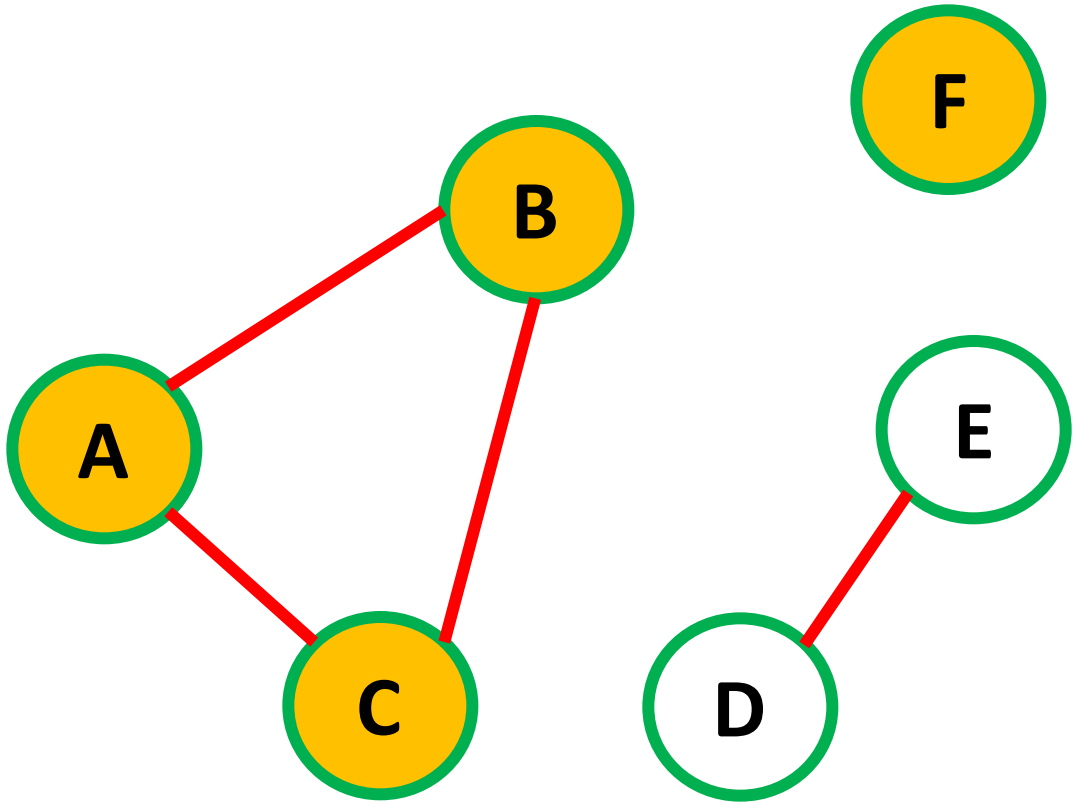
cc



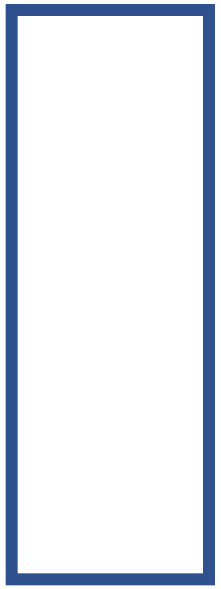

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=F for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



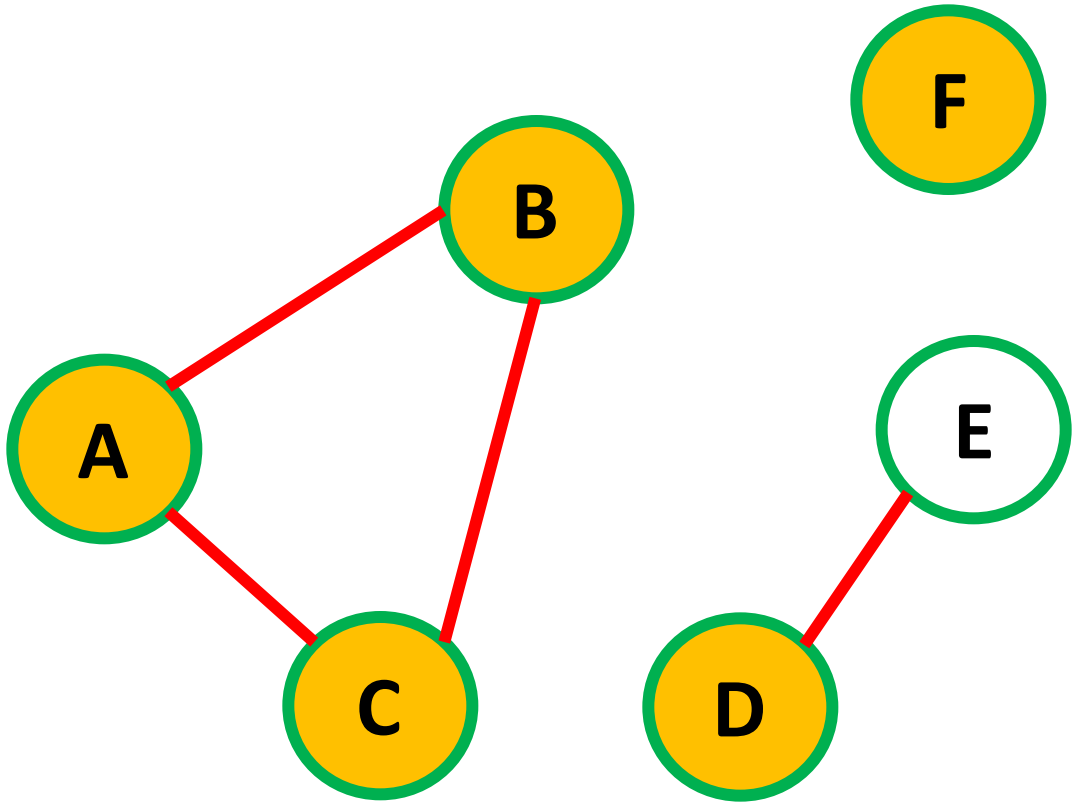
cc

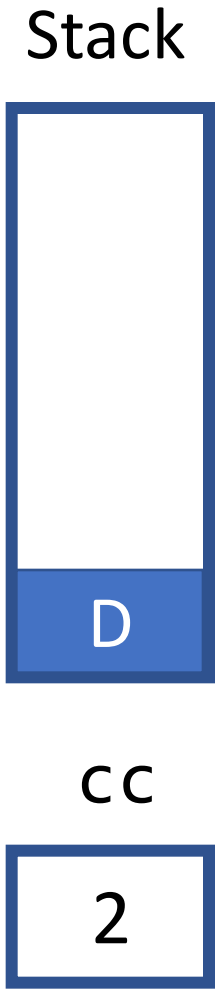
2

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=F for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components

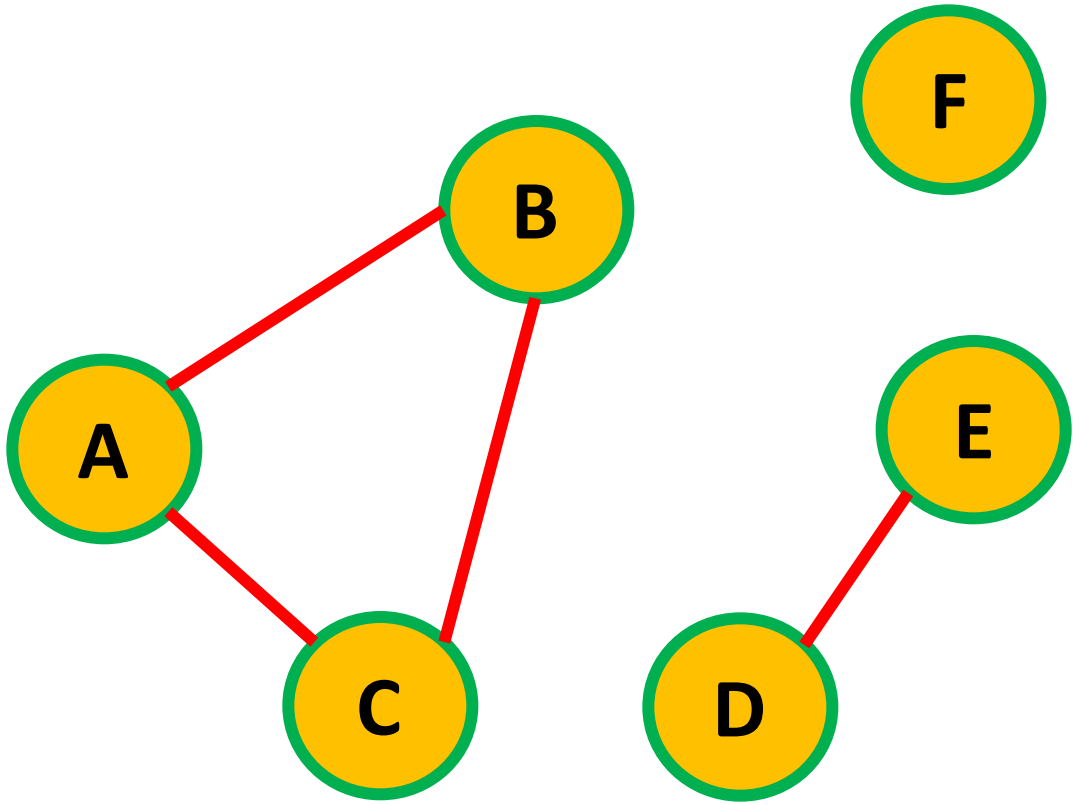




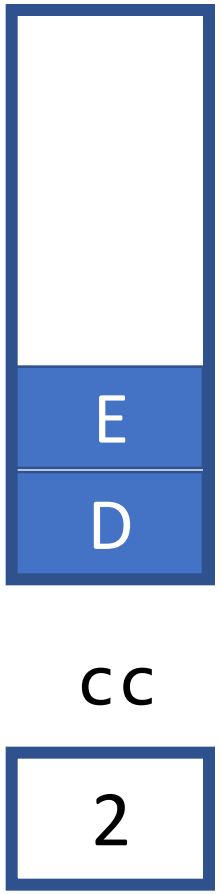

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=D for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



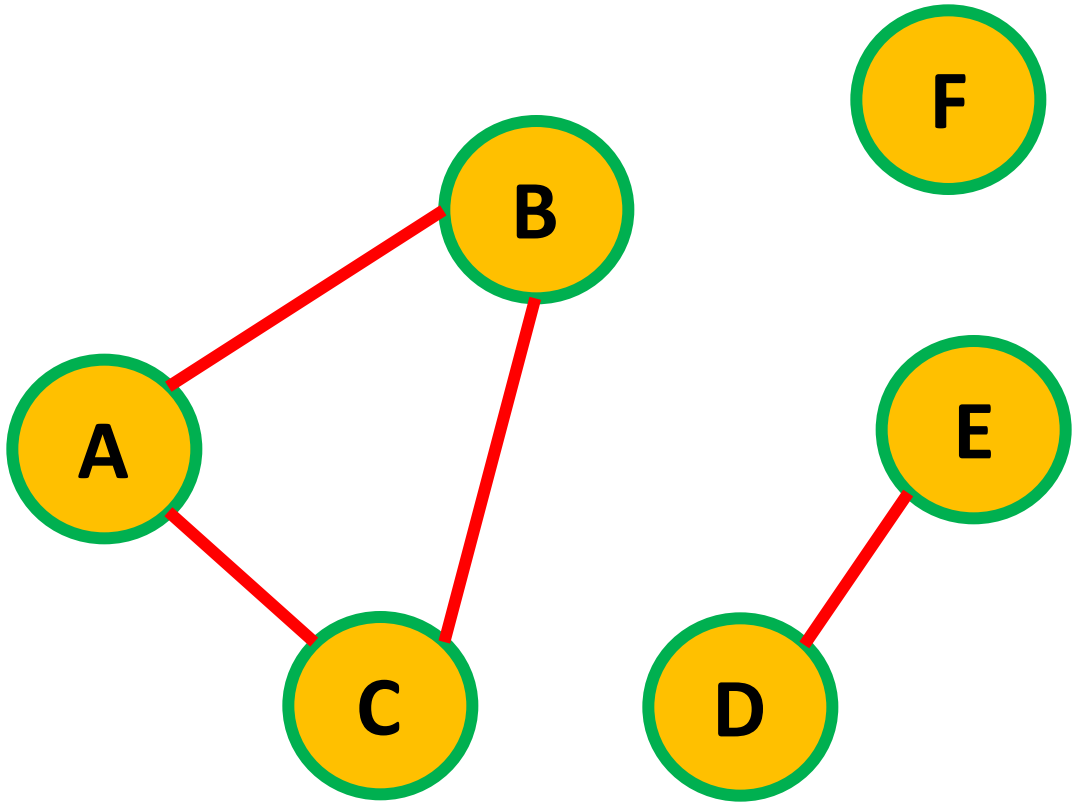
cc

2

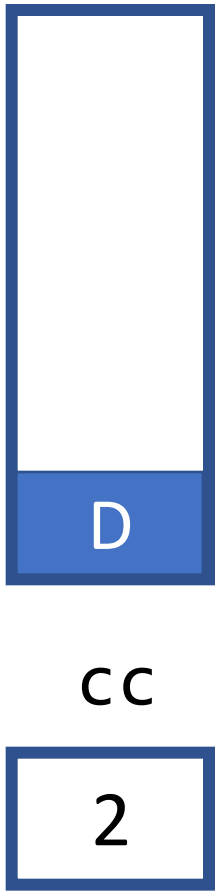
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=D for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



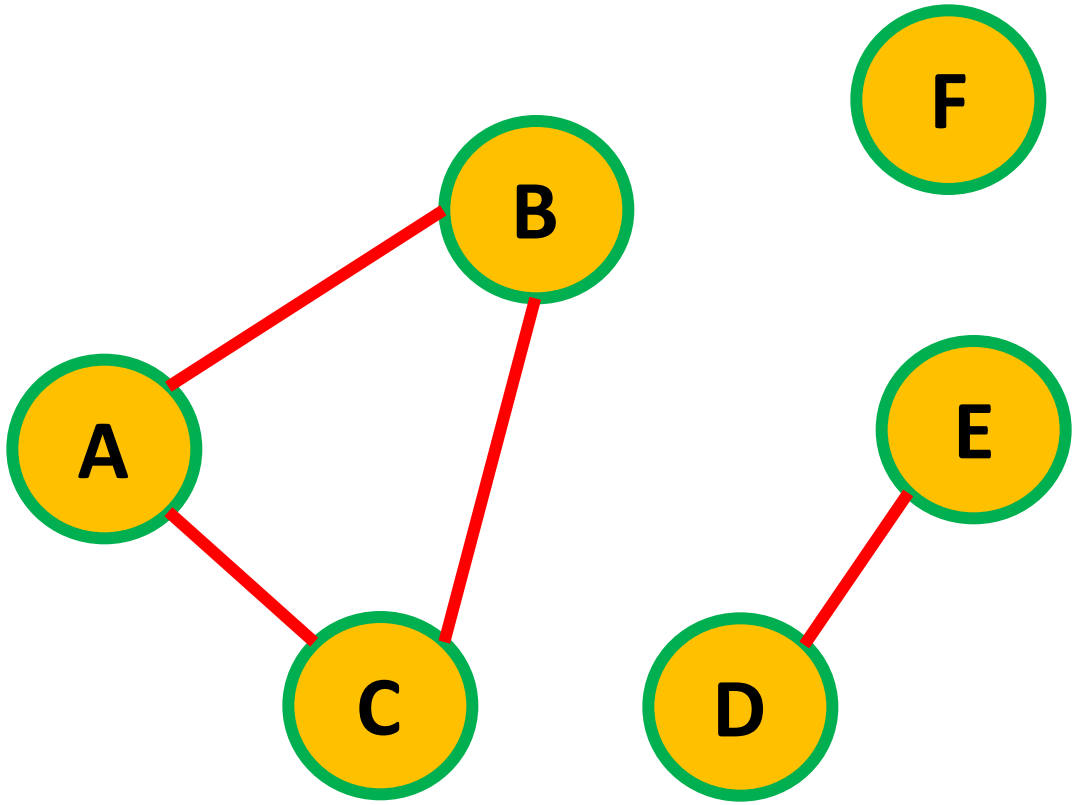
cc

2

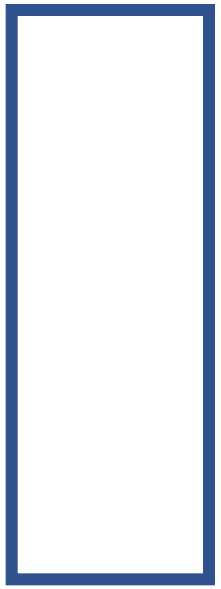
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=D for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

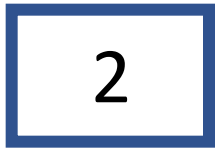
- Find the total number of components



Stack



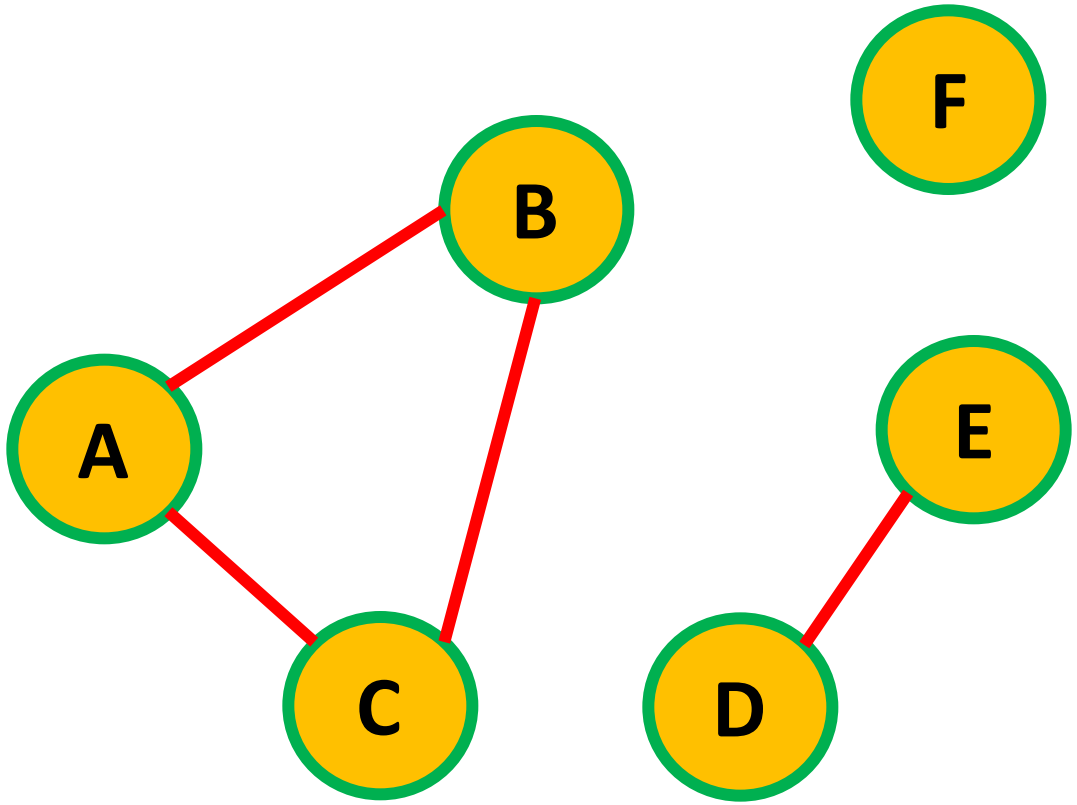
cc



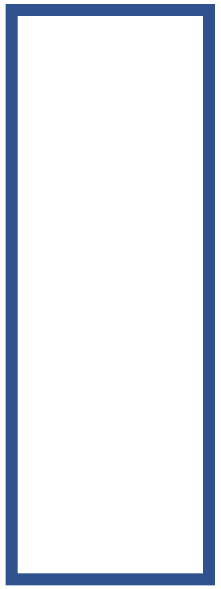

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=D for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



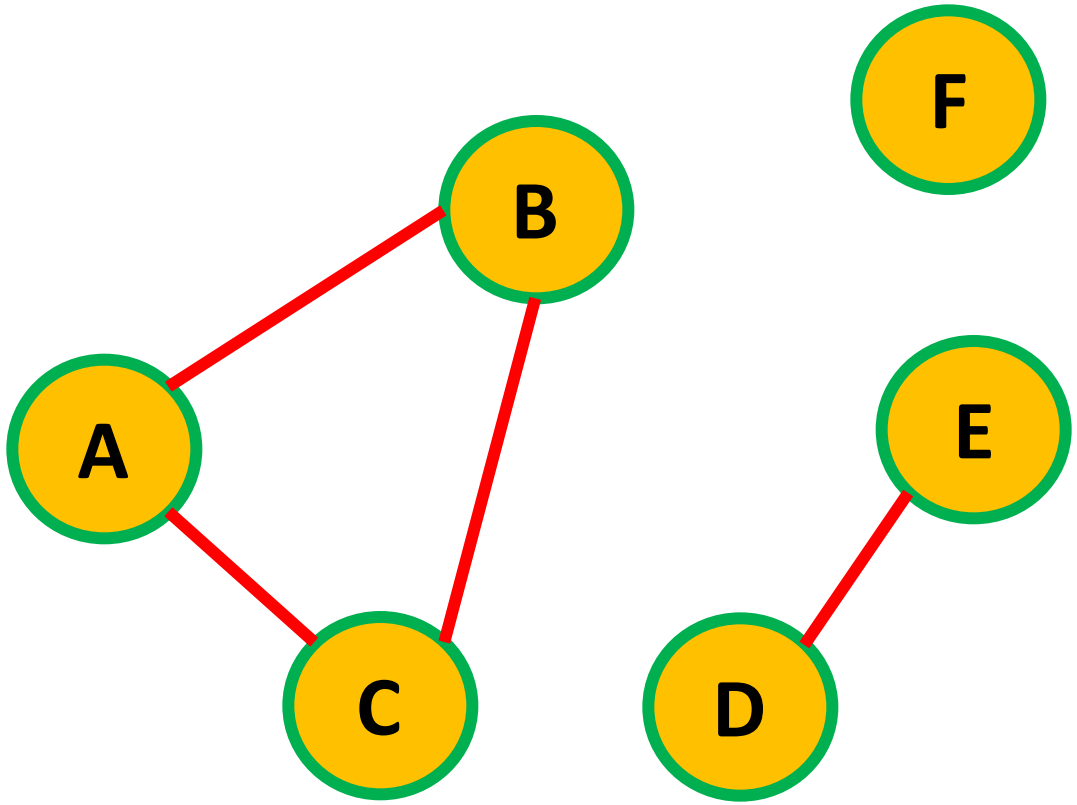
cc

3

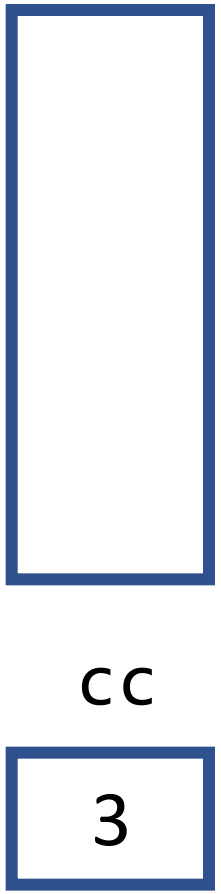
```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=D for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

connected components (walkthrough)

- Find the total number of components



Stack



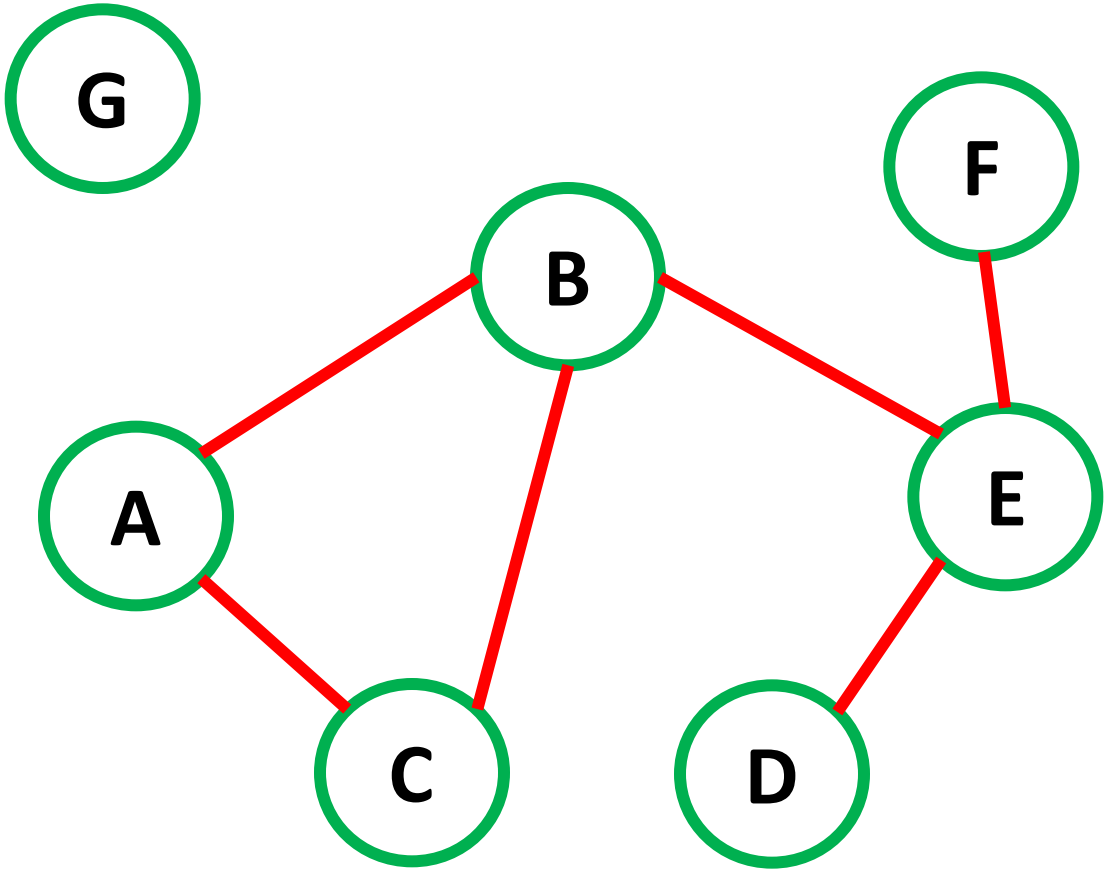
cc

3

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

ConnectedComponents() {
    cc = 0
    for each u ∈ G
        u.visited = false
u=E for each u ∈ G
        If u.visited==false
            DFS(G, u)
            cc += 1
}
```

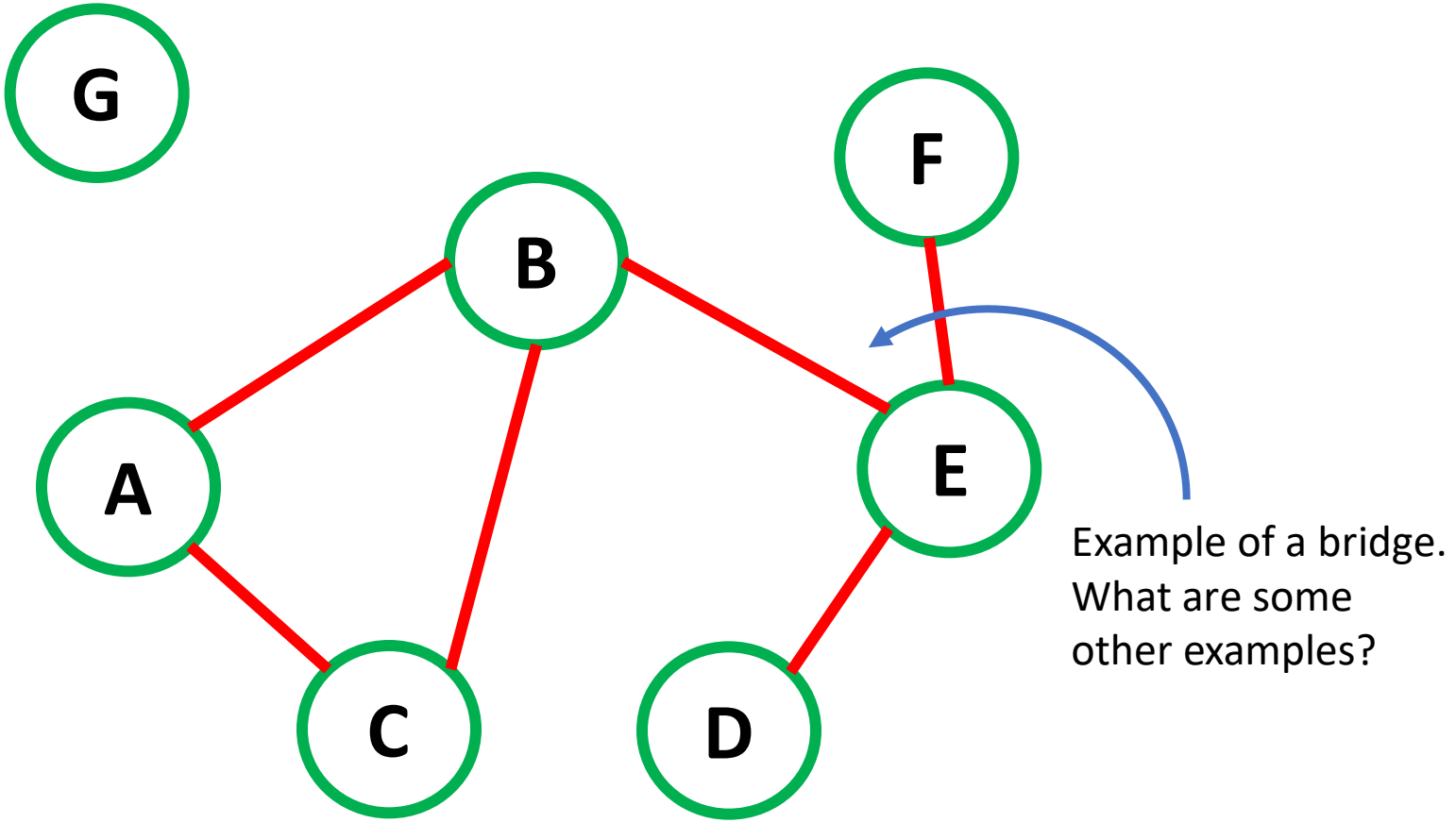
Silver Problem: Finding if an edge is a bridge

- An edge is a bridge if it is the only link between two regions of a graph.



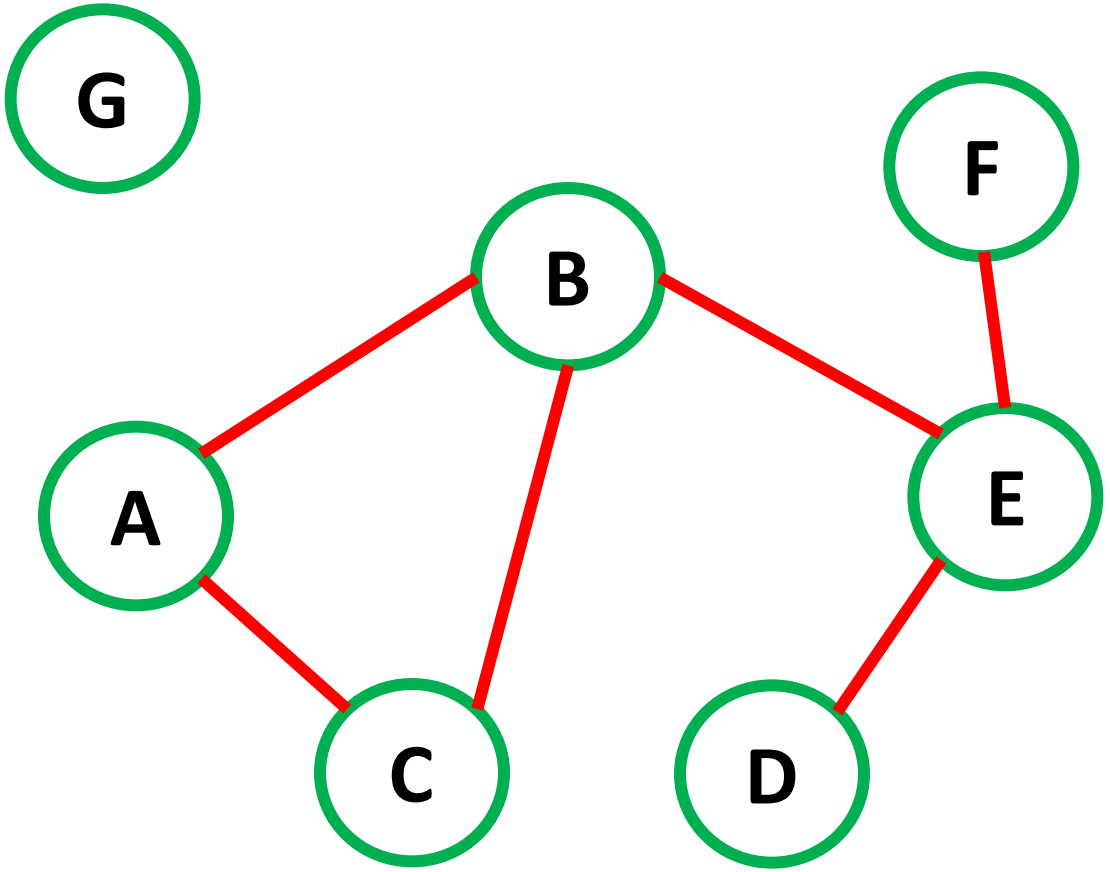
Silver Problem: Finding if an edge is a bridge

- An edge is a bridge if it is the only link between two regions of a graph.



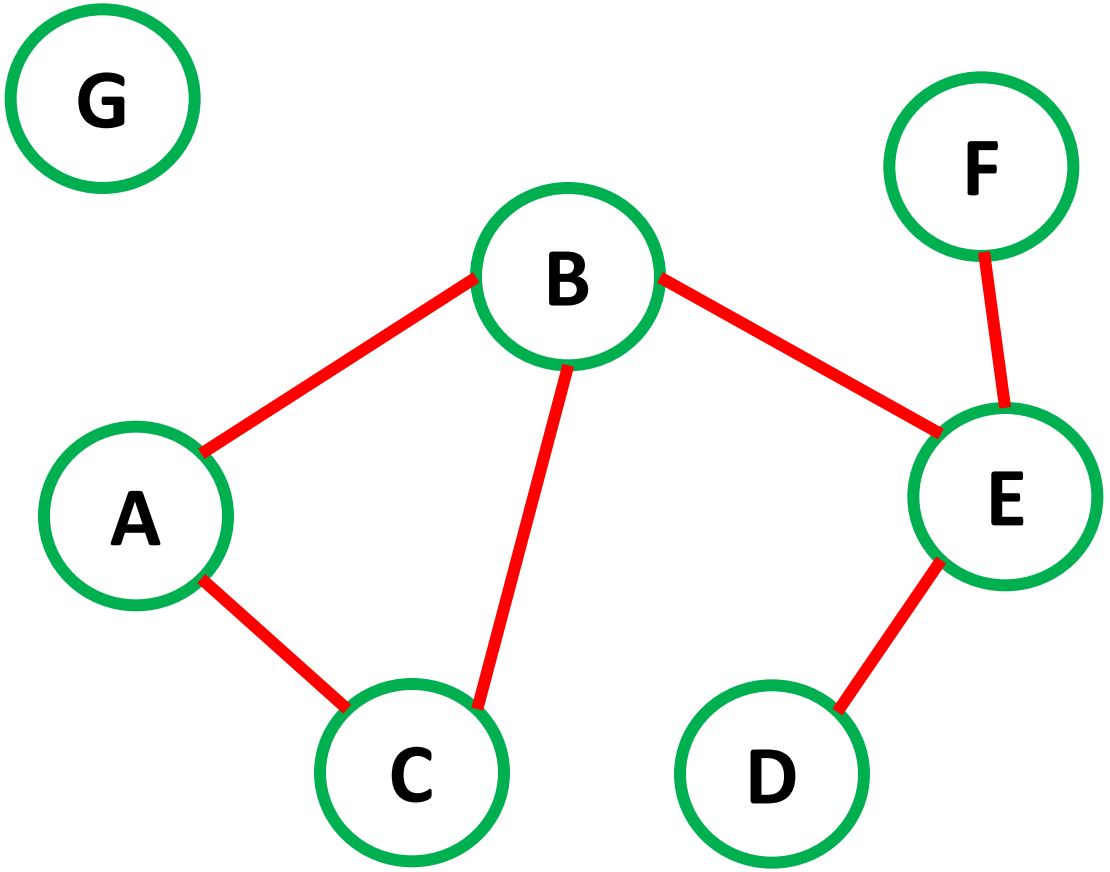
Identifying if an edge is a bridge

- How many components in the graph below?



Identifying if an edge is a bridge

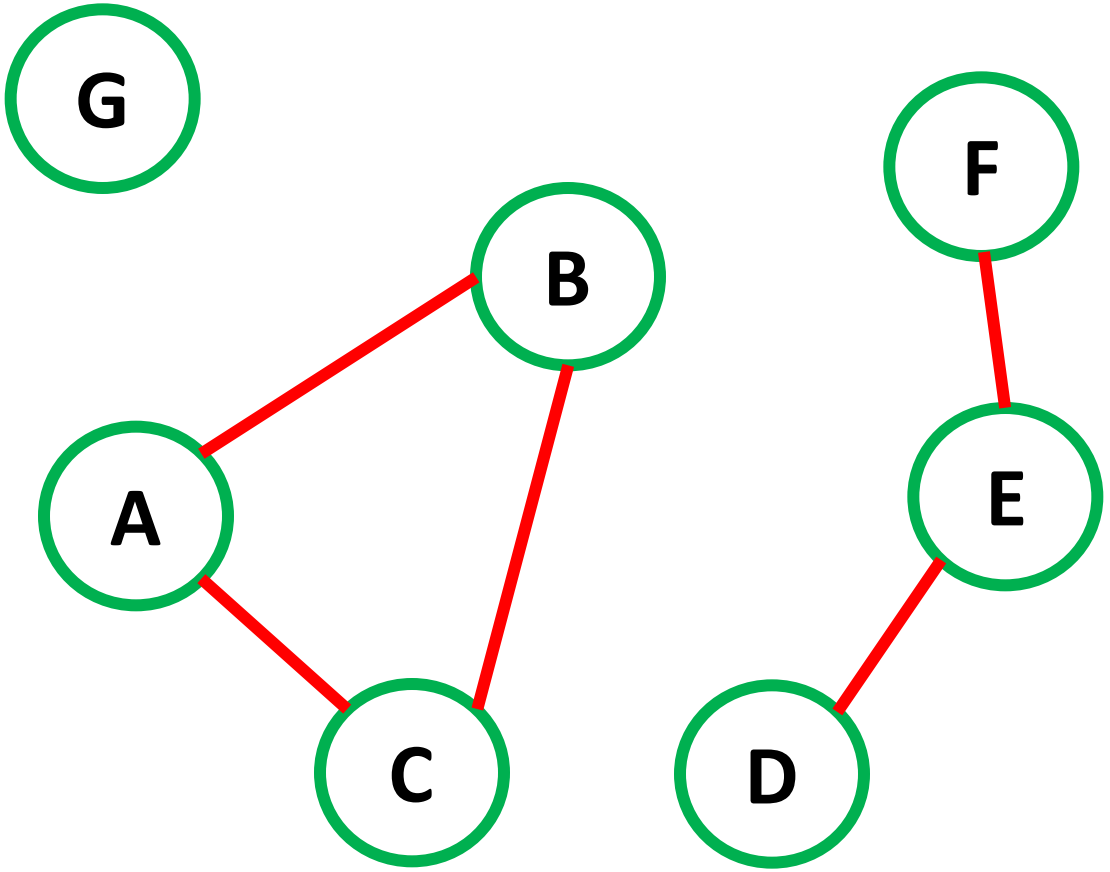
- How many components in the graph below?



components = 2

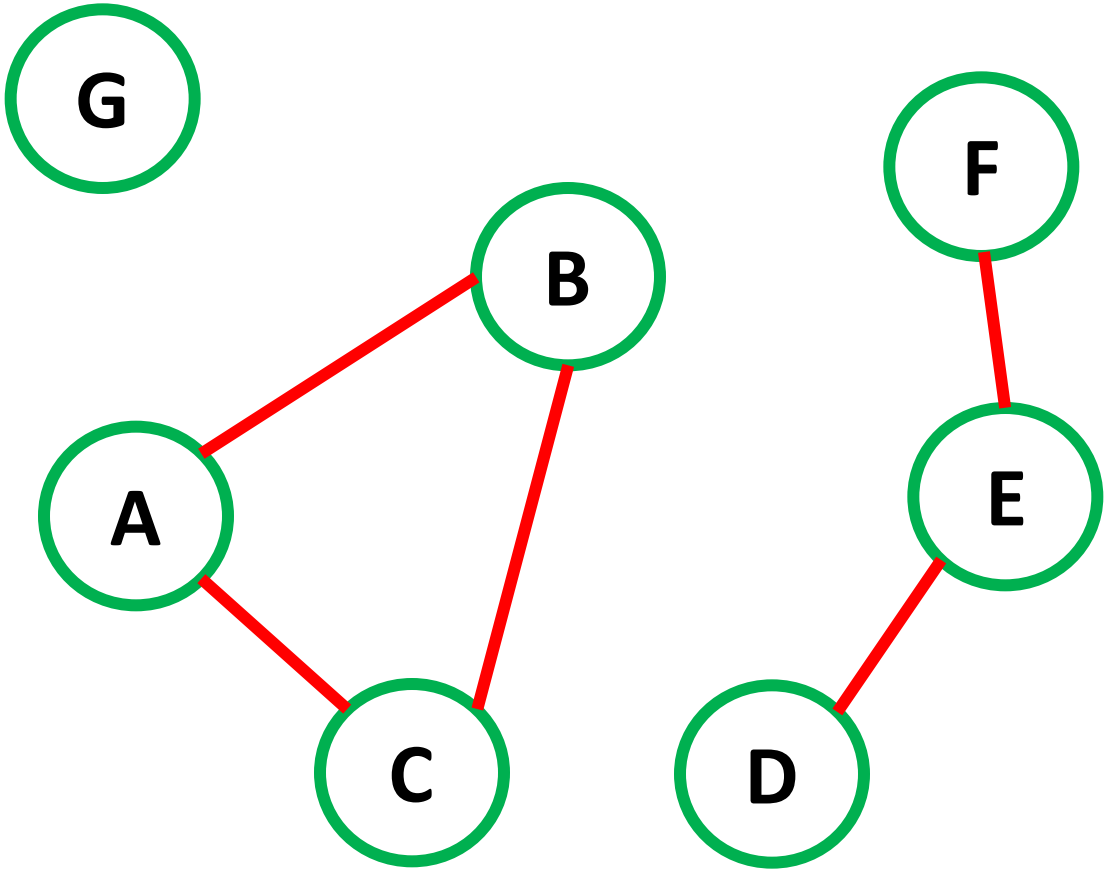
Identifying if an edge is a bridge

- How many components in the graph after removing the edge **BE**?



Identifying if an edge is a bridge

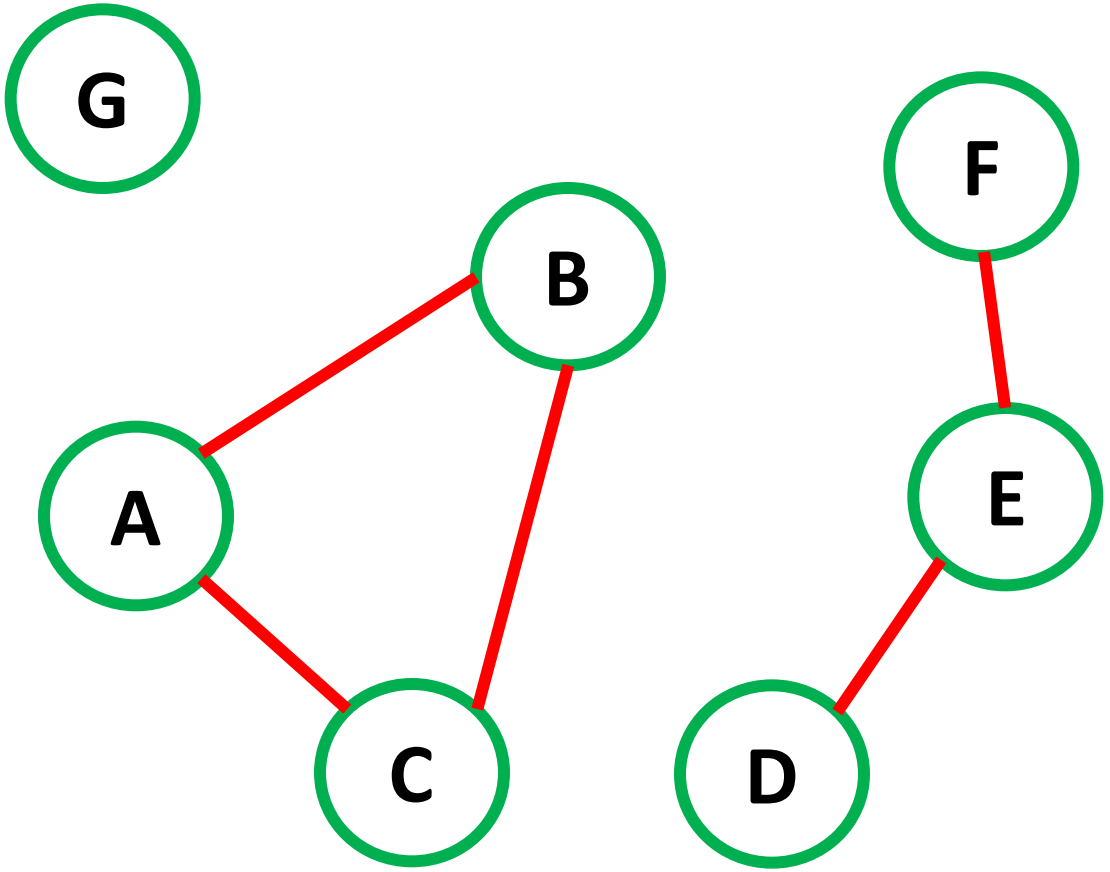
- How many components in the graph after removing the edge **BE**?



components = 3

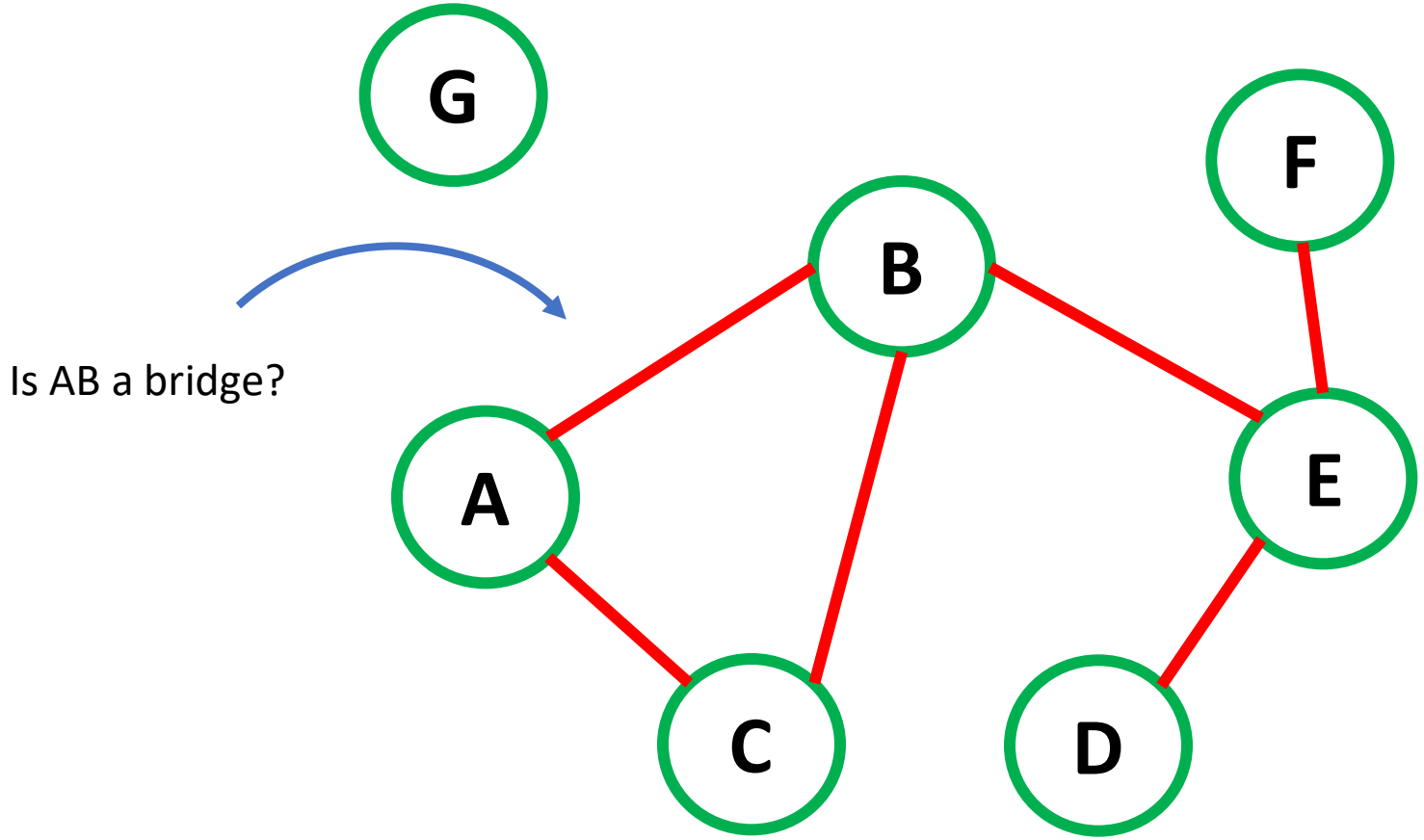
Identifying if an edge is a bridge

- Answer: If the number of components increases by 1 after removing an edge.



components = 3

Identifying if an edge is a bridge



Silver Problem: Implementation

- Step 1: Implement the DFS function as shown in the slides above.
- Hints
  - Access the visited member of vertex indexed by x: `n->adj[x].v->visited`
  - Remember that DFS is recursive; call it recursively: `DFTraversal(n->adj[x].v)`

```
68  // DFS
69  // TODO: Complete this function, so that it can be used in the isBridge function
70  void Graph::DFTraversal(vertex *n)
71  {
72      n->visited = true;
73
74      for(int x = 0; x < n->adj.size(); x++ )
75      {
76          // TODO
77      }
78
79  }
```

Silver Problem: Implementation

- Step 2: Complete the removeEdge function.
- Hints
  - Pointers to vertices and v1 and v2 are available.
  - In a `for` loop, get index `x1` for which `v1->adj[x1].v == v2`.
  - Delete the entry `x1` in `v1->adj`: `v1->adj.erase(v1->adj.begin() + x1)`
  - Do the same for the reverse case to delete `v1` in adj list of `v2`.

```
101    // TODO: Modify the adjacency lists (remember that the graph is undirected)
102    // You can use erase to remove an item from a vector
103    // Ex: say you need to remove ith index item from a vector "myvector",
104    // you can use myvector.erase(myvector.begin()+i)
105
```

Silver Problem: Implementation

- Step 3: Compute the connected components before and after removal.
- Hints:
  - Refer to the **`ConnectedComponents()`** pseudocode on Slide 22.
  - For each vertex, if it is unvisited:
    - Recursively call DFTraversal on the vertex.
    - And increment **`initial_components/components_after_removal.`**

```
// no. of connected componenets in the graph before removing the edge
int initial_components = 0;

// TODO Step1: Get initial_components. Complete and use the DFTraversal function.
```

```
// no. of connected componenets in the modified graph (i.e after removing the edge)
int components_after_removal = 0;

// TODO Step 3: Get components_after_removal (use DFTraversal again)

cout<< "no. of connected components after removal: " <<components_after_removal << endl
```

Silver Problem: Implementation

- Step 4: Check if the number of connected components increases after removing an edge.

```
// TODO Step5: check if the no of connected componenets
// increases after removing the edge and return true
```